Creating XPCOM Components

Doug Turner & Ian Oeschger

Creating XPCOM Components

Published by Brownhen Publishing.

Preface

This is a book about Gecko, and about creating XPCOM components for Gecko-based applications. Though the emphasis is on the practical steps you take to make your C++ code into a component that can be used in Gecko, we hope that these steps will also give us an occasion to discuss all of the tools, techniques, and technologies that make up XPCOM. Accordingly, the book is arranged so that you can follow along and create your own components or learn about different XPCOM topics individually, as in a reference work. For example, the introduction includes a discussion of components and what they are, and the first chapter—in which you compile the basic code and register it with Mozilla—prompts a discussion of the relationship between components and modules, of XPCOM interfaces, and of the registration process in general.

The top of each chapter provides a list of the major topics covered. Sidebar sections are included to highlight technical details. By the end of the book, if we've done our job, you will have leaned how to build a component and you will know something about the framework for these components in Gecko, which is XPCOM.

Who Should Read This Book

Creating XPCOM Components is meant for C++ developers. Though you can create XPCOM components in JavaScript and other languages, and though you might be able to follow along as a C programmer, the component implementation code is written in C++, and much of the discussion of how to make your code *into* an XPCOM component starts from C++. You don't need to be a C++ expert, however. Although basic ideas such as inheritance and encapsulation should be familar to you, wherever possible they are explained in the book as they are used. Also many of the examples are in JavaScript, which is used in Mozilla to access XPCOM components as scriptable objects, and so familiarity with that language is useful as well.

XPCOM stands for the Cross Platform Component Object Model. As this name implies, XPCOM is similar to Microsoft COM. If you have any experience with this technology, much of it can be applied to XPCOM. However, this book does not assume any prior knowledge of COM—all of the basic COM ideas will be introduced.

This book provides a tutorial about building an XPCOM component that controls browsing behavior. Although, XPCOM can be used in many environments which are unrelated to web browsing, XPCOM's main client is *Gecko*, an open source, standards compliant, embeddable web browser, where it's easiest and most practical to illustrate XPCOM's functionality. A full description of the component in this tutorial can be found in the "What We'll Be Working On" section of the tutorial.

Organization of the Tutorial

The following list provides an overview of the steps that we will take to build an XPCOM component called **WebLock**, which provides site blocking functionality to Gecko-based browsers. Each one of these steps has its own chapter, in which a number of topics associated with the step are discussed.

- Create the generic module code for the component.
- Use C++ macros, special string classes and smart pointers to optimize your code.
- Define the functionality for the component; create an XPIDL interface for that functionality; create the implementation code specific to the custom **WebLock** component interface.
- Finish implementing the **WebLock** component: nsIContentPolicy, file I/O, locking, etc.
- Creating the user interface for the **WebLock** component.
- Packaging **WebLock** for distribution and installation.

Following Along with the Examples

There are a couple of different ways to get XPCOM onto your machine so you can begin to create components. If you already have a Mozilla build or the source from Mozilla 1.2 or later, then you can use the XPCOM framework available there. If you don't already have the Mozilla source, then an even easier way to get and use XPCOM is to download the *Gecko SDK*, which is a collection of libraries and tools that features the XPCOM component framework.

Whether you compile your code in the Mozilla source tree or use the Gecko SDK, you can build your own components and which leverage components that already exist in Gecko. The **WebLock** component we describe in this tutorial is a practical (and, we hope, genuinely useful) addition to the browser. In order to build it, your Gecko SDK or Mozilla source tree needs to be *version 1.2 or later*. Releases prior to Mozilla 1.2 did not completely support XPCOM components with frozen interfaces, but changes since then have remedied this.

This book assumes you are using the SDK rather than compiling in a Mozilla source tree, though the difference between these two approaches is minimal. Details about downloading the SDK, building, and getting programmatic access to Gecko components are provided in the appendix to this book, *Setting up the Gecko Platform*.

Conventions

The formatting conventions listed in Table 1 are used to designate specific types of information in the book and make things easier to scan. The goal is to use as few formats as possible, but to distinguish the various different types of information clearly.

TABLE 1. **Formatting Conventions Used in This Book**

Format	**Description**
bold	**component names** appear in bold in the text
`monospace`	`code listings, interface names` and `members` of interfaces (e.g., `createInstance()`) appear in monospaced font. Longer code listings appear in gray boxes.
italic	*variables, filenames* and *directory names* terms appear in italic. Important terms and new concepts are also italicized the first time they appear in the text. Those terms are either explained immediately after they are cited, or else the reader is referred to a section in the book where they are described in detail. References to other chapters (e.g., *Getting Social: Making Instances of Your Component Available*) are also italicized.
“quoted”	References to other sections are double-quoted (e.g. “Organization of the Tutorial”).

Acknowledgements

Thanks to Peter Lubczynski, John Gaunt, Ellen Evans, and Alec Flett for technical reviews. And a special thanks to Darin Fisher for his very acute observations, close reading, and attention to detail.

CHAPTER 1

What Is XPCOM?

This is a book about XPCOM. The book is written in the form of a tutorial about creating XPCOM components, but it covers all major aspects, concepts, and terminology of the XPCOM component model along the way.

This chapter starts with a quick tour of XPCOM—an introduction to the basic concepts and technologies in XPCOM and component development. The brief sections in this chapter introduce the concepts at a very high-level, so that we can discuss and use them with more familiarity in the tutorial itself, which describes the creation of a Mozilla component called **WebLock**.

- "The XPCOM Solution"
- "Components"
- "Interfaces"
- "Factories"
- "XPCOM Services"

The XPCOM Solution

The Cross Platform Component Object Module (XPCOM) is a framework which allows developers to break up monolitic software projects into smaller modularized pieces. These pieces, known as *components*, are then assembled back together at runtime.

The goal of XPCOM is to allow different pieces of software to be developed and built independently of one another. In order to allow interoperability between components within an application, XPCOM separates the *implementation* of a component from the *interface*, which we discuss in the “Interfaces” section. But XPCOM also provides several tools and libraries that enable the loading and manipulation of these components, services that help the developer write modular cross-platform code, and versioning support, so that components can be replaced or upgraded without breaking or having to recreate the application. Using XPCOM, developers create components that can be reused in different applications or that can be replaced to change the functionality of existing applications.

XPCOM not only supports component software development, it also provides much of the functionality that a development platform provides, such as:

- component management
- file abstraction
- object message passing
- memory management

We will discuss the above items in detail in the coming chapters, but for now, it can be useful to think of XPCOM as a *platform for component development*, in which features such as those listed above are provided.

Gecko

Although it is in some ways structurally similar to Microsoft COM, XPCOM is designed to be used principally at the application level. The most important use of XPCOM is within *Gecko*, an open source, standards compliant, embeddable web browser and toolkit for creating web browsers and other applications.

XPCOM is the means of accessing Gecko library functionality and embedding or extending Gecko. This book focuses on the latter—extending Gecko—but the fundamental ideas in the book will be important to developers embedding Gecko as well.

Gecko is used in many internet applications, mostly browsers. The list includes devices such as the Gateway/AOL Instant AOL device and the Nokia Media Terminal. Gecko is also used in the latest Compuserve client, AOL for Mac OS X, Netscape 7, and of course the Mozilla client. At this time, Gecko is the predominant open source web browser.

Components

XPCOM allows you to build a system in which large software projects can be broken up into smaller pieces. These pieces, known as components, are usually delivered in small, reusable binary libraries (a DLL on Windows, for example, or a DSO on Unix), which can include one or more components. When there are two or more related components together in a binary library, the library is referred to as a *module*.

Breaking software into different components can help make it less difficult to develop and maintain. Beyond this, modular, component-based programming has some well-known advantages, as Table 1 describes:

TABLE 1. Benefits from Modular Code

Benefit	**Description**
Reuse	Modular code can be reused in other applications and other contexts
Updates	You can update components without having to recompile the whole application
Performance	When code is modularized, modules that are not necessary right away can be “lazy loaded”, or not loaded at all, which can improve the performance of your application.
Maintenance	Even when you are not updating a component, designing your appication in a modular way can make it easier for you to find and maintain the parts of the application that you are interested in.

Mozilla has over four million lines of code, and no single individual understands the entire codebase. The best way to tackle a project of this size is to divide it into smaller, more managable pieces, use a component programming model, and to organize related sets of components into modules. The network library, for example, consists of components for each of the protocols, HTTP, FTP, and others, which are bundled together and linked into a single library. This library is the networking module, also known as "necko."

But it's not always a good idea to divide things up. There are some things in the world that just go together, and others that shouldn't be apart. For example, one author's son will not eat a peanutbutter sandwich if there isn't jam on it, because in his world, peanut butter and jam form an indelible union. Some software is the same. In areas of code that are tightly-coupled—in classes that are only used internally, for example—the expensive work to divide things may not be worth the effort.

The HTTP component in Gecko doesn't expose private classes it uses as separate components. The "stuff" that's internal to the component stays internal, and isn't exposed to XPCOM. In the haste of early Mozilla development, components were created where they were inappropriate, but there's been an ongoing effort to remove XPCOM from places like this.

Interfaces

It's generally a good idea to break software into components, but how exactly do you do this? The basic idea is to identify the pieces of functionality that are related and understand how they communicate with each other. The communication channels between different component form boundaries between those components, and when those boundaries are formalized they are known as *interfaces*.

Interfaces aren't a new idea in programming. We've all used interfaces since our first "HelloWorld" program, where the interface was between the code we actually wrote—the application code—and the printing code. The application code used an interface from a library, `stdio`, to print the "hello world" string out to the screen. The difference here is that a "HelloWorld" application in XPCOM finds this screen-printing functionality at runtime and never has to know about `stdio` when it's compiled.

Interfaces allow developers to *encapsulate* the implementation and inner workings of their software, and allow clients to ignore how things are made and just use that software.

Interfaces and Programming by Contract

An interface forms a contractual agreement between components and clients. There is no code that enforces these agreements, but ignoring them can be fatal. In component-based programming, a component guarantees that the interfaces it provides will be *immutable*—that they will provide the same access to the same methods across different versions of the component—establishing a contract with the software clients that use it. In this respect, interface-based programming is often referred to as *programming by contract*.

Interfaces and Encapsulation

Between component boundaries, abstraction is crucial for software maintainability and reusability. Consider, for example, a class that *isn't* well encapsulated. Using a freely available public initialization method, as the example below suggests, can cause problems.

```
class SomeClass
{
  public:
    // Constructor
    SomeClass();

    // Virtual Destructor
    virtual ~SomeClass();

    // init method
    void Init();

    void DoSomethingUseful();
};
```

Figure 1. SomeClass Class Initialization

For this system to work properly, the client programmer must pay close attention to whatever rules the component programmer has established. This is the contractual agreement of this unencapsulated class: a set of rules that define when each method can be called and what it is expected to do. One rule might specify that `DoSomethingUseful` may only be called after a call to `Init()`. The `DoSomethingUseful` method may do some kind of checking to ensure that the condition—that `Init` has been called—has been satisfied.

In addition to writing well-commented code that tells the client developer the rules about `Init()`, the developer can take a couple steps to make this contract even clearer. First, the construction of an object can be encapsulated, and a *virtual class* provided that defines the `DoSomethingUseful` method. In this way, construction and initialization can be completely hidden from clients of the class. In this “semi-encapsulated” situation, the only part of the class that is exposed is a well-defined list of callable methods (i.e., the interface). Once the class is encapsulated, the only interface the client will see is this:

```
class SomeInterface
{
public:
  virtual void DoSomethingUseful() = 0;
};
```

Figure 2. Encapsulation of SomeInterface

The implementation can then derive from this class and implement the virtual method. Clients of this code can then use a factory design pattern to create the object (see “Factories” on page 20) and further encapsulate the implementation. In XPCOM, clients are shielded from the inner workings of components in this way and rely on the interface to provide access to the needed functionality.

The nsISupports Base Interface

Two fundamental issues in component and interface-based programming are *component lifetime*, also called *object ownership*, and *interface querying*, or being able to identify which interfaces a component supports at run-time. This section introduces the base interface—the mother of all interfaces in XPCOM—`nsISupports`, which provides solutions to both of these issues for XPCOM developers.

Object Ownership. In XPCOM, since components may implement any number of different interfaces, interfaces must be *reference counted*. Components must keep track of how many references to it clients are maintaining and delete themselves when that number reaches zero.

When a component gets created, an integer inside the component tracks this reference count. The reference count is incremented automatically when the client instantiates the component; over the course of the component's life, the reference count goes up and down, always staying above zero. At some point, all clients lose interest in the component, the reference count hits zero, and the component deletes itself.

When clients use interfaces responsibly, this can be a very straightforward process. XPCOM has tools to make it even easier, as we describe later. It can raise some real housekeeping problems when, for example, a client uses an interface and forgets to decrement the reference count. When this happens, interfaces may never be released and will leak memory. The system of reference counting is, like many things in XPCOM, a contract between clients and implementations. It works when people agree to it, but when they don't, things can go wrong. It is the responsibility of the function that creates the interface pointer to add the initial reference, or *owning reference*, to the count.

Pointers in XPCOM

In XPCOM, *pointers* refer to interface pointers. The difference is a subtle one, since interface pointers and regular pointers are both just address in memory. But an interface pointer is known to implement the `nsISupports` base interface, and so can be used to call methods such as `AddRef`, `Release`, or `QueryInterface`.

`nsISupports`, shown below, supplies the basic functionality for dealing with interface discovery and reference counting. The members of this interface, `QueryInterface`, `AddRef`, and `Release`, provide the basic means for getting the right interface from an object, incrementing the reference count, and releasing objects once they are not being used, respectively. *Figure 3* shows the `nsISupports` interface.

```
class Sample: public nsISupports {
private:
  nsrefcnt mRefCnt;
public:
  Sample();
  virtual ~Sample();

  NS_IMETHOD QueryInterface(const nsIID &aIID, void **aResult);
  NS_IMETHOD_(nsrefcnt) AddRef(void);
  NS_IMETHOD_(nsrefcnt) Release(void);

};
```

Figure 3. The nsISupports Interface

The various types used in this figure are described in the "XPCOM Types" section below. *Figure 4* shows a complete (if spare) implementation of the `nsISupports` interface.

```
Sample::Sample()
{
  // initialize the reference count to 0
  mRefCnt = 0;
}
Sample::~Sample()
{
}

// typical, generic implementation of QI
NS_IMETHODIMP Sample::QueryInterface(const nsIID &aIID,
                                  void **aResult)
{
  if (aResult == NULL) {
    return NS_ERROR_NULL_POINTER;
  }
  *aResult = NULL;
  if (aIID.Equals(kISupportsIID)) {
    *aResult = (void *) this;
  }
  if (*aResult != NULL) {
    return NS_ERROR_NO_INTERFACE;
  }
  // add a reference
  AddRef();
  return NS_OK;
}

NS_IMETHODIMP_(nsrefcnt) Sample::AddRef()
{
  return ++mRefCnt;
}

NS_IMETHODIMP_(nsrefcnt) Sample::Release()
{
  if (--mRefCnt == 0) {
    delete this;
    return 0;
  }
  // optional: return the reference count
  return mRefCnt;
}
```

Figure 4. Implementation of nsISupports Interface

Object Interface Discovery. *Inheritance* is another very important topic in object oriented programming. Inheritance is the means through which one class is derived from another. When a class inherits from another class, the inheriting class may *override* the default behaviors of the base class without having to copy all of that class's code, in effect creating a more specific class, as in the following example:

```
class Shape
{
private:
  int m_x;
  int m_y;

public:
  virtual void Draw() = 0;
  Shape();
  virtual ~Shape();
};


class Circle : public Shape
{
private:
   int m_radius;
public:
   virtual Draw();
   Circle(int x, int y, int radius);
   virtual ~Circle();
};
```

Figure 5. `Simple Class Inheritance`

`Circle` is a derived class of `Shape`. A `Circle` is a `Shape`, in other words, but a `Shape` is not necessarily a `Circle`. In this case, Shape is the *base class* and Circle is a *subclass* of Shape.

In XPCOM, all classes derive from the `nsISupports` interface, so all objects are `nsISupports` but they are also other, more specific classes, which you need to be able to find out about at runtime. In *Figure 5* above, for example, you'd like to be able ask the `Shape` if it's a `Circle` and to be able to use it like a circle if it is. In XPCOM, this is what the `QueryInterface` feature of the `nsISupports` interface is for: it allows clients to find and access different interfaces based on their needs.

In C++, you can use a fairly advanced feature known as a `dynamic_cast<>`, which throws an exception if the `Shape` object is not able to be cast to a `Circle`. But enabling exceptions and RTTI may not be an option because of performance overhead and compatibility on many platforms, so XPCOM does things differently.

“Exceptions” in XPCOM

C++ exceptions are not supported directly by XPCOM. Instead all exceptions must be handled within a given component, before crossing interface boundaries. In XPCOM, all interface methods should return an `nsresult` error value (see the XPCOM API in Appendix B for a listing of these error codes). These error code results become the “exceptions” that XPCOM handles.

Instead of leveraging C++ RTTI, XPCOM uses the special `QueryInterface` method that casts the object to the right interface if that interface is supported.

Every interface is assigned an identifier that gets generated from a tool commonly named “uuidgen”. This universally unique identifier (UUID) is a unique, 128 bit number. Used in the context of an interface (as opposed to a component, which is what the contract ID is for), this number is called an *IID*.

When a client wants to discover if an object supports a given interface, the client passes the IID assigned to that interface into the `QueryInterface` method of that object. If the object supports the requested interface, it adds a reference to itself and passes back a pointer to that interface. If the object does not support the interface an error is returned.

```
class nsISupports {
   public:
      long QueryInterface(const nsIID & uuid,
                 void **result) = 0;
      long AddRef(void) = 0;
      long Release(void) = 0;
};
```

The first parameter of `QueryInterface` is a reference to a class named `nsIID`, which is a basic encapsulation of the IID. Of the three methods on the `nsIID` class, `Equals`, `Parse`, and `ToString`, `Equals` is by far the most important, because it is used to compare two `nsIID`s in this interface querying process.

When you implement the `nsIID` class (and you'll see in the chapter "Tutorial: Using XPCOM Utilities To Make Things Easier" how macros can make this process much easier), you must make sure the class methods return a valid result when the client calls `QueryInterface` with the `nsISupports` IID. `QueryInterface` should support all interfaces that the component supports.

In implementations of `QueryInterface`, the IID argument is checked against the `nsIID` class. If there is a match, the object's `this` pointer is cast to `void`, the reference count is incremented, and the interface returned to the caller. If there isn't a match, the class returns an error and sets the out value to `null`.

In the example above, it's easy enough to use a C-style cast. But casting can become more involved where you must first cast voic then to the requested type, because you must return the interface pointer in the vtable corresponding to the requested interface. Casting can become a problem when there is an ambiguous inheritance hierarchy.

XPCOM Identifiers

In addition to the IID interface identifier discussed in the previous section, XPCOM uses two other very important identifiers to distinguish classes and components.

- "CID"
- "Contract ID"

CID

A CID is a 128 bit number that uniquely identifies a class or component in much the same way that an IID uniquely identifies an interface. The CID for `nsISupports` looks like this:

```
00000000-0000-0000-c000-000000000046
```

The length of a CID can make it cumbersome to deal with in the code, so very often you see #defines for CIDs and other identifiers being used, as in this example:

```
#define SAMPLE_CID \
{ 0x777f7150, 0x4a2b, 0x4301, \
{ 0xad, 0x10, 0x5e, 0xab, 0x25, 0xb3, 0x22, 0xaa}}
```

You also see `NS_DEFINE_CID` used a lot. This simple macro declares a constant with the value of the CID:

```
static NS_DEFINE_CID(kWebShellCID, NS_WEB_SHELL_CID);
```

A CID is sometimes also referred to as a *class identifier*. If the class to which a CID refers implements more than one interface, that CID guarantees that the class implements that whole set of interfaces when it's published or frozen.

Contract ID

A contract ID is a human readable string used to access a component. A CID or a contract ID may be used to get a component from the component manager. This is the contract ID for the LDAP Operation component:

```
"@mozilla.org/network/ldap-operation;1"
```

The format of the contract ID is the *domain* of the component, the *module*, the *component name*, and the *version number*, separated by slashes.

Like a CID, a contract ID refers to an implementation rather than an interface, as an IID does. But a contract ID is not bound to any specific implementation, as the CID is, and is thus more general. Instead, a contract ID only specifies a given set of interfaces that it wants implemented, and any number of different CIDs may step in and fill that request. This difference between a contract ID and a CID is what makes it possible to override components.

XPCOM Identifier Classes

The `nsIID` class is actually a typedef for the `nsID` class. The other typedefs of `nsID`, CID and IID, refer to specific implementations of a concrete class and to a specific interface, respectively.

The nsID class provides methods like Equals for comparing identifiers in the code. See “Identifiers in XPCOM” on page 60 for more discussion of the `nsID` classes.

Factories

Once code is broken up into components, client code typically uses the `new` constructor to instantiate objects for use:

```
SomeClass* component = new SomeClass();
```

This pattern requires that the client know something about the component, however—how big it is at the very least. The *factory design pattern* can be used to encapsulate object construction. The goal of factories is create objects without exposing clients to the implementations and initializations of those objects. In the `SomeClass` example, the construction and initialization of `SomeClass`, which implements the `SomeInterface` abstract class, is contained within the `New_SomeInterface` function, which follows the factory design pattern:

```
int New_SomeInterface(SomeInterface** ret)
{
  // create the object
  SomeClass* out = new SomeClass();
  if (!out) return -1;

  // init the object
  if (out->Init() == FALSE)
  {
    delete out;
    return -1;
  }

  // cast to the interface
  *ret = static_cast<SomeInterface*>(out);
  return 0;
}
```

Figure 6. Encapsulating the Constructor

The factory is the class that actually manages the creation of separate instances of a component for use. In XPCOM, factories are implementations of the `nsIFactory` interface, and they use a factory design pattern like the example above to abstract and encapsulate object construction and initialization.

The example in *Figure 6* above is a simple and stateless version of factories, but real world programming isn't usually so simple, and in general factories need to store state. At a minimum, the factory needs to preserve information about what objects it has created. When a factory manages instances of a class built in a dynamic shared library, for example, it needs to know when it can unload the library. When the factory preserves state, you can ask if there are outstanding references and find out if the factory created any objects.

Another state that a factory can save is whether or not an object is a *singleton*. For example, if a factory creates an object that is supposed to be a singleton, then subsequent calls to the factory for the object should return the same object. Though there are tools and better ways to handle singletons (which we'll discuss when we talk about the `nsIServiceManager`), a developer may want to use this information to ensure that only one singleton object can exist despite what the callers do.

The requirements of a factory class can be handled in a strictly functional way, with state being held by global variables, but there are benefits to using classes for factories. When you use a class to implement the functionality of a factory, for example, you derive from the `nsISupports` interface, which allows you to manage the lifetime of the factory objects themselves. This is important when you want to group sets of factories together and determine if they can be unloaded. Another benefit of using the `nsISupports` interface is that you can support other interfaces as they are introduced. As we'll show when we discuss `nsIClassInfo`, some factories support querying information about the underlying implementation, such as what language the object is written in, interfaces that the object supports, etc. This kind of "future-proofing" is a key advantage that comes along with deriving from `nsISupports`.

XPIDL and Type Libraries

An easy and powerful way to define an interface—indeed, a requirement for defining interfaces in a cross-platform, language neutral development environment—is to use an *interface definition language* (IDL). XPCOM uses its own variant of the CORBA OMG Interface Definition Language (IDL) called XPIDL, which allows you to specify methods, attributes and constants of a given interface, and also to define interface inheritence.

There are some drawbacks to defining your interface using XPIDL. There is no support for multiple inheritence, for one thing. If you define a new interface, it cannot derive from more than one interface. Another limitation of interfaces in XPIDL is that method names must be unique. You can not have two methods with the same name that take different parameters, and the workaround—having multiple function names—isn't pretty:

```
void FooWithInt(in int x);
void FooWithString(in string x);
void FooWithURI(in nsIURI x).
```

However, these shortcomings pale in comparison to the functionality gained by using XPIDL. XPIDL allows you to generate *type libraries*, or typelibs, which are files with the extension *.xpt*. The type library is a binary representation of an interface or interfaces. It provides programmatic control and access of the interface, which is crucial for interfaces used in the non C++ world. When components are accessed from other languages, as they can be in XPCOM, they use the binary type library to access the interface, learn what methods it supports, and call those methods. This aspect of XPCOM is called *XPConnect*. XPConnect is the layer of XPCOM that provides access to XPCOM components from languages such as JavaScript. See "Connecting to Components from the Interface" on page 30 for more information about XPConnect.

When a component is accessible from a language other than C++, such as JavaScript, its interface is said to be "reflected" into that language. Every reflected interface must have a corresponding type library. Currently you can write components in C, C++, JavaScript, or Python, and there are efforts underway to build XPCOM bindings for Ruby and Perl as well.

Writing Components in Other Languages

Though you do not have access to some of the tools that XPCOM provides for C++ developers (such as macros, templates, smart pointers, and others) when you create components in other languages, you may be so comfortable with the language itself that you can eschew C++ altogether and build, for example, Python-based XPCOM components that can be used from JavaScript or vice versa.

See the "References" section in Appendix C for more information about Python and other languages for which support has been added in XPCOM.

All of the public interfaces in XPCOM are defined using the XPIDL syntax. Type libraries and C++ header files are generated from these IDL files, and the tool that generates these files is called the *xpidl compiler.* The section "Defining the Weblock Interface in XPIDL" on page 103 describes the XPIDL syntax in detail.

XPCOM Services

When clients use components, they typically *instantiate* a new object each time they need the functionality the component provides. This is the case when, for example, clients deal with files: each separate file is represented by a different object, and several file objects may be being used at any one time.

But there is also a kind of object known as a *service*, of which there is always only one copy (though there may be many services running at any one time). Each time a client wants to access the functionality provided by a service, they talk to the same instance of that service. When a user looks up a phone number in a company database, for example, probably that database is being represented by an "object" that is the same for all co-workers. If it weren't, the application would need to keep two copies of a large database in memory, for one thing, and there might also be inconsistencies between records as the copies diverged.

Providing this single point of access to functionality is what the singleton design pattern is for, and what services do in an application (and in a development environment like XPCOM).

In XPCOM, in addition to the component support and management, there are a number of services that help the developer write cross platform components. These services include a cross platform file abstraction which provides uniform and powerful access to files, directory services which maintain the location of application- and system-specific locations, memory management to ensure everyone uses the same memory allocator, and an event notification system that allows passing of simple messages. The tutorial will show each of these component and services in use, and *Appendix B* has a complete interface listing of these areas.

XPCOM Types

There are many XPCOM declared types and simple macros that we will use in the following samples. Most of these types are simple mappings. The most common types are described in the following sections:

- “Method Types”
- “Reference Counting”
- “Status Codes”
- “Variable mappings”
- “Common XPCOM Error Codes”

Method Types

The following are a set of types for ensuring correct calling convention and return time of XPCOM methods.

`NS_IMETHOD`	Method declaration return type. XPCOM method declarations should use this as their return type.
`NS_IMETHODIMP`	Method Implementation return type. XPCOM method implementations should use this as their return time.
`NS_IMETHODIMP_(type)`	Special case implementation return type. Some methods such as AddRef and Release do not return the default return type. This exception is regretta-ble, but required for COM compliance.
`NS_IMPORT`	Forced the method to be resolved internally by the shared library.
`NS_EXPORT`	Forces the method to be exported by the shared library.

Reference Counting

Set of macros for managing reference counting.

NS_ADDREF	Calls AddRef on an nsISupports object
NS_IF_ADDREF	Same as above but checks for null before calling AddRef
NS_RELEASE	Calls Release on an nsISupports object
NS_IF_RELEASE	Same as above but check for null before calling Release

Status Codes

These macros test status codes

NS_FAILED	Return true if the passed status code was a failure.
NS_SUCCEEDED	Returns true is the passed status code was a success.

Variable mappings

nsrefcnt	Default reference count type. Maps to an 32 bit integer.
nsresult	Default error type. Maps to a 32 bit integer.
nsnull	Default null value.

Common XPCOM Error Codes

NS_ERROR_NOT_INITIALIZED	Returned when an instance is not initialized.
NS_ERROR_ALREADY_INITIALIZED	Returned when an instance is already initialized
NS_ERROR_NOT_IMPLEMENTED	Returned by an unimplemented method
NS_ERROR_NO_INTERFACE	Returned when a given interface is not supported.
NS_ERROR_NULL_POINTER	Returned when a valid pointer is found to be nsnull.

`NS_ERROR_FAILURE`	Returned when a method fails. Generic error case.
`NS_ERROR_UNEXPECTED`	Returned when an unexpected error occurs.
`NS_ERROR_OUT_OF_MEMORY`	Returned when a memory allocation fails.
`NS_ERROR_FACTORY_NOT_REGISTERED`	Returned when a requested class is not registered.

CHAPTER 2

Using XPCOM Components

One of the best ways to begin working with XPCOM—especially when you are designing the interface to a component that will be used by others, as we do in the chapter “Tutorial: Starting WebLock”—is to look at how clients are already using XPCOM components.

Applications like the Mozilla browser are sophisticated, modularized clients of XPCOM components. In fact, virtually all of the functionality that you associate with a browser—navigation, window management, managing cookies, bookmarks, security, searching, rendering, and other features—is defined in XPCOM components and accessed by means of those component interfaces. Mozilla is *made* of XPCOM components.

This chapter demonstrates how Mozilla uses some of these XPCOM objects, such as the **CookieManager**, and shows how access to the **WebLock** component will be defined.

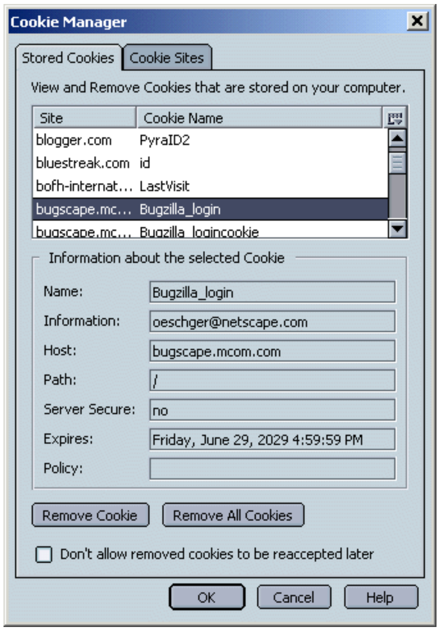
Component Examples

We’ll say more about how you can use the particular components described here in *Appendix B, The XPCOM API Reference*. For now, what’s important to see is how components like the ones in this section are obtained and used by the Mozilla browser.

Cookie Manager

Cookie management is one of the many sets of functionality that is made available to the browser in the form of an XPCOM component and that can be re-used by developers who want similar functionality in their applications. Whenever a user accesses the Cookie Manager dialog to view, organize, or remove cookies that have been stored on the system, they are using the **CookieManager** component behind the scenes. *Figure 1* shows user interface[1] that is presented to the user in Mozilla for working with the **CookieManager** component.

1. Note that the interface is not part of the component itself. XPCOM makes it easy to use components like CookieManager from Mozilla’s Cross Platform Front End (XPFE) and other user interfaces, but the component itself doesn’t provide it’s own UI.



View and Remove Cookies that are stored on your computer.

Site	Cookie Name
blogger.com	PyraID2
bluestreak.com	id
bofh-internat…	LastVisit
bugscape.mc…	Bugzilla_login
bugscape.mc…	Bugzilla_logincookie

Information about the selected Cookie

Name:	Bugzilla_login
Information:	oeschger@netscape.com
Host:	bugscape.mcom.com
Path:	/
Server Secure:	no
Expires:	Friday, June 29, 2029 4:59:59 PM
Policy:	

☐ Don't allow removed cookies to be reaccepted later

Figure 1. The Cookie Manager Dialog

This dialog is written in XUL and JavaScript, and uses a part of XPCOM called *XPConnect* to seemlessly connect to the **CookieManager** component (see XPConnect sidebar below). XUL is just one way to expose the functionality of the CookieManager component—but it's a particularly useful one in the Mozilla world.

The functionality of the **CookieManager** component is available through the `nsICookieManager` interface, which is comprised of the public methods in *Table 1*.

TABLE 1. **The nsICookieManager Interface**

`removeAll`	Remove all cookies from the cookie list.
`enumerator`	Enumerate through the cookie list.
`remove`	Remove a particular cookie from the list.

In XPCOM the interface is guaranteed to stay the same even if the underlying implementation changes. The interfaces are *public*, in other words, and the implementations are private[1]. When a user selects one of the cookies displayed in the list and then clicks the Remove buton, the `Remove` method in the `nsICookieManager` interface is called. The function is carried out by the **CookieManager** component, and the selected cookie is deleted from disk and removed from the displayed list.

Connecting to Components from the Interface

The Mozilla user interface uses JavaScript that has been given access to XPCOM components in the application core with a technology called *XPConnect*.

XPConnect allows interface methods defined via XPIDL to be called from JavaScript, as part of JavaScript objects that represent instances of components like the **CookieManager**.

XPConnect is what binds the application code to the user interface of the Mozilla browser, to other Gecko-based XUL, and to JavaScript environments like xpcshell, which is a command-line JavaScript interpreter and XPCOM tool is built with Mozilla.

See *http://www.mozilla.org/scriptable* for more information about XPConnect and JavaScript.

The snippet in *Figure 2* shows how the `Remove()` method from the XPCOM **CookieManager** component can be called from JavaScript:

1. There are exceptions to this. Some XPCOM interfaces are also private and not made for general use. Private interfaces do not have the same requirements as the ones that are made available publicly in IDL.

```
// xpconnect to cookiemanager
// get the cookie manager component in JavaScript
var cookiemanager = Components.classes["@mozilla.org/
     cookiemanager;1"].getService();
cookiemanager = cookiemanager.QueryInterface
     (Components.interfaces.nsICookieManager);

// called as part of a largerDeleteAllCookies() function
function FinalizeCookieDeletions() {
  for (var c=0; c<deletedCookies.length; c++) {
    cookiemanager.remove(deletedCookies[c].host,
                         deletedCookies[c].name,
                         deletedCookies[c].path);

  }
  deletedCookies.length = 0;
}
```

Figure 2. Getting the CookieManager Component in JavaScript

This isn't quite all there is to it, of course, but this shows an important aspect of XPCOM. The contractual arrangements that XPCOM enforces open up the way to *binary interoperability*—to being able to access, use, and reuse XPCOM components at run-time. And they make it possible to use components written in other languages—such as JavaScript, Python, and others—and to use C++-based XPCOM components *from* these other languages as well.

In the Mozilla browser, components are used as often from JavaScript in the interface as they are from C++ or any other language. In fact, a search of the Mozilla source code reveals that this **CookieManager** component is called *only* from JavaScript. We'll be using this component from JavaScript ourselves as part of this tutorial[1].

1. The CookieManager component is used to persist for the web locking functionality described in this tutorial.

JavaScript and Mozilla

JavaScript is the *lingua franca* of the Mozilla browser front-end, and the bindings between it and XPCOM are strong and well-defined. *Scriptability*, this ability to get and use XPCOM components from JavaScript and other languages for which XPConnect bindings have been created, is a core feature of XPCOM.

The WebBrowserFind Component

Components are used all over—in high-level browser-like functionality such as **nsWebBrowserFind**, which provides `find()` and `findNext()` methods for finding content in web pages, and in low-level tasks such as the manipulation of data. Though not every API in Mozilla is or should be "XPCOMified", much if not all of the typical functionality that browsers provide is available in components that can be reused via browser extensions and/or gecko embedders.

In addition to the **CookieManager** component, for example, the **WebBrowserFind** component is another part of a large set of web browsing interfaces you can use. Its `nsIWebBrowserFind` interface is shown in *Table 2*. To use this component, you access it through the `nsIWebBrowserFind` interface and call its methods.

TABLE 2. The nsIWebBrowserFind Interface

`findNext`	Find the next occurence of the search string
`findBackwards`	Boolean attribute that adjusts `findNext()` to search backwards up the document.
`searchFrames`	Boolean attribute that indicates whether to search sub-frames of current document.
`matchCase`	Boolean attribute that indicates whether to match case in the search.
`entireWord`	Boolean attribute that specifies whether the entire word should be matched or not.

Once you use the interface to get to the component, you can ask the component what other interfaces it supports. This service, which is defined in the basic `nsISupports` interface and implemented by all XPCOM components, allows you to query and switch interfaces on a component as part of the *run-time object typing* capabilities of XPCOM. It is handled by the `QueryInterface` method, which was introduced in the chapter "What Is XPCOM?" *Appendix B* in this book provides a full reference of the XPCOM components available in Mozilla.

The WebLock Component

Now it's time to look at the **WebLock** component as another example of XPCOM components (since you'll be creating it shortly). In object-oriented programming, it's typical to design the interface first—to define the functionality that's going to be provided in the abstract, without worrying about how this functionality will be achieved. So we'll put aside the details of the implementation until the next chapter and look at the component from the outside—at the interface to the **WebLock** component (see *Table 3*).

TABLE 3. **The IWebLock Interface**

`lock`	lock the browser to the current site (or to the whitelist of approved sites read from disk).
`unlock`	unlock the browser for unrestricted use.
`addSite`	add a new site to the whitelist.
`removeSite`	remove a given site from the whitelist.
`sites`	enumerator for the list of approved sites read in the from the whitelist.

The **WebLock** component is software that implements all of these methods in the way described by the interface definition. It registers itself for use when the browser starts up, and provides a factory that creates an instance of it for use when the user or administrator clicks the weblock icon in the browser's user interface.

Component Use in Mozilla

So how are components obtained and used in Mozilla? You've seen some enticing snippets of JavaScript in earlier sections of this chapter, but we haven't explained how XPCOM makes components available in general.

This section discusses practical component use in Mozilla. It’s divided into three subsections: one about actually finding all these binary components in Mozilla and two others corresponding to the two main ways that clients typically access XPCOM components:

- “Finding Mozilla Components”
- “Using XPCOM Components in Your C++”
- “XPConnect: Using XPCOM Components From Script”

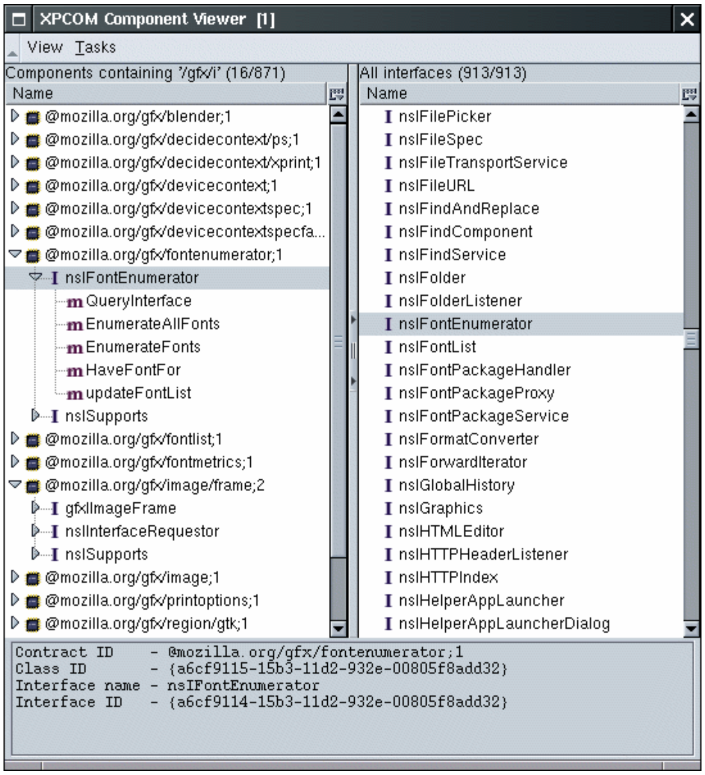
Finding Mozilla Components

This book attempts to provide reference information for XPCOM components and their interfaces that are frozen as of the time of this writing. The Mozilla embedding project (*www.mozilla.org/projects/embedding*) tracks the currently frozen interfaces.

Mozilla also has some tools that can find and display information about the interfaces available in Gecko such as the *XPCOM Component Viewer*, described below, and LXR, which is a web-based source code viewing tool you can access from *http://lxr.mozilla.org*.

The challenge to making good information about XPCOM components available to prospective clients, however, is that the process of freezing the interfaces that are implemented by these components is still ongoing. The Component Viewer does not distinguish between components that are frozen and those that are not. In the source code you view in LXR, interfaces that have been frozen are marked at the top with `@status frozen`.

The XPCOM Component Viewer. The Component Viewer is an add-on you can install in your browser from *www.hacksrus.com/~ginda/cview* (see *Figure 3*),



```
Contract ID    - @mozilla.org/gfx/fontenumerator;1
Class ID       - {a6cf9115-15b3-11d2-932e-00805f8add32}
Interface name - nsIFontEnumerator
Interface ID   - {a6cf9114-15b3-11d2-932e-00805f8add32}
```

Figure 3. XPCOM Component Viewer

The left column shows the components—in this case a subset returned from a search on ‘gfx’ as part of the contractc ID and the right column a list of the interfaces. When you open a component on the left, you can see the interfaces it implements along with a list of the methods provided by each interface.

The XPCOM Component Viewer can be extremely useful for this sort of gross interrogation, but again: it displays *all* of the components and interfaces in your build, many of which are not practical for actual reuse or stable enough to be used reliably in your own application development. Use comprehensive lists like this with caution.

Using XPCOM Components in Your C++

XPConnect makes it easy to acess XPCOM components as JavaScript objects, but using XPCOM components in C++ is not much more difficult.

Figure 4 duplicates code from Figure 5, but in C++ instead of JavaScript.

```
nsCOMPtr<nsIServiceManager> servMan;
nsresult rv = NS_GetServiceManager(getter_AddRefs(servMan));
if (NS_FAILED(rv))
  return -1;

nsCOMPtr<nsICookieManager> cookieManager;
rv = servMan->GetServiceByContractID("@mozilla.org/cookiemanager",
NS_GET_IID(nsICookieManager), getter_AddRefs(cookieManager));

if (NS_FAILED(rv))
  return -1;

PRUint32 len;
deletedCookies->GetLength(&len);

for (int c=0; c<len; c++)
    cookiemanager->Remove(deletedCookies[c].host,
                          deletedCookies[c].name,
                          deletedCookies[c].path);
```

Figure 4. Managing Cookies from C++

If your application written in C++, then *Figure 4* shows the steps you take to get an XPCOM component, specify the interface on that component you want to use, and call methods on that interface.

XPConnect: Using XPCOM Components From Script

The **CookieManager** component we discussed at the beginning of this chapter provides a good opportunity to talk further about using components from JavaScript. In the following code fragment from the Cookie Manager dialog in Mozilla, you can see a singleton of the the **CookieManager** component being created with the `getService()` method and used to provide the functionality that lets users load and remove cookies from the user interface.

```
var cookiemanager = Components.classes["@mozilla.org/
    cookiemanager;1"].getService();
cookiemanager = cookiemanager
     .QueryInterface(Components.interfaces.nsICookieManager);

function loadCookies() {
  // load cookies into a table
  var enumerator = cookiemanager.enumerator;
  var count = 0;
  var showPolicyField = false;
  while (enumerator.hasMoreElements()) {
    var nextCookie = enumerator.getNext();
    nextCookie = nextCookie.QueryInterface
      (Components.interfaces.nsICookie);
    ....
}
function FinalizeCookieDeletions() {
  for (var c=0; c<deletedCookies.length; c++) {
    cookiemanager.remove(deletedCookies[c].host,
                         deletedCookies[c].name,
                         deletedCookies[c].path,
  }
  deletedCookies.length = 0;
}
```

Figure 5. Managing Cookies from JavaScript

Beyond the methods that are being called on the **CookieManager** itself (e.g., `cookiemanager.remove`, which maps to the `remove()` function from the IDL in *Table 1* above), note the special XPConnect objects and methods that reflect the XPCOM component into JavaScript.

`Components` is the JavaScript object that controls the connection to components, and `classes` is an array of all of the classes you can ask for by contract ID. To instantiate an XPCOM component in JavaScript, you create a new `Component` object and pass in the contract ID for the component you want and ask for either a singleton or an instance of that component to be returned:

```
var cookiemanager = Components.classes
     ["@mozilla.org/cookiemanager;1"].getService();
```

The resulting `cookiemanager` object then provides access to all of the methods for that component that have been defined in IDL and compiled into the type library. Using the **CookieManager** component, you could write code like this to delete all cookies from the system:

```
cmgr = Components.classes
    ["@mozilla.org/cookiemanager;1"]
     .getService();
cookiemanager = cookiemanager.QueryInterface
  (Components.interfaces.nsICookieManager);

// delete all cookies
function trashEm() {
   cmgr.removeAll()
}
```

Another vital feature of the XPConnect glue this example shows is the availability of the `QueryInterface` method on all objects that are reflected into JavaScript from XPCOM. As in C++, you can use this method to ask for other interfaces that are available on the given object.

Services Versus Regular Instances

Whether to have clients use your component as an instance or a service is a design question, really, and something you should be clear about in the documentation for your component. Actually, the `getService()` method in the example here calls through to the `createInstance()` method that is also available from the Component object and caches the result, making it a singleton rather than a normal instance.

The singleton design pattern that is used to create services is described in “XPCOM Services” on page 23.

Remember, `QueryInterface` allows you to query an object for the interfaces it supports. In the case of the snippet in *Figure 5*, the `QueryInterface` method is being used to get the `nsICookie` interface from the enumerator so that, for instance, the JavaScript code can access the `value` and `name` attributes for each cookie.

CHAPTER 3

Component Internals

Where the previous chapter described components from the perspective of a client of XPCOM components, this chapter discusses components from the perspective of the software developer. Read on to see how components are generally implemented in XPCOM, or you can skip to the next chapter, where the **WebLock** component tutorial takes you step by step through the component creation process.

Creating Components in C++

Let’s start by examining how XPCOM components are written in C++. The most common type of component is one that is written in C++ and compiled into a shared library (a DLL on a Windows system or a DSO on Unix).

The illustration below shows the basic relationship between the shared library containing the component implementation code you write and the XPCOM framework itself. In this diagram, the outer boundary is that of the module, the shared library in which a component is defined.

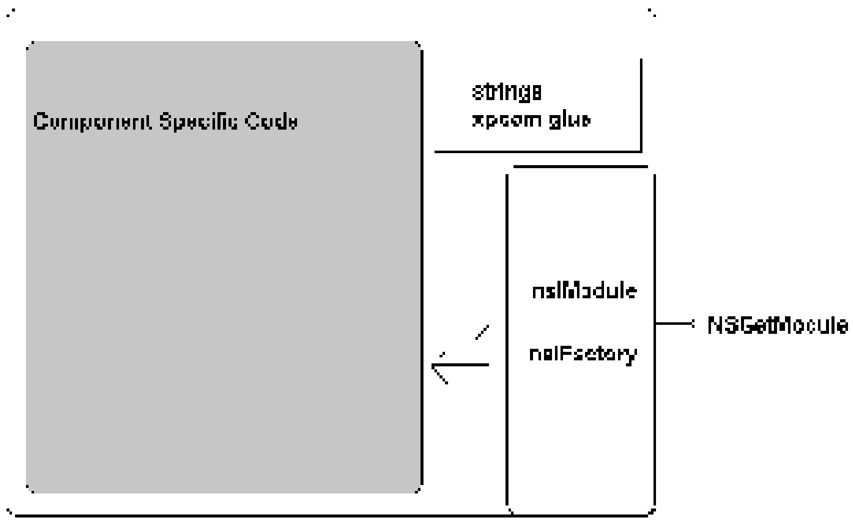


Figure 1. A Component in the XPCOM Framework

When you build a component or module and compile it into a library, it must export a single method named `NSGetModule`. This `NSGetModule` function is the entry point for accessing the library. It gets called during registration and unregistration of the component, and when XPCOM wants to discover what interfaces or classes the module/library implements. In this chapter we will outline this entire process.

As *Figure 1* illustrates, in addition to the `NSGetModule` entry point, there are `nsIModule` and `nsIFactory` interfaces that control the actual creation of the component, and also the string and XPCOM glue parts, which we'll discuss in some detail in the next section (see "XPCOM Glue" on page 48"). These latter supply ease-of-development utilites like smart pointers, generic modules support, and simple string implementations. The largest and possibly most complex part of a component is the code specific to the component itself.

But *Where* Are the Components?

Components reside in modules, and those modules are defined in shared library files that typically sit in the *components* directory of an XPCOM application.

A set of default libraries stored in this components directory makes up a typical Gecko installation, providing functionality that consists of networking, layout, composition, a cross-platform user interface, and others.

Another, even more basic view of this relationship of components to the files and interfaces that define them is shown in Figure 2 on page 57. The component is an abstraction sitting between the actual module in which it is implemented and the objects that its factory code creates for use by clients.

XPCOM Initialization

To understand why and when your component library gets called, it is important to understand the XPCOM initalization process. When an application starts up, that application may *initialize* XPCOM. The sequence of events that kicks off this XPCOM initialization may be triggered by user action or by the application startup itself. A web browser that embeds Gecko, for example, may initialize XPCOM at startup through the embedding APIs. Another application may delay this startup until XPCOM is needed for the first time. In either case, the initialization sequence within XPCOM is the same.

XPCOM starts when the application makes a call to initialize it. Parameters passed to this startup call allow you to configure some aspects of XPCOM, including the customization of location of specific directories. The main purpose of the API at this point is to change which *components* directory XPCOM searches when it looks for XPCOM components. This is how the API is used, for example, in the *Gecko Run-time Environment* (GRE).

XPCOM Startup

The six basic steps to XPCOM startup are as follows:

1. Application starts XPCOM.
2. XPCOM sends a notification that it's beginning startup.
3. XPCOM finds and processes the *component manifest* (see *"Component Manifests"* below).
4. XPCOM finds and processes the *type library manifest* (see *"Type Library Manifests"* below).
5. If there are new components, XPCOM registers them:
   a. XPCOM calls autoregistration start.
   b. XPCOM registers new components.
   c. XPCOM calls autoregistration end.
6. Complete XPCOM startup: XPCOM notifies that it's begun.

Component manifests and type library manifests are described in the following section, " XPCOM Registry Manifests".

XPCOM Registry Manifests

XPCOM uses special files called *manifests* to track and persist information about the components to the local system. There are two types of manifests that XPCOM uses to track components:

- " Component Manifests"
- " Type Library Manifests"

Component Manifests. When XPCOM first starts up, it looks for the *component manifest*, which is a file that lists all registered components, and stores details on exactly what each component can do. XPCOM uses the component manifest to determine which components have been overridden. Starting with Mozilla 1.2, this file is named *compreg.dat* and exists in the *components* directory, but there are efforts to move it out of this location to a less application-centric (and thus more user-centric) location. Any Gecko-based application may choose to locate it elsewhere.

XPCOM reads this file into an in-memory database.

Component Manifests

The component manifest is a mapping of files to components and components to classes. It specifies the following information:

- Location on disk of registered components with file size.
- Class ID to Location Mapping
- Contract ID to Class ID Mapping

The component manifest maps component files to unique identifiers for the specific implementations (class IDs), which in turn are mapped to more general component identifiers (contract IDs).

Type Library Manifests. Another important file that XPCOM reads in is the *type library manifest* file. This file is also located in the *components* directory and is named *xpti.dat*. It includes the location and search paths of all type library files on the system. This file also lists all known interfaces and links to the type library files that define these interface structures. These type library files are at the core of XPCOM scriptablity and the binary component architecture of XPCOM.

Type Library Manifests

Type library manifests contain the following information:

- location of all type library files
- mapping of all known interfaces to type libraries where these structures are defined

Using the data in these two manifests, XPCOM knows exactly which component libraries have been installed and what implementations go with which interfaces. Additionally, it relates the components to the type libraries in which the binary representations of the interfaces they support are defined.

The next section describes how to hook into the XPCOM startup and registration process and make the data about your component available in these manifests, so that your component will be found and registered at startup.

Registration Methods in XPCOM

What Is XPCOM Registration?

In a nutshell, registration is the process that makes XPCOM aware of your component(s). As this section and the next describe, you can register your component explicitly during installation, or with the `regxpcom` program, or you can use the autoregistration methods in the Service Manager to find and register components in a specified components directory:

- XPInstall APIs
- `regxpcom` command-line tool
- `nsIComponentRegistrar` APIs from Service Manager

The registration process is fairly involved. This section introduces it in terms of XPCOM initialization, and the next chapter describes what you have to do in your component code to register your component with XPCOM.

Once the manifest files are read in, XPCOM checks to see if there are any components that need to be registered. There are two supported ways to go about registering your XPCOM component. The first is to use *XPInstall*, which is an installation technology that may or may not come with a Gecko application and provides interfaces for registering your component during installation. Another, more explicit way to register your component is to run the application *regxpcom*, which is built as part of Mozilla and also available in the Gecko SDK. *regxpcom* registers your component in the default component registry.

A Gecko embedding application may also provide its own way of registering XPCOM components using the interface that is in fact used by both XPInstall and regxpcom, `nsIComponentRegistrar`. An application, for example, could provide a "registration-less" component directory whose components are automatically registered at startup and unregistered at shutdown. Component discovery does not currently happen automatically in non-debug builds of Gecko, however.

When the registration process begins, XPCOM broadcasts to all registered observers a notification that says XPCOM has begun the registration of new components. After all components are registered, another notification is fired saying that XPCOM is done with the registration step. The `nsIObserver` interface that handles this notification is discussed in the chapter “Tutorial: Starting WebLock”.

Once registration is complete and the notifications have fired, XPCOM is ready to be used by the application. If XPCOM registered your component, then it will be available to other parts of the XPCOM system. The “XPCOM Initialization” section in this chapter describes registration in more detail.

Autoregistration

The term *autoregistration* is sometimes used synonymously with registration in XPCOM. In the *What is Registration?* box on the previous page, we describe the three ways you can register components with XPCOM. Sometimes, applications use the `nsIComponentRegistrar` interface and create their own code for watching a particular directory and registering new components that are added there, which is what’s often referred to as *autoregistration*. You should always know what the installation and registration requirements are for the applications that will be using your component.

The Shutdown Process

When the application is ready to shutdown XPCOM, it calls `NS_ShutdownXPCOM`. When that method is called, the following sequence of events occurs:

1. XPCOM fires a shutdown notification to all registered observers.
2. XPCOM closes down the Component Manager, the Service Manager and associated services.
3. XPCOM frees all global services.
4. `NS_ShutdownXPCOM` returns and the application may exit normally.

The Unstoppable Shutdown

Note that shutdown observation is unstoppable. In other words, the event you observe cannot be used to implement something like a "Are you sure you want to Quit?" dialog. Rather, the shutdown event gives the component or embedding application a last chance to clean up any leftovers before they are released. In order to support something like an "Are you sure you want to quit" dialog, the application needs to provide a higher-level event (e.g., `startShutdown()`) which allows for cancellation.

Note also that XPCOM services may deny you access once you have received the shutdown notification. It is possible that XPCOM will return an error if you access the `nsIServiceManager` at that point, for example, so you may have to keep a reference-counted pointer to the service you are interested in using during this notification.

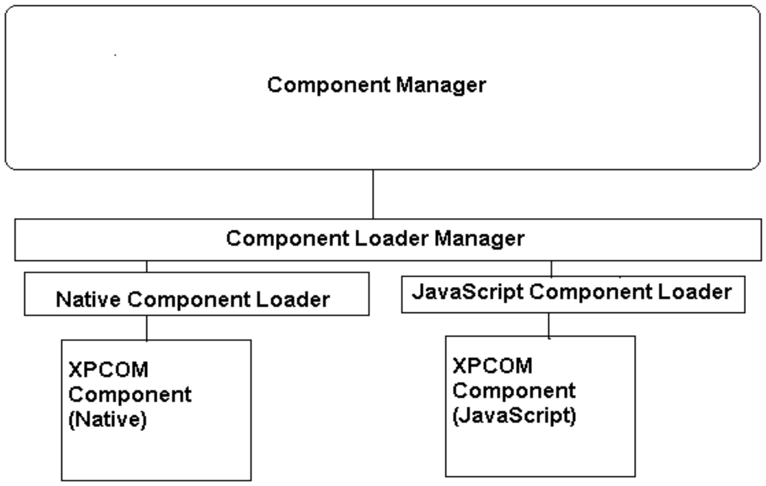
Component Loaders

Components can be written in many languages. So far this book has been focusing on “native components,” shared libraries exporting a `NSGetModule` symbol. But if there is a *component loader* for Javascript installed, then you can also write a JavaScript component.

To register, unregister, load and manage various component types, XPCOM abstracts the interface between the XPCOM component and XPCOM with the Component Loader. This loader is responsible for initialization, loading, unloading, and supporting the `nsIModule` interface on behalf of each component. It is the Component Loader’s responsibility to provide scriptable component support.

When building a “native” component, the component loader looks for an exported symbol from the components shared library. “Native” here includes any language that can generate a platform native dynamically loaded library. Scripting languages and other “non-native” languages usually have no way to build native libraries. In order to have “non native” XPCOM components work, XPCOM must have a special component loader which knows how to deal with these type of components.

XPConnect, for example, provides a component loader that makes the various types, including the interfaces and their parameters, available to JavaScript. Each language supported by XPCOM must have a component loader.



Three parts of a XPCOM Component Library

XPCOM is like an onion. XPCOM components have at least three layers. From the innermost and moving outward these layers include:

- The core XPCOM object
- The factory code
- The module code

The core XPCOM object is the object that will implement the functionality you need. For example, this is the object that may start a network download and implement interfaces that will listen to the progress. Or the object may provide a new content type handler. Whatever it does, this object is at the core of the XPCOM component, and the other layers are supporting it, plugging it into the XPCOM system. A single library may have many of these core objects.

One layer above the core object is the factory code. The factory object provides a basic abstraction of the core XPCOM object. Chapter 1 discussed the factory design pattern that’s used in a factory object. At this layer of the XPCOM Component Library, the factory objects are factories for the core XPCOM objects of the layer below.

One more layer outward is the module code. The module interface provides yet another abstraction—this time of the factories—and allows for multiple factory objects. From the outside of the component library, there is only the single entry point, `NSGetModule()`. This point of entry may fan out to any number of factories, and from there, any number of XPCOM objects.

The following chapter details these layers in terms of the XPCOM interfaces that represent them. Here we will just introduce them. The factory design pattern in XPCOM is represented by the `nsIFactory` interface. The module layer is represented by the `nsIModule` interface. Most component libraries only need these two interfaces, along with the `nsISupport` interface, to have XPCOM load, recognize, and use their core object code.

In the next section, we'll be writing the code that actually compiles into a component library, and you will see how each layer is implemented and how each interface is used. Following this initial, verbose demonstration of the API's, we will introduce a faster more generic way of implementing the module and factory code using macros, which can make components much easier to create.

XPCOM Glue

XPCOM contains a lot of stuff. Most of the XPCOM interfaces are not frozen and are meant to be used only by the Gecko internals, not by clients. XPCOM provides many data structures from linked lists to AVL trees. Instead of writing your own linked list, it's tempting to reuse `nsVoidArray` or another publicly available class, but this might be a fatal mistake. At any time the class can change and give you unexpected behavior.

XPCOM makes for a very open environment. At runtime you can acquire any service or component merely by knowing a CID, or Contract ID, and an IID. At last count there were over 1300 interfaces defined in XPIDL. Of those 1300 interfaces, less than 100 were frozen, which means that a developer has a good chance of stumbling upon useful interfaces that aren't frozen. If an interface isn't explicitly marked "FROZEN" in the IDL comments, however—and most of them aren't—it will cause your component to possibly break or crash when the version changes.

The Glue Library

In general you should avoid any unfrozen interfaces, any symbols in XPCOM, or any other part of Gecko libraries that aren't frozen. However, there are some unfrozen tools in XPCOM that are used so often they are practically required parts of component programming.

The smart pointer class, `nsCOMPtr`, for example, which can make reference counting a lot less tedious and error-prone, is not actually frozen, and neither are `nsDebug`, a class for aiding in tracking down bugs, or `nsMemory`, a class to ensure that everyone uses the same heap, generic factory, and module. Instead of asking every developer to find and copy these various files into their own application, XPCOM provides a single library of "not-ready-to-freeze-but-really-helpful" classes that you can link into your application, as the following figure demonstrates.

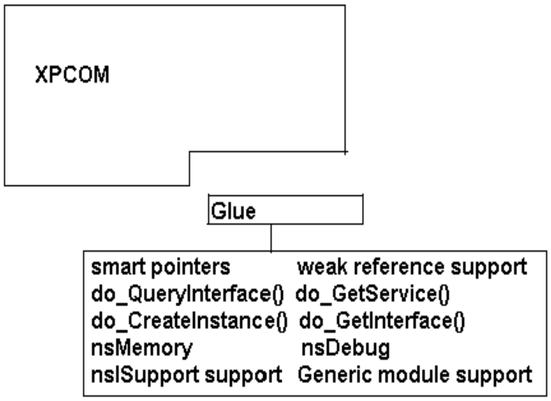


Figure 5. XPCOM Glue and Tools

This is the glue library. It provides a bridge, or "glue" layer, between your component and XPCOM.

A version of the glue library is built into XPCOM, and when your component uses it, it links a snapshot of this library: it includes a copy of these unfrozen classes directly, which allows the XPCOM library version to change without affecting the software. There is a slight footprint penalty to linking directly, but this gives your component freedom to work in any post Mozilla 1.2 environment. If footprint is a big issue in your component or application, you can trim out the pieces you don’t need.

XPCOM String Classes

The base string types that XPCOM uses are `nsAString` and `nsACString`. These classes are described in the Mozilla String guide (see the “Resources” section in Appendix C).

The string classes that implement these abstract classes are another set of helpful, unfrozen classes in XPCOM. Most components and embedding applications need to link to some kind of string classes in order to utilize certain Gecko APIs, but the string code that Mozilla uses is highly complex and even more expensive than the glue code in terms of footprint (~100k). `nsEmbedString` and `nsEmbedCString` are available as very light string class implementations for component development, especially in small embedded applications. This string implementation does the bare minimum to support `nsAString/nsACString` string classes

In your own component, you can go “slim” and restrict yourself to the `nsEmbedString` or go “hog wild” and use all of the functionality of the other strings. **WebLock** restricts itself to using the simple `nsEmbedString` family of classes.

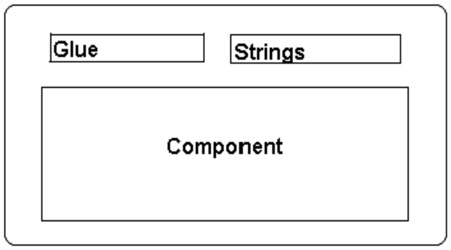


Figure 6. String Classes and XPCOM

The glue library provides stub functions for the public functions that XPCOM provides (see *xpcom/build/nsXPCOM.h*). When the glue library is initialized, it dynamically loads these symbols from the XPCOM library, which allows the component to avoid linking directly with the XPCOM library. You shouldn’t have to link to the XPCOM library to create a XPCOM component—in fact, if your component has to, then something is wrong. .

CHAPTER 4

Tutorial : Creating the Component Code

Topics covered in this chapter:

- “What We’ll Be Working On”
- “Overview of the WebLock Module Source”
- “Digging In: Required Includes and Constants”
- “webLock1.cpp”

This chapter goes over the basic code required to handle the relationship between your component and XPCOM. Having the component found and registered properly is the goal of this first chapter of the tutorial. In the subsequent chapters, we can begin to work on the example **WebLock** component functionality itself.

‘

Use the Calculator (After Learning Long Division)

You have to write a fair amount of code to create a component library that gets loaded into XPCOM. An XPCOM component needs to implement at least three interfaces that XPCOM requires, and often others as well. There is more code in this chapter than you’ll eventually need, however. Chapter 5 shows some simpler and more elegant ways to create an XPCOM component using generic macros, and this chapter is more about learning the basics. As in grade school when you learned long division, better tools like calculators come *after* you figure out what’s actually happening. In this case, the long-hand implementation gives us an opportunity to talk about various features of XPCOM.

What We’ll Be Working On

The component we’ll be working on in this book controls a special mode in your browser that prevents users from leaving the current domain or a set of safe domains. Once enabled, this weblock mode is password protected and persists until it is turned off by the password holder. It can be used to make the browser into a safe viewer for children, or for targeted “kiosk browsing,” where the content is restricted to a particular server. *Figure 1* shows the icon that is used to activate the web lock mode (leftmost in the status bar) once you have installed the **WebLock** component and the extra user interface.

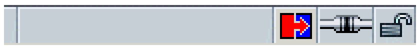


Figure 1. Web Lock User Interface

Most of the actual work in the **WebLock** component is preparing the component itself, finding the XPCOM interfaces the component needs to use, and hooking into existing functionality within the Gecko Browser.

Component Registration

All XPCOM components—whether they're stored in shared libraries (DLLs or DSOs), JavaScript files, or some other file—need to be *registered* before they can be used. Registration is a process that happens in all XPCOM applications, whether they're embedded Gecko clients, Mozilla, Netscape 7, Compuserve, or any other software that uses XPCOM. Registration provides the information that applications need in order to use components properly.

The **WebLock** component must do a number of things to register itself. Specifically, the component library has to contain implementations for the component-related interfaces described in this chapter: `nsIModule` and `nsIFactory`, which are entry points for your implementation code.

Once your component implements these interfaces, the rest of the registration process itself is simple. Applications typically use *`regxpcom`*, described in the next section.

The regxpcom Program

An explicit way to register a component is to run the application *regxpcom*. Without any arguments passed to *regxpcom*, the program registers the component in the default component registry. We suggest that when you are testing your component in the Mozilla or Netscape client, you copy your component into the "components" directory in the client's installation folder. When it is copied there, run *regxpcom* from the command line to register that component and all the others in that directory.

Note: Several new options have been added to *regxpcom* in versions 1.4 and later. You can run *regxpcom* with the *-h* option to see full usage options.

Registration Alternatives

A Gecko embedding application may provide other ways to register XPCOM components. XPInstall, the cross-platform installation technology that Mozilla uses to install the browser and other components, is one such alternative. It is described further in the chapter "Tutorial: Packaging WebLock" on page 157. You should consult with the authors of the application you wish to extend to see if there are other supported ways.

Overview of the WebLock Module Source

As we mentioned in the previous section, components have layers. There are three main parts to every XPCOM Component. From the innermost and moving outward, the first object is the XPCOM object. This is the object that contains the business logic, that implements functionality such as starting a network download, implementing interfaces that listen to the download progress, or providing a new content type handler. In **Weblock**, this is the part that brings together various Gecko services and prevents users from leaving the list of acceptable domains. In a way, he factory and module layers are glue to plug the XPCOM object into the larger XPCOM system.

One layer above the object itself is the `nsIFactory` object. This object provides basic abstraction of the XPCOM object itself. As you can see in the diagram in *Figure 2,* the main accessor for the XPCOM object is `CreateInstance`, which is expected to return the object that matches a given CID and IID pair.

Moving another layer outward is the `nsIModule`. This interface provides yet another abstraction of the `nsIFactory` object, and may allow for multiple `nsIFactory` objects. The key to this interface is that the return type of `getClassObject` does not have to be an `nsIFactory`. Instead, the `nsIModule` can ask for implementation details about the XPCOM object. This is very useful if the caller is required to know information about the component like its threading module, whether it’s singleton or not, its implementation language, and so forth. The interface used in this case is `nsIClassInfo`. Starting from the outside in, *Figure 2* represents the sequence for constructing an XPCOM object.

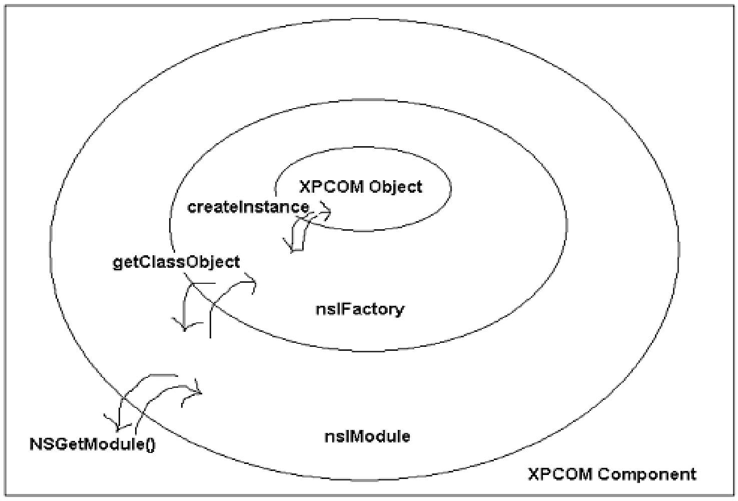


Figure 2. Onion Peel View of XPCOM Component Creation

Before we begin looking at the various parts of the component and how they'll be implemented in the source, let's look at the module in *weblock.cpp* as a whole to see where we're going. The source we're referring to is listed in its entirety at the end of this chapter (see "webLock1.cpp" on page 68).

The source file for the **WebLock** component contains three classes. In order to make the **WebLock** component work in Mozilla, you have to implement a new interface to the **WebLock** component, `iWebLock`, where the actual work specific to the the web locking features happens. You also have to create `WebLockModule` to implement the necessary `nsIModule` interface, and you have to create `WebLockFactory` to implement `nsIFactory` and create a factory that hands instances of your component to clients. These three interface implementations—of the component functionality, of the `nsIModule` interface, and of the `nsIFactory` interface that creates and manages instances for clients—are the basic sets of code you need to write to create an XPCOM component.

Basic Structure of the WebLock Component Source

The *weblock1.cpp* source file that defines these classes and the code you need to create a basic component has the following structure:

- required includes and constants
- **WebLock**: public `iWebLock`
- **WebLockFactory**: public `nsIFactory`
- **WebLockModule**: public `nsIModule`

In XPCOM, all of these classes also derive from the `nsISupports` base interface.

Digging In: Required Includes and Constants

Let's take a look at the first several lines of code in the component and discuss what they mean in XPCOM. The includes and definitions at the top of an XPCOM source file can give you an idea about some of the data types and techniques we'll be discussing more in the upcoming chapters.

For example, `MOZILLA_STRICT_API` is a variable that shields you from certain private, non-XPCOM headers. For example, including *nsIComponentManager.idl* without `MOZILLA_STRICT_API` defined will include the following headers, which are not supported across versions (unfrozen):

- *nsComponentManagerUtils.h*
- *nsComponentManagerObsolete.h*

These variables are picked up by files that do not specify themselves as `MOZILLA_STRICT_API`.

```
#include <stdio.h>

// may be defined at the project level
// in the makefile
#define MOZILLA_STRICT_API

#include "nsIModule.h"
#include "nsIFactory.h"

#include "nsIComponentManager.h"
#include "nsIComponentRegistrar.h"

// use classes to handle IIDs
// classes provide methods for comparison: Equals, etc.
static const nsIID kIModuleIID   = NS_IMODULE_IID;
static const nsIID kIFactoryIID   = NS_IFACTORY_IID;
static const nsIID kISupportsIID = NS_ISUPPORTS_IID;
static const nsIID
    kIComponentRegistrarIID = NS_ICOMPONENTREGISTRAR_IID;


// generate unique ID here with uuidgen
#define SAMPLE_CID \
{ 0x777f7150, 0x4a2b, 0x4301, \
{ 0xad, 0x10, 0x5e, 0xab, 0x25, 0xb3, 0x22, 0xaa}}

static const nsCID kSampleCID = SAMPLE_CID;
```

Figure 3. Listing 1: Includes and Constants in weblock1.cpp

nsIModule.h and *nsIFactory.h* are required to build your module successfully. They define the module and factory interfaces, and they contain a couple of important macros as well (see the following chapter for information about using these macros). The two other includes, *nsIComponentManager.h* and *nsIComponentRegistrar.h*, provide functions such as `RegisterFactoryLocation` that are required to implement the module and factory classes in your code.

Identifiers in XPCOM

The series of `nsIID` variables initialized here are actually classes created for handing the 128 bit identifiers that XPCOM uses to support contractual relationships between the client and component interfaces. The variable `kIFactoryIID`, for example, provides methods like `Equals()` that can be used to facilitate comparisons in the code, as in the following example from the Mozilla source in *Figure 4*.

```
if (aIID.Equals(NS_GET_IID(nsISupports))) {
  *aInstancePtr = (void*)(nsISupports*)this;
  NS_ADDREF_THIS();
  return NS_OK;
}
```

Figure 4. Listing 2: Using Class Methods to Handle Identifiers

Finally, `SAMPLE_CID` is an example of the CID that uniquely identifies each component. All of the 128 bit numbers used in XPCOM—the class and the interface IDs—are examples of UUIDs, or *universal unique identifiers*, which were discussed in the “Object Interface Discovery” section of the chapter “What Is XPCOM?”

Generating CIDs

To generate a CID for your component, you can use the uuidgen tool that comes with most Unix distributions and with Microsoft Visual C++. uuidgen is a command-line tool that returns a unique 128 bit number when you call it with no arguments:

```
$ uuidgen
ce32e3ff-36f8-425f-94be-d85b26e634ee
```

On Windows, a program called guidgen.exe does the same thing and also provides a graphical user interface if you'd rather point and click. Or you can use one of the special "bots" on IRC at the irc.mozilla.org server (irc://irc.mozilla.org/#mozilla), where you can also get help from human beings.

```
irc irc.mozilla.org
/join #mozilla
/msg mozbot uuid
```

This command makes the bot generate and return a UUID, which you can then copy into your component source code.

Now that we've looked at the preliminaries, it's time to discuss the classes that this module provides and the way that they define the relationships of the component in XPCOM.

Coding for the Registration Process

When XPCOM discovers your component for the first time (via XPInstall or `regxpcom`, both of which are discussed in "Component Installation Overview"), the first thing it tries to do is load your library and find the symbol `NSGetModule`. When this special entry point is called, it is passed XPCOM's Component Manager and the location of the shared library where the component lives.

The Component Manager is an interface implemented by XPCOM that encapsulates the creation of objects and provides summary information about all registered components. The location on disk is passed via another interface named `nsIFile`. This interface is XPCOM's abstraction of files and directories. An `nsIFile` object is usually a file or directory on a local volume, but it may represent something on a network volume as well.

```
nsresult NSGetModule(nsIComponentManager *servMgr,
                     nsIFile* location,
                     nsIModule** result);
```

XPCOM expects a successful call to NSGetModule to return an implementation of the interface nsIModule. When you write a XPCOM component, you implement nsIModule to do all of the necessary registration, unregistration, and object creation. nsIModule has four methods that must be implemented.

The Registration Methods

Two closely related registration methods are declared below.

```
NS_IMETHOD RegisterSelf(nsIComponentManager *aCompMgr,
                         nsIFile *aLocation,
                         const char *aLoaderStr,
                         const char *aType) = 0;

NS_IMETHOD UnregisterSelf(nsIComponentManager *aCompMgr,
                          nsIFile *aLocation,
                          const char *aLoaderStr) = 0;
```

RegisterSelf is called when a component is first registered with XPCOM. It is only called once, which gives you a chance to add any one time setup functionality. The RegisterSelf call allows your component to tell XPCOM exactly what it supports. Note that whatever you do in RegisterSelf should be undone in UnregisterSelf.

First, the NSGetModule entry point is called in your component library, and it returns an interface pointer to a nsIModule implementation. Then XPCOM calls RegisterSelf, passing parameters that we'll examine here.

The RegisterSelf Method. The first parameter is the nsIComponentManager, which provides a kind of entry point into managing the registration process. You can QueryInterface it to access to the other component management interfaces described below.

The Many Faces of the XPCOM Component Manager

The three core component management interfaces, `nsIComponentManager`, `nsIServiceManager`, and `nsIComponentRegistrar`, are described below.

- `nsIComponentManager` - Creates objects and gets implementation details about objects
- `nsIServiceManager` - Provides access to singleton objects and discovers singleton state
- `nsIComponentRegistrar` - Registers and unregisters factories and components; handles autoregistration and the discovery and enumeration of registered components.

Your `RegisterSelf` method may call `QueryInterface` on the `nsIComponentManager` interface parameter to obtain the `nsIComponentRegistrar` or `nsIServiceManager`. `nsIServiceManager` can be used to obtain a singleton service, which can be useful if you have to register with a service other than the `nsIComponentRegistrar` if necessary. For example, you may want to get the service that is responsible for an event you want to be notified about. See the section “Getting Called at Startup” on page 91 for an example of this.

The second parameter in `RegisterSelf` is the location of the component being registered. This parameter is useful when the component needs to know where it has been installed or registered—as, for example, when other files must be stored or accessed relative to the component. This method is only called once, so you have to persist the location if you are going to use it later.

The next two parameters are usually passed into the `nsIComponentRegistrar` methods and used by XPCOM to determine how to handle the component's registration. The `aLoaderStr` parameter, which is opaque and should not be modified, distinguishes components that are loaded from the same location specified by the `nsIFile` parameter. A single ZIP archive may store several XPCOM components, where every component in the archive has the same `nsIFile` parameter but the `aLoaderStr` parameter can be used to refer to the location within the ZIP archive.

The last parameter specifies what kind of loader to use on the component. This is reserved as an optimization, for the most part, but it can be a useful way to extend XPCOM. Since XPCOM already knows internally what kind of file it has just loaded and called `RegisterSelf` on, passing this value to the registration methods is a shortcut for determining what kind of component is being registered.

nsIComponentRegistrar Methods. To tell XPCOM what implementation is in the component library, call this method:

```
NS_IMETHOD RegisterFactoryLocation(const nsCID & aClass,
                const char *aClassName,
                const char *aContractID,
                nsIFile *aFile,
                const char *aLoaderStr,
                const char *aType) = 0;
```

The last three parameters are the same as the three passed into the `RegisterSelf` method of `nsIModule` objects. All you have to do is forward these parameters from your `RegisterSelf` call into this method, leaving just the first three parameters.

For any class that implements an XPCOM interface, the implementation must have a class identifier if it is to be shared with other parts of code via XPCOM. This identifier, called a CID, uniquely specifies the implementation. This CID can be created via the tool uuidgen on most operating systems, as in the sidebar above (“The Many Faces of the XPCOM Component Manager”). Given a CID and an IID, you can refer to any class in XPCOM. Consider the following:

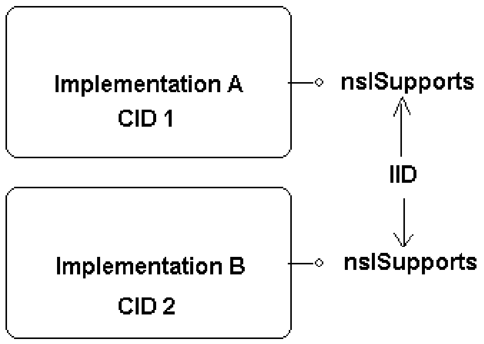


Figure 5. Figure X: Referencing Objects by ID

In this case, you have two implementations of the `nsISupports` interface. Each implementation has a separate CID. The interface also as an IID which is the same for both implementations. When specifying implementation A, the two required pieces of information are the CID of A and the IID of the interface that A supports. The code to register such an object is simple:

```
NS_IMETHODIMP
SampleModule::RegisterSelf(nsIComponentManager *aCompMgr,
  nsIFile* aPath,
  const char* registryLocation,
  const char* componentType)
{
    printf("Hello Mozilla Registration!\n\n");
    nsIComponentRegistrar* compReg = nsnull;
    nsresult rv = aCompMgr->
    QueryInterface(kIComponentRegistrarIID,(void**)& comp
    if (NS_FAILED(rv))
        return rv;
    rv = compReg->RegisterFactoryLocation(kSampleCID,
                                          "Sample Class",
                                          nsnull,
                                          aPath,
                                          registryLocation,
                                          componentType);
    compReg->Release();
    return rv;
}
```

Unregistration follows the same logic. To unregister, all you have to do is pass the CID and the file which is passed into `UnregisterSelf`.

Creating an Instance of Your Component

The example above uses a CID, but after the component is registered, anyone that uses XPCOM can access your classes if they know either the contract ID or CID. (Note that `RegisterSelf` method above does not register a contract ID—it simply passes null. This prevents clients from ever accessing the component with a contract ID.)

To be accessible to others, you need to publish the CID and/or contract ID of the component along with the interfaces it supports. Given the example above, someone could create the **Sample** object via the component manager as follows:

```
nsIComponentManager* compManger;  // assume initialized

nsISupports* sample;
compManager->CreateInstance(kSampleCID,
        nsnull,
        kISupportsIID,
        (void**)&sample);
```

In the above snippet, we assume that the component manager has been initialized. In many cases this value is passed in or easily accessible. If not, it can always be obtained by a call to `NS_GetComponentManager()`. A listing of this and other global XPCOM functions is in *Appendix B: The XPCOM API Reference*.

The first parameter of the call to `CreateInstance` specifies the component the client code is looking for, which is the same value passed to `RegisterFactoryLocation`. The next parameter is for aggregation, which the **WebLock** component does not support. The third parameter is the interface used to talk to the component. The last parameter is the out variable which will contain a valid object if and only if the method succeeds[1]. The implementation of CreateInstance will ensure that the result will support the passed IID, `kISupportsIID`. The type of the variable `sample` should match the IID passed in as `kISupportsIID`.

When `CreateInstance` is called, XPCOM looks through all registered components to find a match for the given CID. XPCOM then loads the component library associated with the CID if it isn't loaded already. XPCOM then calls the function `NSGetModule` on the library. Finally, it calls the method `GetClassObject` on the module. This method, which you must implement in your component code, is expected to return an `nsIFactory` object for a give CID/IID pair. To prepare your component code, you need to create a factory object for each object that you have registered with XPCOM.

1. Note: the `CreateInstance` method guarantees that if the out variable is non-null, it is valid.

The main function that must be implemented in the `nsIFactory` interface is `CreateInstance`. The implementation follows a simple algorithm:

1. Create the raw object.
2. If that fails, return an out of memory error code.
3. Call `QueryInterface` on the new object.
4. If that fails, null the out param and free the new object.
5. Return the `nsresult` value from `QueryInterface`.

Often, you don't have to create the object first because the factory implicitly knows what IIDs are supported. When this is not the case, however, doing it this way further abstracts the factories from their concrete classes. If you have a factory that knows every IID supported by the concrete base class, for example, then when you go to add a new supported interface you add this IID comparison in both the factory and the `QueryInterface` implementation in the concrete class.

```
NS_IMETHODIMP
SampleFactory::CreateInstance(
       nsISupports *aOuter,
    const nsIID & iid,
    void * *result)
{
    if (!result)
        return NS_ERROR_INVALID_ARG;

    Sample* sample = new Sample();
    if (!sample)
        return NS_ERROR_OUT_OF_MEMORY;

    nsresult rv = sample->QueryInterface(iid, result);

    if (NS_FAILED(rv)) {
        *result = nsnull;
        delete sample;
    }

    return rv;
}
```

webLock1.cpp

Before any of the improvements and XPCOM tools we describe in the following chapter are brought in, the source code for the **WebLock** component that implements all the necessary interfaces looks like this.

```
#include <stdio.h>

#define MOZILLA_STRICT_API

#include "nsIModule.h"
#include "nsIFactory.h"

#include "nsIComponentManager.h"
#include "nsIComponentRegistrar.h"

static const nsIID kIModuleIID   = NS_IMODULE_IID;
static const nsIID kIFactoryIID   = NS_IFACTORY_IID;
static const nsIID kISupportsIID = NS_ISUPPORTS_IID;
static const nsIID kIComponentRegistrarIID = NS_ICOMPONENTREGISTRAR_IID;


#define SAMPLE_CID \
{ 0x777f7150, 0x4a2b, 0x4301, \
{ 0xad, 0x10, 0x5e, 0xab, 0x25, 0xb3, 0x22, 0xaa}}

static const nsCID kSampleCID = SAMPLE_CID;

class Sample: public nsISupports {
private:
  nsrefcnt mRefCnt;
public:
  Sample();
  virtual ~Sample();

  NS_IMETHOD QueryInterface(const nsIID &aIID, void **aResult);
  NS_IMETHOD_(nsrefcnt) AddRef(void);
  NS_IMETHOD_(nsrefcnt) Release(void);

};

Sample::Sample()
{
  :mRefCnt(0);
}
Sample::~Sample()
{
}
NS_IMETHODIMP Sample::QueryInterface(const nsIID &aIID,
                                     void **aResult)
{
  if (aResult == NULL) {
    return NS_ERROR_NULL_POINTER;
  }
  *aResult = NULL;
  if (aIID.Equals(kISupportsIID)) {
    *aResult = (void *) this;
  }
  if (aResult != NULL) {
    return NS_ERROR_NO_INTERFACE;
  }
  AddRef();
  return NS_OK;
}

NS_IMETHODIMP_(nsrefcnt) Sample::AddRef()
{
  return ++mRefCnt;
}

NS_IMETHODIMP_(nsrefcnt) Sample::Release()
{
  if (--mRefCnt == 0) {
    delete this;
    return 0;
  }
  return mRefCnt;
}



// factory implementation class for component
class SampleFactory: public nsIFactory{
private:
  nsrefcnt mRefCnt;
public:
  SampleFactory();
  virtual ~SampleFactory();

  NS_IMETHOD QueryInterface(const nsIID &aIID, void **aResult);
  NS_IMETHOD_(nsrefcnt) AddRef(void);
  NS_IMETHOD_(nsrefcnt) Release(void);

  NS_IMETHOD CreateInstance(nsISupports *aOuter, const nsIID & iid, void * *result);
  NS_IMETHOD LockFactory(PRBool lock);

};

SampleFactory::SampleFactory()
{
  mRefCnt = 0;
}
SampleFactory::~SampleFactory()
{
}
NS_IMETHODIMP SampleFactory::QueryInterface(const nsIID &aIID,
                                  void **aResult)
{
  if (aResult == NULL) {
    return NS_ERROR_NULL_POINTER;
  }
  *aResult = NULL;
  if (aIID.Equals(kISupportsIID)) {
    *aResult = (void *) this;
  }
  else
  if (aIID.Equals(kIFactoryIID)) {
    *aResult = (void *) this;
  }
  if (aResult != NULL) {
    return NS_ERROR_NO_INTERFACE;
  }
  AddRef();
  return NS_OK;
}

NS_IMETHODIMP_(nsrefcnt) SampleFactory::AddRef()
{
  return ++mRefCnt;
}

NS_IMETHODIMP_(nsrefcnt) SampleFactory::Release()
{
  if (--mRefCnt == 0) {
    delete this;
    return 0;
  }
  return mRefCnt;
}


NS_IMETHODIMP
SampleFactory::CreateInstance(nsISupports *aOuter, const nsIID & iid, void * *result)
{
    if (!result)
        return NS_ERROR_INVALID_ARG;

    Sample* sample = new Sample();
    if (!sample)
        return NS_ERROR_OUT_OF_MEMORY;

    nsresult rv = sample->QueryInterface(iid, result);

    if (NS_FAILED(rv)) {
        *result = nsnull;
        delete sample;
    }

    return rv;
}


NS_IMETHODIMP
SampleFactory::LockFactory(PRBool lock)
{
    return NS_ERROR_NOT_IMPLEMENTED;
}





// Module implementation
class SampleModule : public nsIModule
{
public:
    SampleModule();
    virtual ~SampleModule();
    //  nsISupports methods:
    NS_IMETHOD QueryInterface(const nsIID & uuid, void * *result);
    NS_IMETHOD_(nsrefcnt) AddRef(void);
    NS_IMETHOD_(nsrefcnt) Release(void);

    //  nsIModule methods:
    NS_IMETHOD GetClassObject(nsIComponentManager *aCompMgr, const nsCID & aClass, const nsIID & aIID,
void * *aResult);
    NS_IMETHOD RegisterSelf(nsIComponentManager *aCompMgr, nsIFile *aLocation, const char *aLoaderStr,
const char *aType);
    NS_IMETHOD UnregisterSelf(nsIComponentManager *aCompMgr, nsIFile *aLocation, const char *aLoaderStr);
    NS_IMETHOD CanUnload(nsIComponentManager *aCompMgr, PRBool *_retval);

private:
    nsrefcnt mRefCnt;
};


//----------------------------------------------------------------------

SampleModule::SampleModule()
{
    mRefCnt = 0;
}

SampleModule::~SampleModule()
{
}


// nsISupports implemention
NS_IMETHODIMP_(nsrefcnt)
SampleModule::AddRef(void)
{
  ++mRefCnt;
  return mRefCnt;
}


NS_IMETHODIMP_(nsrefcnt)
SampleModule::Release(void)
{
  --mRefCnt;
  if (mRefCnt == 0) {
    mRefCnt = 1; /* stabilize */
    delete this;
    return 0;
  }
  return mRefCnt;
}

NS_IMETHODIMP
SampleModule::QueryInterface(REFNSIID aIID, void** aInstancePtr)
{
  if ( !aInstancePtr )
    return NS_ERROR_NULL_POINTER;

  nsISupports* foundInterface;

  if ( aIID.Equals(kIModuleIID) )
      foundInterface = (nsIModule*) this;
  else if ( aIID.Equals(kISupportsIID) )
    foundInterface = (nsISupports*) this;

  else
    foundInterface = 0;

  if (foundInterface) {
      foundInterface->AddRef();
      *aInstancePtr = foundInterface;
      return NS_OK;
  }

  *aInstancePtr = foundInterface;
  return NS_NOINTERFACE;
}


// Create a factory object for creating instances of aClass.
NS_IMETHODIMP
SampleModule::GetClassObject(nsIComponentManager *aCompMgr,
                             const nsCID& aClass,
                             const nsIID& aIID,
                             void** result)
{

    if (!kSampleCID.Equals(aClass))
        return NS_ERROR_FACTORY_NOT_REGISTERED;

    if (!result)
        return NS_ERROR_INVALID_ARG;

    SampleFactory* factory = new SampleFactory();
    if (!factory)
        return NS_ERROR_OUT_OF_MEMORY;

    nsresult rv = factory->QueryInterface(aIID, result);

    if (NS_FAILED(rv)) {
        *result = nsnull;
        delete factory;
    }

    return rv;
}


//----------------------------------------


NS_IMETHODIMP
SampleModule::RegisterSelf(nsIComponentManager *aCompMgr,
                           nsIFile* aPath,
                           const char* registryLocation,
                           const char* componentType)
{

    nsIComponentRegistrar* compReg = nsnull;

    nsresult rv = aCompMgr->QueryInterface(kIComponentRegistrarIID, (void**)&compReg);
    if (NS_FAILED(rv))
        return rv;
    rv = compReg->RegisterFactoryLocation(kSampleCID,
                                          "Sample Class",
                                          nsnull,
                                          aPath,
                                          registryLocation,
                                          componentType);

    compReg->Release();

    return rv;
}

NS_IMETHODIMP
SampleModule::UnregisterSelf(nsIComponentManager* aCompMgr,
                             nsIFile* aPath,
                             const char* registryLocation)
{

    nsIComponentRegistrar* compReg = nsnull;

    nsresult rv = aCompMgr->QueryInterface(kIComponentRegistrarIID, (void**)&compReg);
    if (NS_FAILED(rv))
        return rv;

    rv = compReg->UnregisterFactoryLocation(kSampleCID, aPath);

    compReg->Release();

    return rv;
}

NS_IMETHODIMP
SampleModule::CanUnload(nsIComponentManager *aCompMgr, PRBool *okToUnload)
{
    *okToUnload = PR_FALSE;  // we do not know how to unload.
    return NS_OK;
}

//----------------------------------------------------------------------

extern "C" NS_EXPORT nsresult NSGetModule(nsIComponentManager *servMgr,
                                          nsIFile* location,
                                          nsIModule** return_cobj)
{
    nsresult rv = NS_OK;

    // Create and initialize the module instance
    SampleModule *m = new SampleModule();
    if (!m) {
        return NS_ERROR_OUT_OF_MEMORY;
    }

    // Increase refcnt and store away nsIModule interface to m in return_cobj
    rv = m->QueryInterface(kIModuleIID, (void**)return_cobj);
    if (NS_FAILED(rv)) {
        delete m;
    }
    return rv;
}
```

CHAPTER 5

Tutorial: Using XPCOM Utilities To Make Things Easier

Topics covered in this chapter:

- “Generic XPCOM Module Macros”
- “String Classes in XPCOM”
- “Smart Pointers”
- “weblock2.cpp”

This chapter goes back over the code you’ve already created in the first part of the tutorial (see “webLock1.cpp” in the previous chapter) and uses XPCOM tools that make coding a lot easier and more efficient. It also introduces a basic string type that is used with many of the APIs in both XPCOM and Gecko.

To begin with, the first section describes *C++ macros* that can replace a lot of the code in the *webLock1.cpp*. Much of the code created to get the software recognized and registered as a component can be reduced to a small data structure and a single macro.

XPCOM Macros

The XPCOM framework includes a number of macros for making C++ development easier. Though they overlap somewhat (e.g., high-level macros expand to other macros), they fall into the following general categories.

- "Generic XPCOM Module Macros"
- "Common Implementation Macros"
- "Declaration Macros"

Generic XPCOM Module Macros

The work in the *Getting Started* chapter was useful in setting up the generic component code. But there are only a few places in that code that are unique to the **WebLock** component, and it was a lot of typing. To write a different component library, you could copy the listing at the end of the chapter, change very little, and paste it into a new project. To avoid these kinds of redundancies, to regulate the way generic code is written, and to save typing, XPCOM provides *generic module macros* that expand into the module code you've already seen.

Since these macros expand into "generic" implementations, they may not offer as much flexibility as you have when you are writing your own implementation. But they have the advantage of allowing much more rapid development. To get an idea about how much can be handled with the macros described in this section, compare the code listing in the "weblock2.cpp" section at the end of the chapter with "webLock1.cpp" in the previous.

The module macros include one set of macros that define the exported `NSGetModule` entry point, the required `nsIModule` implementation code and another that creates a generic factory for your implementation class. Used together, these macros can take care of a lot of the component implementation code and leave you working on the actual logic for your component.

> Note that all of the macros described in this section are similar, but used in slightly different situations. Some differ only in whether or not a method is called when the module is created and/or destroyed. *Table 1* lists the macros discussed in this section.

TABLE 1. **XPCOM Module Macros**

Macro	**Description**
`NS_IMPL_NSGETMODULE (name, components)`	Implements the `nsIModule` interface with the module name of _name and the component list in _components.
`NS_IMPL_NSGETMODULE_WITH_CTOR (name, components, ctor)`	Same as above but allows for a special function to be called when the module is created.
`NS_IMPL_NSGETMODULE_WITH_DTOR (name, components, dtor)`	Same as the first macro but allows for a special function to be called when the module is destroyed.
`NS_IMPL_NSGETMODULE_WITH_CTOR_DTOR (name, components, ctor, dtor)`	This combines the last to macros so that you can define functions to be called at the construction and destruction of the module object.

Module Implementation Macros. The general case is to use `NS_IMPL_NSGETMODULE`, which doesn't take any callbacks, but all the macros follow the same general pattern. All of these macros work on an array of structures represented by the `_components` parameter. Each structure describes a CID that is to be registered with XPCOM.

The first parameter for each of these macros is an arbitrary string that names the module. In a debugging environment, this string will be printed to the screen when the component library is loaded or unloaded. You should pick a name that makes sense and helps you keep track of things. The four required parts[1] of the structure contain the following information:

- A human readable class name
- the class ID (CID)
- the contract ID (This is an optional but recommended argument.)
- a constructor for the given object

```
static const nsModuleComponentInfo components[] =
{
    { "Pretty Class Name",
      CID,
      CONTRACT_ID,
      Constructor
    },
   ....
}
```

The important thing to note in the fictitious listing above is that it can support multiple components in a module. Modules such as the networking libraries in Gecko ("necko") have over 50 components declared in a single `nsModuleComponentInfo` array like this.

The first entry of the `nsModuleComponentInfo` above is the name of the component. Though it isn't used that much internally at the present time, this name should be something that meaningfully describes the module.

The second entry of the `nsModuleComponentInfo` is the CID. The usual practice is to put the class ID (CID) into a `#define` and use the define to declare the CID in the components list. Many CIDs take the following form:

1. This section discusses the main parameters of this structure. For a complete listing of all available options you can look at the complete reference in *Appendix B*.

```
#define NS_IOSERVICE_CID                                \
{ /* 9ac9e770-18bc-11d3-9337-00104ba0fd40 */            \
    0x9ac9e770,                                         \
    0x18bc,                                             \
    0x11d3,                                             \
    {0x93, 0x37, 0x00, 0x10, 0x4b, 0xa0, 0xfd, 0x40} \
}
```

The next entry is the Contract ID string, which is also usually defined in a `#define` in a header file.

These three entries constitute the required parameters for the `RegisterFactoryLocation` method we looked at in the prior chapter. When you use these implementation macros, you must declare a constructor for the object, and this keeps you from having to write a factory object.

Factory Macros. The factory macro makes it easy to write factory implementations. Given the class name `ConcreteClass`, the factory macro declaration is:

```
NS_GENERIC_FACTORY_CONSTRUCTOR(ConcreteClass)
```

This results in a function called `ConcreteClassConstructor` that can be used in the `nsModuleComponentInfo` structure.

```
#include "nsIGenericFactory.h"

static const nsModuleComponentInfo components[] =
{
  { "Pretty Class Name",
    SAMPLE_CID,
    "@company.com/sample"
    SampleConstructor
  }
}

NS_IMPL_NSGETMODULE(nsSampleModule, components)
```

Most of the components in the Mozilla browser client use this approach.

Common Implementation Macros

Every XPCOM object implements `nsISupports`, but writing this implementation over and over is tedious. Unless you have very special requirements for managing reference counting or handling interface discovery, the *implementation macros* that XPCOM provides can be used. Instead of implementing the `nsISupports` yourself, `NS_IMPL_ISUPPORTS1` can expand to the implementation of `AddRef`, `Release`, and `QueryInterface` for any object.

```
NS_IMPL_ISUPPORTS1(classname, interface1)
```

Also, if you implement more than one interface, you can simply change the ’1’ in the macro to the number of interfaces you support and list the interfaces, separated by commas. For example:

```
NS_IMPL_ISUPPORTS2(classname, interface1, interface2)
NS_IMPL_ISUPPORTSn(classname, interface1, …, interfacen)
```

These macros automatically add the `nsISupports` entry for you, so you don’t need to do something like this:

```
NS_IMPL_ISUPPORTS2(classname, interface1, nsISupports)
```

Take a close look at the above example. Note that it uses the actual name of the interface and not an IID. Inside the macro, the interface name expands to `NS_GET_IID()`, which is another macro that extracts the IID from the generated header of the interface. When an interface is written in XPIDL, the headers include static declarations of their IIDs. On any interface that generated with XPIDL, you can call `NS_GET_IID()` to obtain the IID which is associated with that interface.

```
// returns a reference to a shared nsIID object.
static const nsIID iid1 = NS_GET_IID(nsISupports);

// constructs a new nsIID object
static const nsIID iid2 = NS_ISUPPORTS_IID;
```

In order to use `NS_IMPL_ISUPPORTSn`, you must be sure that a member variable of type `nsrefcnt` is defined and named `mRefCnt` in your class. But why even bother when you can use another macro?

Declaration Macros

NS_DECL_NSISUPPORTS declares AddRef, Release, and QueryInterface for you, and it also defines the mRefCnt required by NS_IMPL_ISUPPORTS. Furthermore, NS_DECL_ appended with any interface name in all caps will declare all of the methods of that interface for you. For example, NS_DECL_NSIFOO will declare all of the methods of nsIFoo provided that it exists and that *nsIFoo.h* was generated by the XPIDL compiler. Consider the following real class:

```
class myEnumerator : public nsISimpleEnumerator
{
public:
    NS_DECL_ISUPPORTS
    NS_DECL_NSISIMPLEENUMERATOR

    myEnumerator();
    virtual ~myEnumerator() {}
};
```

The declaration of this nsISimpleEnumerator class doesn't include any methods other than the contructor and destructor. Instead, the class uses the NS_DECL_ macro[1].

Using these declaration macros not only saves a tremendous amount of time when you're writing the code, it can also save time if you make changes to your IDL file, since the C++ header file will then automatically include the updated list of methods to be supported.

The NS_INIT_ISUPPORTS macro is also a bit of a special case. Historically, it gets called in the constructor for your class and sets mRefCnt to zero. But a change has gone into XPCOM recently that makes NS_INIT_ISUPPORTS no longer necessary: The mRefCnt type has been changed from an integer to a class that provides its own auto-initialization. If you are building with versions earilier than Mozilla 1.3, this macro is still required.

1. Note that NS_DECL_ISUPPORTS doesn't obey the general rule in which every interface has a declaration macro of the form NS_DECL_INTERFACENAME, where INTERFACENAME is the name of the interface being compiled.

Table 2 summarizes the macro usage in this portion of the *weblock.cpp* source file:

TABLE 2. **Common XPCOM Macros**

`NS_IMPL_ISUPPORTSn`	Implements nsISupports for a given class with n number of interfaces.
`NS_DECL_ISUPPORTS`	Declares methods of nsISupports including mRefCnt
`NS_INIT_ISUPPORTS`	Initalizes mRefCnt to zero. Must be called in classes constructor
`NS_GET_IID`	Returns the IID given the name of an interface. Interface must be generated by XPIDL

Using the macros described here, the code for the **WebLock** component has gone from around 340 lines of code to just under 40. Clearly from a code maintenance point of view, this kind of reduction is outstanding. The entire source file with these macros included appears in the next section, “weblock2.cpp”.

weblock2.cpp

The listing below shows the generic module code from the “webLock1.cpp” section of the previous chapter using the macros described in this chapter.

```
#include "nsIGenericFactory.h"

#define SAMPLE_CID \
{ 0x777f7150, 0x4a2b, 0x4301, \
{ 0xad, 0x10, 0x5e, 0xab, 0x25, 0xb3, 0x22, 0xaa}}

class Sample: public nsISupports {
public:
  Sample();
  virtual ~Sample();

  NS_DECL_ISUPPORTS
};

Sample::Sample()
{
  // note: in newer versions of Gecko (1.3 or later)
  // you don’t have to do this:
  NS_INIT_ISUPPORTS();
}
Sample::~Sample()
{
}

NS_IMPL_ISUPPORTS(Sample, nsISupports);

NS_GENERIC_FACTORY_CONSTRUCTOR(Sample);

static const nsModuleComponentInfo components[] =
{
  { "Pretty Class Name",
    SAMPLE_CID,
    "@company.com/sample"
    SampleConstructor
  }
};

NS_IMPL_NSGETMODULE(nsSampleModule, components)
```

Figure 1. weblock2.cpp

String Classes in XPCOM

Strings are usually thought of as linear sequences of characters. In C++, the string literal “XPCOM”, for example, consists of 6 consecutive bytes, where ‘X’ is at byte offset zero and a null character is at byte offset 5. Other kinds of strings like “wide” strings use two bytes to represent each character, and are often used to deal with Unicode strings.

The string classes in XPCOM are not just limited to representing a null terminated sequence of characters, however. They are fairly complex because they support the Gecko layout engine and other subsystems that manage large chucks of data. The string classes can support sequences of characters broken up into multiple fragments (fragments which may or may not be null terminated)[1].

All string classes in XPCOM derive from one of two abstract classes[2]: `nsAString` and `nsACString`. The former handles double byte characters, and the latter tends to be used in more general circumstances, but both of these classes define the functionality of a string. You can see these classes being passed as arguments in many of the XPCOM interfaces we’ll look at in the following chapters.

Using Strings

Explaining how all the string classes work is outside the scope of this book, but we can show you how to use strings in the **WebLock** component. The first thing to note is that the string classes themselves are not frozen, which means that you should not link against them when you can avoid it.

Linking the full string library (.lib or .a) into a component may raise its footprint by more than 100k (on Windows), which in many cases is an unacceptable gain (see the online string guide at *http://www.mozilla.org/projects/xpcom/string-guide.html*). For **WebLock**, where the string classes need to be only wrappers around already existing string data, trading advanced functionality for a much smaller footprint is the right way to go. The **WebLock** string classes don’t need to

1. The string classes may also support embedded nulls.
2. There are other abstract string classes, but they are outside the scope of this book.

append, concatenate, search, or do any other real work on the string data, they just need to represent `char*` and other data and pass them to methods that expect an `nsACString`.

nsEmbedString and nsEmbedCString

The strings used in this tutorial are `nsEmbedString` and `nsEmbedCString`, which implement the `nsAString` abstract class and the `nsACString` abstract classes, respectively. This first example shows an `nsEmbedCString` being used to pass an `nsACString` to a method that's not expected to modify the string.

```
// in IDL: method(in ACString thing);

char* str = "How now brown cow?";
nsEmbedCString data(str);
rv = object->Method(data);
```

In this next example, the method is going to set the value of the string—as it might need to do when it returns the name of the current user or the last viewed URL.

```
// in IDL:  attribute ACString data;

nsEmbedCString data;
method->GetData(data);

// now to extract the data from the url class:

const char* aStringURL = data.get();
```

Note that the memory pointed to by `aStringURL` after the call to `url.get()` is owned by the URL string object. If you need to keep this string data around past the lifetime of the string object, you must make a copy.

String Size

The examples above illustrate the use of the single byte string class, `nsEmbedCString`. The double byte version, `nsEmbedString`, has the same functionality but the constructor takes and the `.get()` method returns the type `PRUnichar*`. Note that `PRUnichar` is a two byte value. In the coming chapters, you'll see examples that use this version in the **WebLock** component.

Smart Pointers

All of the interfaces that you've seen so far are reference counted. Leaking a reference by not releasing an object, as the code below demonstrates, can be a major problem.

```
{
    nsISupports* value = nsnull;
    object->method(&value);
    if (!value) return;

    ...

    if (NS_FAILED(error))
        return;   // <------------ leaks |value|
    ...

    NS_RELEASE(value);  // release our reference
}
```

A method returns an `nsISupports` interface pointer that has been reference counted before it is returned (assuming it wasn't null). If you handle an error condition by returning prematurely, whatever `value` points at will leak—it will never be deleted. This is a trivial fix in this example, but in real code, this can easily happen in "goto" constructs, or in deep nesting with early returns.

Having more than one interface pointer that needs to be released when a block goes out of scope begs for a tool that can aid the developer. In XPCOM, this tool is the `nsCOMPtr`, or *smart pointer* class, which can save you countless hours and simplify your code when you're dealing with interface pointers. Using smart pointers, the code above can be simplified to:

```
{
    nsCOMPtr<nsISupports> value;
    object->method(getter_AddRefs(value));
    if (!value) return;

    ...

    if (NS_FAILED(error))
        return;
    ...
}
```

The style or syntax may be unfamilar, but smart pointers are worth learning and using because they simplify the task of managing references. `nsCOMPtr` is a C++ template class that acts almost exactly like raw pointers, that can be compared and tested, and so on. When you pass them to a getter, you must do something special, however: You must wrap the variable with the function `getter_AddRefs`, as in the example above.

You cannot call the `nsISupports` `AddRef` or `Release` methods on a `nsCOMPtr`.. But this restriction is desirable, since the `nsCOMPtr` is handling reference counting for you. If for some reason you need to adjust the reference count, you must assign the `nsCOMPtr` to a new variable and `AddRef` that. This is a common pattern when you have a local `nsCOMPtr` in a function and you must pass back a reference to it, as in the following:

```
SomeClass::Get(nsISupports** aResult)
{
    if (! aResult)
        return NS_ERROR_NULL_POINTER;

    nsCOMPtr<nsISupports> value;
    object->method(getter_AddRefs(value));

    *aResult = value.get();
    NS_IF_ADDREF(*aResult);
    return NS_OK;
}
```

The first thing that this method does is check to see that the caller passed a valid address. If not, it doesn't even try to continue. Next, it calls another method on an object that is presumed to exist in this context. You can call a `.get()` method on the `nsCOMPtr` and have it returned for use as a raw pointer. This raw pointer can then be assigned to a variable and have its reference updated by `NS_IF_ADDREF`. Be very careful with the result of `.get()`, however. You should never call `Release` on this result because it may result in a crash. Instead, to explicitly release the object being held by a `nsCOMPtr`, you can assign zero to that pointer.

Another nice feature of smart pointers—the part that makes them smart—is that you can `QueryInterface` them quite easily. For example, there are two interfaces for representing a file on a file system, the `nsIFile` and `nsILocalFile`, and they are both implemented by an object. Although we haven't formally introduced these two interfaces, the next code sample shows how simple it is to switch between these two interface:

```
SomeClass::DoSomething(nsIFile* aFile)
{
   if (! aResult)
       return NS_ERROR_NULL_POINTER;

   nsresult rv;
   nsCOMPtr<nsILocalFile> localFile = do_QueryInterface(aFile, &rv);
```

If the `QueryInterface` is successful, `localFile` will be non-null, and `rv` will be set to `NS_OK`. If `QueryInterface` fails, `localFile` will be null, and `rv` will be set to a specific error code corresponding to the reason for the failure. In this construct, the result code `rv` is an optional parameter. If you don’t care what the error code is, you can simply drop it from the function call.

From this point on, we’ll be using `nsCOMPtrs` as much as possible in **WebLock**. For a complete listing of smart pointer functionality, see *http://www.mozilla.org/projects/xpcom/nsCOMPtr/*.

CHAPTER 6

Tutorial: Starting WebLock

In this chapter, we begin to design and implement the web locking functionality itself. We have already created a module that implements most of the generic component functionality (e.g,. registration). This chapter will focus on the functionality that actually handles the web locking.

Topics covered in this chapter:

- “Getting Called at Startup”
- “Providing Access to WebLock”
- “Creating the WebLock Programming Interface”
- “Defining the Weblock Interface in XPIDL”
- “Implementing WebLock”

Getting Called at Startup

No person is an island to themselves, and neither are components. The sample component you’ve built up so far doesn’t do anything. After having its registration procedure called, the component does nothing.

In order to be started up or notified when some event happens, the sample component has to hook into Mozilla, which it can do either by overriding an existing component or by registering for some event that will cause it to start up. **WebLock** does the latter and gets called when a Gecko Profile Startup occurs. When a Gecko application starts up, registered components are created and notified via the general purpose observer interface `nsIObserver`.

Observers are objects that are notified when various events occur. Using them is a good way for objects to pass messages to each other without the objects having explicit knowledge of one another.

Usually, there is one object notifying a group of observers. For example, an object may be created and have its `observe` method called at startup, or it may register to be notified prior to XPCOM shutdown. The method at the core of this interface is `observe`:

```
void observe( in nsISupports aSubject,
   in string aTopic,
   in wstring aData );
```

There aren't really any restrictions on what the parameters of the observer method may be. These parameters are defined according to the event being observed. For example, in the case of the XPCOM shutdown observation, `aSubject` and `aData` are not defined, and `aTopic` is defined as the string "xpcom-shutdown". If your object would like to register for this or other events, it first must implement the `nsIObserver` interface. Once you do this, the observer service implementing `nsIObserverService` can notify your object of registered events by means of this interface, as in the figure below.

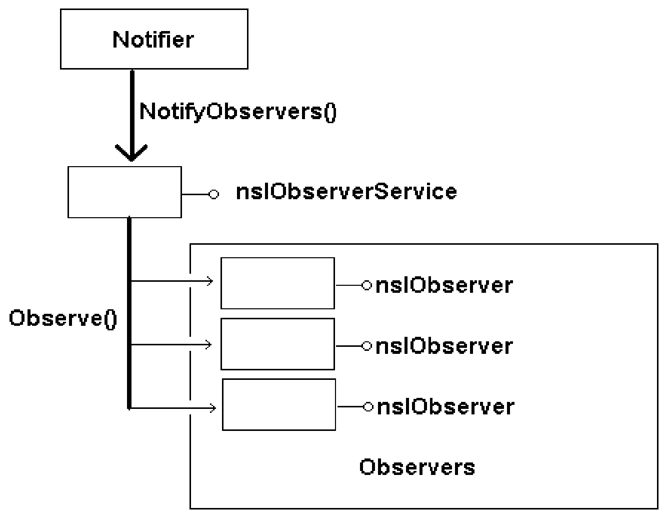


Figure 1. The Observer Interfaces

The above figure shows the observer service maintaining a list of all registered `nsIObserver` objects. When the notification is made the `nsIObserverService` broadcasts the notification from the caller of the `NotifyObserver()` to the `nsIObserver` object's `Observe()` method. This is a very useful decoupling of different objects. The `nsIObserver` is a generic interface for passing messages between two or more objects without defining a specific frozen interface, and its one of the ways in which extensibility is built into XPCOM.

The implementation of the `nsIObserver` interface in the **WebLock** component is similar to the implementation for the `nsIFactory` interface. Following Example 2, you change the class definition to support the `nsIObserver` interface and change `NS_IMPL_ISUPPORTS1` so that the `QueryInterface` implementation knows that the component also supports `nsIObserver`. The `WebLock` class definition with support for start up observation is below.

```
class WebLock: public nsIObserver {
public:
  WebLock();
  virtual ~WebLock();

  NS_DECL_ISUPPORTS
  NS_DECL_NSIOBSERVER
};

NS_IMPL_ISUPPORTS1(WebLock, nsIObserver);
```

The standard implementation of `Observe()` simply compares the `aTopic` string with the value defined by the event the object is expecting. When there is a match, you can handle the event any way you see fit. If the object has only registered for one notification, then you can ignore the `aTopic` string and simply handle the event as it occurs. In other words, the `Observe` method should never be called in response to some event for which the object is not registered.

```
NS_IMETHODIMP
WebLock::Observe(nsISupports *aSubject,
                 const char *aTopic,
                 const PRUnichar *aData)
{
    return NS_OK;
}
```

Notification via the observer service is somewhat indirect. The only way to register directly for a notification via the observer service is to instantiate an `nsIObserver` object. This works for most cases, but consider the case when you have this notification create a component. Since the component hasn't been created yet, there are no instantiated `nsIObserver` objects that can be passed into the `nsIObserverService`, nor can the component code do anything until it is loaded.

Registering for Notifications

The `nsIObserverService` interface has methods for registering and unregistering an `nsIObserver` object. These two methods are used to dynamically add or remove an observer to a notification topic. But **WebLock** needs to be instantiated and added to the observer service automatically, which also implies some sort of persistent data (after all, we want to have the component start up every time the application does).

This is where a new service that manages sets of related data comes in handy. This service, the `nsICategoryService`, is what XPCOM and Gecko embedding applications use to persist lists of `nsIObserver` components that want to have startup notification.

The `nsICategoryService` maintains sets of name-value pairs like the one in *Figure 2*.

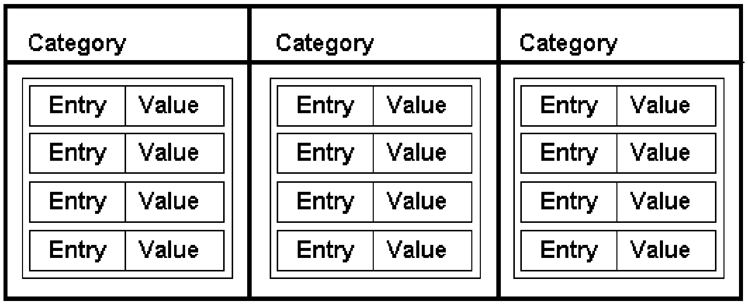


Figure 2. The Category Manager

Every category is identified by a string that represents the name of the category. Each category contains a set of name-value pairs. For example, you might have a category named “Important People”in which the name-value pairs would be names and phone numbers. The format of the name-value pair is left up to you.

This data structure is more than enough to support the persisting of components that what to be started up. The category name also maps nicely onto the notion of a notification “topic.” The topic name could be something like “xpcom-startup”, for instance, and the name-value pair could contain the Contract IDs required to create the components requesting startup. In fact, this is exactly how categories are used to handle registration with XPCOM for startup notification. You will see the code which does this in the next section.

Getting Access to the Category Manager

Two fields in the nsModuleComponentInfo structure introduced in the last section are addresses for registration and unregistration callbacks. The first callback is called when the component's nsIModule::RegisterSelf method is called. This callback allows the component to execute any one-time registration code it may need. The inverse of this function is the unregistration callback, where it's a good idea to undo whatever the registration function did. The two functions look like this:

```
static NS_METHOD
WebLockRegistration(nsIComponentManager *aCompMgr,
                    nsIFile *aPath,
                    const char *registryLocation,
                    const char *componentType,
                    const nsModuleComponentInfo *info)

static NS_METHOD
WebLockUnregistration(nsIComponentManager *aCompMgr,
                      nsIFile *aPath,
                      const char *registryLocation,
                      const nsModuleComponentInfo *info)
```

The names of the functions can be anything you wish. Both functions are passed the Component Manager and the path to the component, including the opaque registryLocation. These are also parameters in the nsIModule implementation in Example 1. In addition to these parameters, the callback functions are passed the nsModuleComponentInfo struct, which is the same structure initially passed into NS_IMPL_NSGETMODULE.

During registration, the registration callback is where you get the nsICategoryManager. Once you have it, you can add the component to the category of components that get started automatically. As a service, the nsICategoryManager is accessible via the nsIServiceManager. Also note that the nsIComponentManager is passed into the callback. Since the object that implements the nsIComponentManager interface also implements nsIServiceManager, all you have to do is QueryInterface the nsIComponentManager to nsIServiceManager to get the Service Manager. You can then use the Service Manager to add the component to the category:

```
nsresult rv;

nsCOMPtr<nsIServiceManager> servman =
     do_QueryInterface((nsISupports*)aCompMgr, &rv);

if (NS_FAILED(rv))
     return rv;
```

do_QueryInterface

The previous code uses the special `nsCOMPtr` function `do_QueryInterface` that lets you `QueryInterface` without having to worry about reference counting, error handling, and other overhead. The `do_QueryInterface` knows what interface to QI to based on the `nsCOMPtr` that is being assigned into. We could have just as easily have used the raw `QueryInterface()` method, but using `nsCOMPtr` is much more economical (see “Smart Pointers” on page 86).

Once you have a `nsIServiceManager` reference, you can ask it for the service you are interested in. This process is similar to using `CreateInstance` from the `nsIComponentManager`, but there is no aggregation parameter since the object has already been constructed.

```
nsCOMPtr<nsICategoryManager> catman;
rv = servman->GetServiceByContractID(NS_CATEGORYMANAGER_CONTRACTID,
                                     NS_GET_IID(nsICategoryManager),
                                     getter_AddRefs(catman));
if (NS_FAILED(rv))
     return rv;
```

There are two service getters on the `nsIServiceManager` interface: one that takes a CID and another interface that takes a Contract ID. Here we’ll use the latter. The first parameter to the `GetServiceByContractID` is of course the contract ID, which is defined in the *nsXPCOM.h* header file. The next parameter is a nifty macro that returns the IID for the interface name that you pass in. The last parameter assigns an out interface pointer to a `nsCOMPtr`. Assuming there weren't any unexpected errors, the variable `catman` holds the `nsICategoryManager` interface pointer, which you can use to add the component as a startup observer by calling a method on the `nsICategoryManager`.

The next step is to figure out which parameters to pass to the method. There is a category name and a name-value pair, but since the name-value pair meaning is category specific, you need to figure out which category to use.

There are two startup notifications, both of which create the observer if it isn't already created. The first is provided by XPCOM. This notification will occur during initalization of XPCOM, where all XPCOM services are guaranteed to be available during the calls. Embedding applications may provide other notifications.

TABLE 1. Common XPCOM Notifications

Category	Name	Value	Creates Component
xpcom-startup	Any	Contract ID	Yes
xpcom-shutdown	Any	Contract ID	No
xpcom-autoregistration	Any	Contract ID	No
app-startup	Any	service, Contract ID	*

The table above summaries the popular persistent notifications registered through the category manager. The name of the category itself is a well defined string, but the name-value pairs can be anything.

When naming your component in the category, take care to use something that means something and doesn't muddy up the namespace. In this case, "WebLock" is unique and provides context to anyone looking at the category. The value of the name-value part is expected to be the contract ID of the component.

Since every category can define the name-value pairs, the application "app-startup" category can support not only services but component instances as well. For the app-startup notification, you must explicitly pass the string "service," prior to the component's Contract ID. If you do not, the component will be created and then released after the notification, which may cause the component to be deleted.

In short, to register the **WebLock** component as an xpcom-startup observer, do the following:

```
nsEmbedCString previous;
rv = catman->AddCategoryEntry("xpcom-startup",
                              "WebLock",
                              WebLock_ContractID,
                              PR_TRUE,  // persist category
                              PR_TRUE,  // replace existing
                              previous);
```

The unregistration, which should occur in the unregistration callback, looks like this:

```
rv = catman->DeleteCategoryEntry("xpcom-startup",
                                 "WebLock",
                                  PR_TRUE);  // persist
```

A complete code listing for registering WebLock as a startup observer follows.

```
#define MOZILLA_STRICT_API

#include "nsIGenericFactory.h"

#include "nsCOMPtr.h"
#include "nsXPCOM.h"
#include "nsIServiceManager.h"
#include "nsICategoryManager.h"

#include "nsIObserver.h"

#include "nsEmbedString.h"

#define WebLock_CID \
{ 0x777f7150, 0x4a2b, 0x4301, \
{ 0xad, 0x10, 0x5e, 0xab, 0x25, 0xb3, 0x22, 0xaa}}

#define WebLock_ContractID "@dougt/weblock"

class WebLock: public nsIObserver {
public:
  WebLock();
  virtual ~WebLock();

  NS_DECL_ISUPPORTS
  NS_DECL_NSIOBSERVER
};

WebLock::WebLock()
{
  NS_INIT_ISUPPORTS();
}

WebLock::~WebLock()
```

```
{
}

NS_IMPL_ISUPPORTS1(WebLock, nsIObserver);

NS_IMETHODIMP
WebLock::Observe(nsISupports *aSubject, const char *aTopic, const PRUnichar
*aData)
{
    return NS_OK;
}

static NS_METHOD WebLockRegistration(nsIComponentManager *aCompMgr,
                                          nsIFile *aPath,
                                          const char *registryLocation,
                                          const char *componentType,
                                          const nsModuleComponentInfo *info)
{
    nsresult rv;

    nsCOMPtr<nsIServiceManager> servman =
do_QueryInterface((nsISupports*)aCompMgr, &rv);
    if (NS_FAILED(rv))
        return rv;


    nsCOMPtr<nsICategoryManager> catman;
    servman->GetServiceByContractID(NS_CATEGORYMANAGER_CONTRACTID,
                                    NS_GET_IID(nsICategoryManager),
                                    getter_AddRefs(catman));

    if (NS_FAILED(rv))
        return rv;

    char* previous = nsnull;
    rv = catman->AddCategoryEntry("xpcom-startup",
                                  "WebLock",
                                  WebLock_ContractID,
                                  PR_TRUE,
                                  PR_TRUE,
                                  &previous);
    if (previous)
        nsMemory::Free(previous);

    return rv;
}

static NS_METHOD WebLockUnregistration(nsIComponentManager *aCompMgr,
                                        nsIFile *aPath,
                                        const char *registryLocation,
                                        const nsModuleComponentInfo *info)
{
    nsresult rv;

    nsCOMPtr<nsIServiceManager> servman =
do_QueryInterface((nsISupports*)aCompMgr, &rv);
    if (NS_FAILED(rv))
        return rv;


    nsCOMPtr<nsICategoryManager> catman;
    servman->GetServiceByContractID(NS_CATEGORYMANAGER_CONTRACTID,
                                    NS_GET_IID(nsICategoryManager),
                                    getter_AddRefs(catman));

    if (NS_FAILED(rv))
        return rv;

    rv = catman->DeleteCategoryEntry("xpcom-startup",
                                     "WebLock",
                                     PR_TRUE);

    return rv;
}



NS_GENERIC_FACTORY_CONSTRUCTOR(WebLock)

static const nsModuleComponentInfo components[] =
{
  { "WebLock",
    WebLock_CID,
    WebLock_ContractID,
    WebLockConstructor,
    WebLockRegistration,
    WebLockUnregistration
  }
};

NS_IMPL_NSGETMODULE(WebLockModule, components)
```

Providing Access to WebLock

At this point, the component will be called when XPCOM starts up. **WebLock** has already implemented the `nsISupports`, `nsIFactory`, `nsIModule`, and `nsIObserver` interfaces that handle generic component functionality including being initialized at startup. And it speaks to the Component Manager, Service Manager, Category Manager, and the Component Registrar to register itself properly with XPCOM.

The next step is to expose additional functionality to Gecko applications and other clients to query and control the **WebLock** component. For example, the user interface needs to be able to enable and disable the web locking functionality, see what sites are in the whitelist, and add or remove sites from that list. **WebLock** needs to provide an API, and it needs to hook into Gecko in order to implement the actual locking functionality.

The WebLock User Interface

The **WebLock** component in this tutorial uses XUL to define the additional browser UI in a cross-platform way, and XUL uses JavaScript to access and control XPCOM components, but Gecko's pluggable UI allows any user interface to call into Gecko and the components you create as easily as you can from XUL. See "XUL" on page 149 for a discussion of how XUL interacts with JavaScript and XPCOM.

Creating the WebLock Programming Interface

Design is one of the hardest parts of any programming problem. The question the interface for the **WebLock** component must answer is: How should **WebLock** look to the outside world? What, in other words, is the interaction of clients with the **WebLock** component? In this section, we enumerate the basic functionality the component should expose and create the single interface that organizes and provides this functionality.

Instead of starting with the implementation, developers use XPIDL (see "XPIDL and Type Libraries" on page 21 for more information about XPIDL) to define the interface to the component: how the functionality should be organized, expressed, and exposed to its clients.

In general, the **WebLock** service interface needs to include the following functionality.

- Lock - Enable web locking so that any browser in the Gecko application is restricted to the white list of website domains.
- Unlock - Disable web locking. This should allow any browser in the Gecko application to browse any website regardless of the white list.
- AddSite - Add the current URL to the white list.
- RemoveSite - Remove the current URL from the white list.
- EnumerateSites - Allows the enumeration of all sites in the white list. `EnumerateSites` might be used in the user interface to provide something like an editable listbox of all sites in the white list.

Even this simple outline presents some ambiguity, however. It’s certainly not enough to spell out the interface for the **WebLock** component in this way. For example, `AddSite` is supposed to add the current URL to the white list, but is the URL an input parameter to the method, is it the topmost web page in the Gecko application, or is it something more random—a URL picked from global history or that's been given context in some other way?

As a strongly typed and implementation-agnostic language, XPIDL requires that you be quite specific about the APIs, the list of parameters, their order, and their types. XPIDL requires that you spell it all out, in other words. And it’s this formality that makes the interfaces in XPCOM effective contracts between services and clients.

The next section shows the interface of the **WebLock** component, `iWebLock`, in XPIDL. Once the interface has been described in the XPIDL language, the interface file can be used to generate the header files needed for the implementation code, the binary type library files that let you use the interface of the **WebLock** component from JavaScript, and even javadoc style HTML documentation.

Defining the Weblock Interface in XPIDL

Most interfaces in the XPCOM world are described in XPIDL. The XPIDL file for the `iWebLock` interface can be used to generate the C++ header file, which you’ll need to implement the interface in the component and also a type library that makes the component accessible from JavaScript or other interpreted languages. In Mozilla, JavaScript is the bridge between components and the XUL-based user interface.

The XPIDL Syntax

The XPIDL syntax is a mix of C++ and Java, and of course it’s very much like the OMG IDL upon which it is closely based. The XPIDL for `iWebLock` appears in *Figure 3*.

```
#include "nsISupports.idl"
interface nsISimpleEnumerator;
[scriptable, uuid(ea54eee4-9548-4b63-b94d-c519ffc91d09)]
interface iWeblock : nsISupports
{
  void lock();
  void unlock();

  // assume strings are UTF-8
  void addSite(in string url);
  void removeSite(in string url);
  attribute nsISimpleEnumerator sites;
};
```

Figure 3. iWebLock

The first line includes the file *nsISupports.idl*, which defines the `nsISupports` interface from which all XPCOM interfaces must derive, and makes it possible for the `iWebLock` interface to subclass that base interface.

```
#include "nsISupports.idl"
```

The next line of the XPIDL is a forward declaration of the interface `nsISimpleEnumerator`. Again, this is similar to the forward declare in C++ (except that C++ does not have the `interface` keyword seen here).

```
interface nsISimpleEnumerator;
```

See the XPCOM references in Appendix C for more information about the XPIDL syntax.

Scriptable Interfaces

The thid line in *Figure 3* is more complex. The first thing it says is that `iWebLock` will be *scriptable*.

```
[scriptable, uuid(ea54eee4-9548-4b63-b94d-c519ffc91d09)]
```

The rest of the line provides a UUID for this interface. Recall that every interface has a unique number that is assigned to it. In the case of interfaces, the identifier is an IID. In the case of the components, which also require unique identifiers, the identifier is the CID.

Subclassing nsISupports

The next line in *Figure 3* names the interface and defines its base interface. `iWeblock` derives from `nsISupports`. XPIDL has no way to define multiple inheritance–something that all scriptable objects must deal with.

```
interface iWeblock : nsISupports
```

The Web Locking Interface

The body of the block (the stuff between the curly braces) defines the methods and attributes of our interface. There are basically two functional sets on this interface. The first section of the interface controls whether or not **WebLock** checks to see if a web page can be loaded. If locked, **WebLock** will prevent sites not on the white list from loading.

```
    void lock();
    void unlock();
```

This interface does not enforce any policy with respect to how the user enables or disables this feature. This allows maximum flexibility in the implementation. Any place in the application can acquire this interface via the Service Manager and call `unlock` or `lock`. For example, the user interface may bring up a dialog asking the user for a password before calling `unlock`. Another area of code, such as a “Profile Manager” that starts up and lets users choose which profileto use, may unconditionally call `unlock` on such a component when switching a profile.

The next set of functionality manages the white list where acceptable domains are stored:

```
    void addSite(in string url);
    void removeSite(in string url);
    attribute nsISimpleEnumerator sites;
```

Operations in this set—`add`, `remove`, and `enumerate`—will be called from a user interface that manages the white list and adds the current website to the white list. There is no policy applied to what sites get added or removed to this list, or who can remove a site.

The most interesting method definition is the enumerator. First of all, it does not look like a method at all:

```
readonly attribute nsISimpleEnumerator sites;
```

This line defines an attribute in the interface. In C++, this is considered a public variable and “compiled” into a Get method (e.g., `getSites`). If an attribute is not marked `readonly`, then both Get and Set methods are generated.

The getter created by this attribute returns a `nsISimpleEnumerator` interface pointer. This interface allows you to pass a list of elements between interfaces. It has two methods: `hasMoreElements()` and `getNext()`.

```
[scriptable, uuid(D1899240-F9D2-11D2-BDD6-000064657374)]
interface nsISimpleEnumerator : nsISupports {
  /**
   * Called to determine whether or not the enumerator has
   * any elements that can be returned via getNext(). This method
   * is generally used to determine whether or not to initiate or
   * continue iteration over the enumerator, though it can be
   * called without subsequent getNext() calls. Does not affect
   * internal state of enumerator.
   *
   * @see getNext()
   * @return PR_TRUE if there are remaining elements
   * in the enumerator.
   *         PR_FALSE if there are no more elements in the enumerator.
   */
  boolean hasMoreElements();

  /**
   * Called to retrieve the next element in the enumerator. The "next"
   * element is the first element upon the first call. Must be
   * pre-ceeded by a call to hasMoreElements() which returns PR_TRUE.
   * This method is generally called within a loop to iterate over
   * the elements in the enumerator.
   *
   * @see hasMoreElements()
   * @return NS_OK if the call succeeded in returning a non-null
   *               value through the out parameter.
   *         NS_ERROR_FAILURE if there are no more elements
   *                          to enumerate.
   * @return the next element in the enumeration.
   */
  nsISupports getNext();
};
```

Implementing WebLock

Once you have defined the interfaces that the component will implement, you can begin to write the implementation code that will actually carry out the web locking functionality.

The **WebLock** component implements three interfaces:

- `nsISupports`
- `nsIObserver`
- `iWebLock`

`nsISupports` is the base interface that all XPCOM objects must implement. The `nsIObserver` interface is for listening to various events that Gecko generates. And the `iWebLock` interface is the interface that actually controls the web locking functionality. The first two have already been implemented as part of the generic module code. Recall from the *Tools* chapter that implementing these basic interfaces can be easy and straightforward if you use the macros and other utilities that XPCOM provides.

Declaration Macros

The class declaration for the `WebLock` class that implements these three interfaces is as follows:

```
class WebLock: public nsIObserver, public iWebLock
{
  public:
    WebLock();
    virtual ~WebLock();

    NS_DECL_ISUPPORTS
    NS_DECL_NSIOBSERVER
    NS_DECL_IWEBLOCK
};
```

Note that we derive from the `nsIObserver` interface as well as the `iWeblock` class. We do not need to explicitly derive from `nsISupports` as both of these two other interfaces are already subclasses of `nsISupports`:

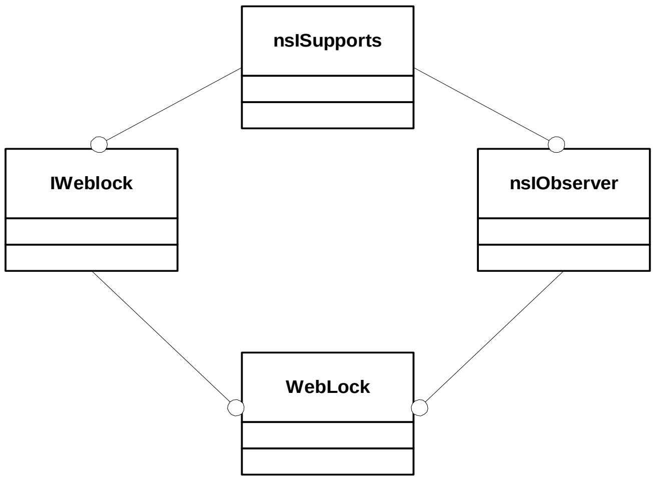


Figure 4. Interface Hierarchy for WebLock

The body of the class declaration uses declaration macros that are generated from an XPIDL interface file. Every header generated from an XPIDL file has a similar macro that defines all the methods in that interface. This makes changes to the interface when designing a bit simpler, as you do not have to modify any class declarations.

There are times, of course, when you cannot use these macros—as when two interfaces share the same method signatures. In these cases you have to manually declare the methods in your class. But in practice, manually declaring class methods in XPCOM is the exception and not the rule. The `NS_DECL_IWEBLOCK` declaration macro expands into the following:

```
  NS_IMETHOD Lock(void);
  NS_IMETHOD Unlock(void);
  NS_IMETHOD AddSite(const char *url);
  NS_IMETHOD RemoveSite(const char *url);
  NS_IMETHOD GetSites(nsISimpleEnumerator * *aSites);
  NS_IMETHOD GetSites(nsISimpleEnumerator * *aSites);
  NS_IMETHOD SetSites(nsISimpleEnumerator *aSites);
```

Representing Return Values in XPCOM

The code sample above is the C++ version of the `iWebLock` interface methods. The return result of XPCOM methods generated from XPIDL is always of the type `nsresult`, and the small macro used in these expansions, `NS_IMETHOD`, actually represents that return type. `nsresult` is returned even when in XPIDL you specify that the method return a `void`. If you require the return result to be something else, the methods are not truly XPCOM methods. If you really want to change the return result type you can use a special flag in your XPIDL that denotes this (see the XPIDL reference at *http://www.mozilla.org/scriptable/xpidl/*). However, we suggest that you simply add an out parameter to the method.

XPIDL Code Generation

The XPIDL compiler also generates a stub implementation of the interface in a commented section of the generated header file, in which each method returns `NS_ERROR_NOT_IMPLEMENTED`. If you copy the stub implementation from the header file into the source, then rename the dummy class name ("`_MYCLASS_`") to the `WebLock` class name already defined, you should be able to compile the source successfully.

Getting the WebLock Service from a Client

At this point, you can install the XPCOM component and have other systems use it. The component doesn't do anything useful, of course, but you have written enough of the code to have it recognized and accessed as a component in XPCOM. The code snippet below illustrates how to get the **WebLock** service when the component is present:

```
nsCOMPtr<nsIServiceManager> servMan;
nsresult rv = NS_GetServiceManager(getter_AddRefs(servMan));
if (NS_FAILED(rv))
{
    printf("ERROR: XPCOM error [%x].\n", rv);
    return -1;
}
nsCOMPtr<iWebLock> weblock;
rv = servMan->GetServiceByContractID("@dougt/weblock",
    NS_GET_IID(iWeblock), getter_AddRefs(weblock));

if (NS_FAILED(rv))
{
    printf("ERROR: XPCOM obtaining service [%x].\n", rv);
    return -1;
}
```

Implementing the iWebLock Interface

Once the interface is defined, you can focus on implementing the web lock startup functionality itself. The **WebLock** component starts automatically when XPCOM is started up because it's been registered as a category in XPCOM. When **WebLock** is called, one of the first things it wants to do is read in a file that lists the URLs that the browser is allowed to load. This file can exist anywhere on the local system, but we've placed it next to the application to keep things simple. The first step in this implementation phase, then, is to create the functionality that accesses this **WebLock** white list and uses its data to determine which domains are allowed and which are to be blocked. For this, we need to use the file interfaces available in XPCOM.

File Interfaces. Files and directory are abstracted and encapsulated by interfaces. There are a few reasons for not using strings to represent file locations, but the most important one is that not all file systems can be represented by a series of characters separated by a slash. On the Macintosh platform, for example, files are represented as a triplet—two numbers and one string—so using a string on the Macintosh does not adequately identify files on that operating system.

`nsIFile`, the file interface in XPCOM, provides most of the functionally that file handling requires. That interface includes members representing the file name, file attributes, permissions, existence, and others. A related interface called `nsILocalFile` provides access to operations specific to local files, but the `nsIFile` functionality is adequate for the **WebLock** component.

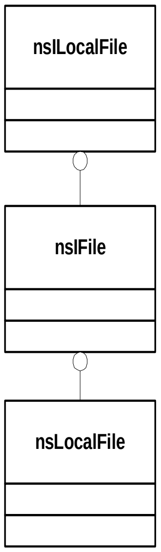


Figure 5. File Interface Hierarchy

Remote Files and nsIFile

It is not inconceivable for remote files to be represented by the `nsIFile` interface. Someone could write an `nsIFile` implementation that represented FTP files on some server. The existing code would need to change very little for a **WebLock** implementation to take advantage of files that do not actually exists on disk. This kind of implementation does not exist, but this expandability shows some of the flexibility that interface-based programming can provide.

Appendix B, The XPCOM API Reference, contains detailed information on `nsIFile` and other XPCOM interfaces.

The Directory Service

The file interfaces are most useful when you can use them to find and manipulate files that are relative to the application. The Directory Service provides directory and file locations in a cross platform uniform way to make this easier. This service, available as `nsIDirectoryService`, stores the location of various common system locations, such as the the directory containing the running process, the user's `HOME` directory, and others. It can be expanded so that applications and components can define and store their own special locations—an application plugin directory, for example, preference files and/or directories, or other application specific paths. For example, to expose the location of the "white list" file containing all of the URL's that are safe for **WebLock**, you can add its location to the `nsDirectoryService`, which clients can then query for this infomation.

The Directory Service implements the `nsIProperties` interface, which allows you to `Get()`, `Set()`, and `Undefine()` interface pointers. In the case of **WebLock**, these interface pointers will be `nsIFile` objects.

```
[scriptable, uuid(78650582-4e93-4b60-8e85-26ebd3eb14ca)]
interface nsIProperties : nsISupports
{
    /**
     * Gets a property with a given name.
     *
     * @return NS_ERROR_FAILURE if a property with that
     * name doesn’t exist.
     * @return NS_ERROR_NO_INTERFACE if the
     * found property fails to QI to the
     * given iid.
     */
    void get(in string prop, in nsIIDRef iid,
             [iid_is(iid),retval] out nsQIResult result);

    /**
     * Sets a property with a given name to a given value.
     */
    void set(in string prop, in nsISupports value);

    /**
     * Returns true if the property with the given name exists.
     */
    boolean has(in string prop);

    /**
     * Undefines a property.
     * @return NS_ERROR_FAILURE if a property with that name doesn’t
     * already exist.
     */
    void undefine(in string prop);

    /**
     *  Returns an array of the keys.
     */
   void getKeys(out PRUint32 count, [array, size_is(count), retval]
      out string keys);
};
```

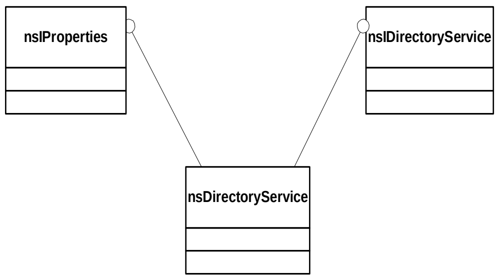


Figure 6. Directory Service Hierarchy

There are two steps involved to find directories or files with the Directory Service (`nsIDirectoryService`). You must know the string key (or property) that refers to the location you are interested in, which is published in the file *nsDirectoryServiceDefs.h* that comes with the Gecko SDK (for a listing of these locations, see *Appendix B*, *the XPCOM API Reference*). The string key for the directory containing the application executable is `NS_XPCOM_CURRENT_PROCESS_DIR`. Given this key, you can acquire the directory service, call `Get()`, and pass the key. In the example below, `theFile` will point to the directory that contains the executable.

```
nsCOMPtr<nsIServiceManager> servMan;
nsresult rv = NS_GetServiceManager(getter_AddRefs(servMan));
if (NS_FAILED(rv)) return -1;

nsCOMPtr<nsIProperties> directoryService;
rv = servMan->GetServiceByContractID(
    NS_DIRECTORY_SERVICE_CONTRACTID,
    NS_GET_IID(nsIProperties),

getter_AddRefs(directoryService));

if (NS_FAILED(rv)) return -1;

nsCOMPtr<nsIFile> theFile;
rv = directoryService->Get(NS_XPCOM_CURRENT_PROCESS_DIR,
                           NS_GET_IID(nsIFile),
                           getter_AddRefs(theFile));

if (NS_FAILED(rv)) return -1;
```

Most of the useful functionality is exposed by the `nsIProperties` interface, but the directory service also implements `nsIDirectoryService`. This interface allows you to extend and override `nsIFile` objects registered with the directory service. There are currently two ways to add a file location to the directory service: directly and using the delayed method. The direct method is to add a new `nsIFile` object using the `nsIProperties` interface, in which case you pass the `nsIFile` object as an `nsISupports` to the `Set()` method of the `nsIProperties` interface.

In the delayed method, you register to be a callback that can provide an `nsIFile`. To do this, you must get the implementation like we did above. When you have it, `QueryInterface` for the `nsIDirectoryService` interface. In this interface, there is a function which allows you to register an `nsIDirectoryServiceProvider` interface. The interface callback looks like this:

```
[scriptable, uuid(bbf8cab0-d43a-11d3-8cc2-00609792278c)]
interface nsIDirectoryServiceProvider: nsISupports
{
 /**
  * getFile
  *
  * Directory Service calls this when it gets the first request for
  * a prop or on every request if the prop is not persistent.
  *
  * @param prop         The symbolic name of the file.
  * @param persistent   TRUE - The returned file will be cached by Directory
  *                     Service. Subsequent requests for this prop will
  *                     bypass the provider and use the cache.
  *                     FALSE - The provider will be asked for this prop
  *                     each time it is requested.
  *
  * @return             The file represented by the property.
  *
  */
  nsIFile getFile(in string prop, out PRBool persistent);
};
```

Modifying Paths with nsIFile

The directory service returns an `nsIFile` object, but that object points to the application directory and not the file itself. To modify this `nsIFile` so that it points to the file, you must call the `Append` method of the `nsIFile`. `Append` adds the input string to the path already specified in the `nsIFile`. On Unix, for example, calling `Append("b")` on an `nsIFile` modifies that `nsIFile` representing */u/home/dougt/a* to point to */u/home/dougt/a/b*. The next operation on the `nsIFile` returns results associated with the “b” path. If “a” wasn’t a directory, further operations would fail, even if the initial `Append` was successful. This is why `Append` is considered a string operation.

The **WebLock** component manipulates a file named *weblock.txt*. The following snippet adjusts the `theFile` object representing that file:

```
nsEmbedCString fileName("weblock.txt");
appDir->AppendNative(fileName);
```

Manipulating Files with nsIFile

Once you have an `nsIFile` object pointing to the file that you’re interested in, you can open it and read its contents into memory. There are many ways to do this: You can use Standard ANSI File I/O, or NSPR (see the sidebar “The Netscape Portable Runtime Library” below for a brief description of NSPR), or you can use the networking APIs that Gecko provides.

The Netscape Portable Runtime Library

The *Netscape Portable Runtime Library* (NSPR) is a platform-independent library that sits below XPCOM. As a layer of abstraction above the operating system, the NSPR allows Gecko applications to be platform independent by providing the following system-level facilities:

- Threads
- Thread synchronization
- File and network I/O
- Timing and intervals
- Memory management
- Shared library linking

The NSPR is included in the Gecko SDK.

To keep things as simple as possible, we’ll read the file into memory using standard ANSI file I/O, but for examples and information about how to use *necko*, the Gecko networking libraries, see *http://www.mozilla.org/projects/netlib/*.

Using nsILocalFile for Reading Data

An `nsIFile` object returned from the directory service may also implement the `nsILocalFile` interface, which has a method that will return a `FILE` pointer that can be used in `fread()`. To implement the actual read, you need to allocate a buffer the length of the file, use the `nsILocalFile` interface pointer to obtain a `FILE *`, use this result with `fread`, and close the file pointer.

The following code loads the contents of the file referenced by the `nsIFile` object `theFile` into the buffer `buf`:

```
    nsCOMPtr<nsILocalFile> localFile = do_QueryInterface(theFile);
    if (!localFile) return -1;

    PRBool exists;
    rv = theFile->Exists(&exists);
    if (NS_FAILED(rv)) return -1;

    char *buf = NULL;

    if (exists)
    {
        // determine file size:
        PRUint32 fs, numread;
        PRInt64 fileSize;
        rv = theFile->GetFileSize(&fileSize);
        if (NS_FAILED(rv)) return -1;

        // Converting 64 bit value to unsigned int
        LL_L2UI(fs, fileSize);

        FILE* openFile;
        rv = localFile->OpenANSIFileDesc("rw", &openFile);
        if (NS_FAILED(rv)) return -1;

        char *buf = (char *)malloc((fs+1) * sizeof(char));
        if ( ! bug ) return -1;

        numread = fread(buf, sizeof( char ), fs, openFile);

        if (numread != fs)
            ;// do something useful.

        // ...
    }

    if (buf)
        free(buf);
```

The first line of the code calls `QueryInterface` on `theFile`, and if that succeeds assigns the new interface pointer to `localFile`. If the `QueryInterface` call fails, `localFile` will have a value of `NULL`.

Note that the out parameter of the method `GetFileSize` is a 64 bit integer. The type of this variable is `PRInt64`, but this type is not represented as a primitive on all platforms. On some platforms, `PRInt64` is a struct with two fields—a high and a low 32 bit integer. So operations on this type must use special macros that do the right thing on each platform. On windows or Linux, for example, it is possible to multiply a `PRInt64` by a long like this:

```
PRInt64 x = 1, y = 2;
y = x * 2;
```

However, this same snippet will not compile on a platform like Macintosh OS 9, where you need to use macros to perform the calculation:

```
PRInt64 x, y, two;
LL_I2L(x, 1);
LL_I2L(y, 2);
LL_I2L(two, 2);
LL_MUL(y, x, two);
```

A full listing of NSPR's `long long` support can be found at *http://www.mozilla.org/projects/nspr.*

The **WebLock** component doesn't have to deal with files that are longer than 2^32 bytes. Truncating this value to whatever can fit into a 32 bit unsigned integer may not work for every application, but in this case it doesn't really matter.

Processing the White List Data

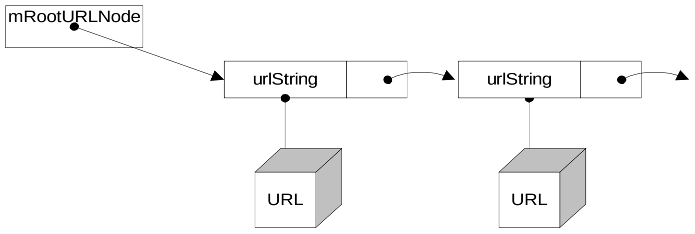
There are various ways to process the file data itself. The file *weblock.txt* consists of URL tokens separated by return characters, which makes them easy to read into a data structure.

The white list file can be read in as soon as the component starts up (i.e., as **WebLock** intercepts the startup notification in the `Observe` method of the `nsIObserver` interface that we implement). Since we have only registered to receive a notification when XPCOM starts up, it's a safe assumption that `Observe` will only called during the startup event, so we can read the file data in the callback.

After you've read the data into memory, you need to store it in some way to make data access quick and efficient.

URL Checking

The way in which URL checking is implemented in the **WebLock** component is not at all optimal. The **WebLock** component manages a simple linked list of URL strings. A linear search through the data in the white list may not be terribly bad if the number of URLs is under a couple of dozen, but it decays as the list grows. There's also a large bottleneck in the network request. URL data is accessed as in the diagram below:



You might construct hash values for each of the URL strings instead, or add them to some kind of database. But we leave optimizations and real-world performance for web locking to the reader.

iWebLock Method by Method

The implementation of the `iWeblock` interface is straightforward. **WebLock** is designed so that the user interface notifies this service when we should go into lock mode. During this time, any new URL request that is not in our list of "good" URLs will be denied. Through scriptable access to the `iWebLock` interface, the user interface can also add, remove, and enumerate the list of URLs that it knows about.

Lock and Unlock

The `lock` and `unlock` methods simply set a Boolean representing state in the object. This Boolean value will be used later to determine if we should be denying URL requests:

```
/* void lock (); */
NS_IMETHODIMP WebLock::Lock()
{
    mLocked = PR_TRUE;
    return NS_OK;
}

/* void unlock (); */
NS_IMETHODIMP WebLock::Unlock()
{
    mLocked = PR_FALSE;
    return NS_OK;
}
```

AddSite

For `AddSite`, we add a new node to our linked list. The link list nodes contain a `char*` which points to the string URL that we care about and, of course, a pointer to the next element in the list.

nsMemory for Cross-component Boundaries

WebLock maintains ownership of all the memory it alloates, so you can use just about any allocator that you want for **WebLock,** but this is not always the case. In other places, where allocated buffers cross-interface boundaries, you must ensure that the correct allocator is used—namely `nsMemory`—so that the allocators can match the allocation with the deallocation.

Suppose you call `malloc` from object A and pass this buffer to another object B, for example. But if object B is using a special allocator that does garbage collection, then when object B deletes a buffer allocated by object A's allocator, the results are unpredictable: probably an assertion will be raised, possibly a memory leak, or a crash. The `nsMemory` class is a wrapper around the `nsIMemory` interface, whose only implementation is part of XPCOM. When you use `nsMemory`, you are guaranteed to be using this same memory allocator in all cases, and this avoids the problem described here.

RemoveSite

`RemoveSite` deletes a node from the linked list:

```
// a simple link list.
struct urlNode
{
    char* urlString;
    struct urlNode* next;
};

/* void addSite (in string url); */
NS_IMETHODIMP WebLock::AddSite(const char *url)
{
    // we don't special-case duplicates here
    urlNode* node = (urlNode*) malloc(sizeof(urlNode));
    node->urlString = strdup(url);
    node->next = mRootURLNode;
    mRootURLNode = node;

    return NS_ERROR_NOT_IMPLEMENTED;
}

/* void removeSite (in string url); */
NS_IMETHODIMP WebLock::RemoveSite(const char *url)
{
    // find our entry.
    urlNode* node = mRootURLNode;
    urlNode* prev = nsnull;

    while (node)  // test this!
    {
        if (strcmp(node->urlString, url) == 0)
        {
            free(node->urlString);
            if (prev)
               prev->next = node->next;
            free(node);
            return NS_OK;
        }
        prev = node;
        node = node->next;
    }

    return NS_ERROR_FAILURE;
}
```

SetSites

The purpose of `SetSites` is to allow clients to pass an enumeration, or set, of URL strings to add to the white list of URLs. `SetSites` uses an `nsISimpleEnumerator` and shows how primitive data can be passed as an `nsISupport` object. The `nsISimpleEnumerator` interface is shown in the section “The Web Locking Interface” on page 105.

The first method returns a Boolean if there are more elements in the set. Internally, the object knows the number of elements it has in its enumeration, and every time a client calls `getNext`, it decrements a counter—or adjusts a pointer to the next element. When the counter goes to zero or the pointer points to a non-element, `hasMoreElements` will return false.

There is no way to reset an `nsISimpleEnumerator`. For example, you can’t re-enumerate the set. If you need random access to the elements in a `nsISimpleEnumerator`, you can read them from the `nsISimpleEnumerator`, store them in an array, and access them there. The `getNext` method returns a `nsISupports` interface pointer.

When you want to pass primitive data type like numbers, strings, a character, `void *`, and others, the solution is to use *nsISupportsPrimitives*, which is a set of interfaces that wraps the primitive data types and derives from `nsISupports`. This allows types like the strings that represent URLs in the **WebLock** component to be passed though methods that take an `nsISupports` interface pointer. This becomes clear when when you see the implementation of `SetSites`:

```
NS_IMETHODIMP WebLock::SetSites(nsISimpleEnumerator * aSites)
{
    PRBool more = PR_TRUE;
    while (more) {
        nsCOMPtr<nsISupports> supports;
        aSites->GetNext(getter_AddRefs(supports));

        nsCOMPtr<nsISupportsCString> supportsString =
           do_QueryInterface(supports);

        if (supportsString) {
            nsEmbedCString url;
            supportsString->GetData(url);
            AddSite(url.get());
        }

        aSites->HasMoreElements(&more);
    }

    return NS_OK;
}
```

GetNext

`GetNext` is called with the `nsCOMPtr` of an `nsISupportsCString`. `nsCOMPtrs` are nice because they do whatever `QueryInterface` calls are necessary under the hood. For example, we know that the `GetNext` method takes an `nsISupports` object, but we may not be sure whether the return result supports the interface we want, `nsISupportsCString`. But after `GetNext` returns, the `nsCOMPtr` code takes the out parameter from `GetNext` and tries to `QueryInterface` it to the `nsCOMPtr`'s type. In this case, if the out parameter of `GetData` does not return something that is `QueryInterface`'able to an `nsISupportsCString`, the variable will be set to null. Once you know that you have an `nsISupportsCString`, you can grab the data from the primitive supports interface.

To get something you can pass into the `AddSite` method, you need to convert from an `nsEmbedCString` to a `const char*`. To do this, you can take advantage of the `nsEmbedCString` described in "String Classes in XPCOM" on page 84.

GetSites

The implementation of `GetSites` is more involved. You must construct an implementation of `nsISimpleEnumerator` and return it when `GetSites` is called. The class needs to walk the list of `urlNode`'s for every call to `GetNext`, so it makes sense for the constructor itself to take an `urlNode`:

```
class myEnumerator : public nsISimpleEnumerator
{
public:
    NS_DECL_ISUPPORTS
    NS_DECL_NSISIMPLEENUMERATOR

    myEnumerator(urlNode* node) { mNode = node; }
    virtual ~myEnumerator(void) {}

protected:
    urlNode* mNode;
    nsCOMPtr<nsIComponentManager> mCompMgr;
};
```

The `myEnumerator` class is going to implement the `nsISupports` interface and also `nsISimpleEnumerator`. The only state that it needs to maintain is the current URL node—the one that will be return on the next call to `GetNext`. There is also an `nsCOMPtr` to the `nsIComponentManager`, which is used in every call to `GetNext` so that you can create `nsISupportsCString` objects and cache the interface pointer as an optimization.

HasMoreElements

`HasMoreElements` is simple. All you need to do is make sure that `mNode` isn't null:

```
NS_IMETHODIMP
myEnumerator::HasMoreElements(PRBool* aResult)
{
    if (!aResult)
        return NS_ERROR_NULL_POINTER;

    if (!mNode) {
        *aResult = PR_FALSE;
        return NS_OK;
    }

    *aResult = PR_TRUE;
    return NS_OK;
}
```

`GetNext` needs to create an `nsISupportsCString` so that you can pass the URL string out through the nsISupports parameter. You must also move `mNode` to point to the next `urlNode`.

```
NS_IMETHODIMP
myEnumerator::GetNext(nsISupports** aResult)
{
    if (! aResult)
        return NS_ERROR_NULL_POINTER;

    *aResult = nsnull;

    if (!mNode)
        return NS_ERROR_FAILURE;

    if (!mCompMgr) {
        NS_GetComponentManager(getter_AddRefs(mCompMgr));
        if (!mCompMgr)
            return NS_ERROR_UNEXPECTED;
    }


    nsISupportsCString* stringSupports;
    mCompMgr->CreateInstance(kSupportsCStringCID,
                             nsnull,
                             NS_GET_IID(nsISupportsCString),
                             (void**)&stringSupports);
    if (!stringSupports)
        return NS_ERROR_UNEXPECTED;


    nsEmbedCString str(mNode->urlString);
    stringSupports->SetData(str);

    *aResult = stringSupports; // addref’ed above.

    mNode = mNode->next;

    return NS_OK;
}
```

In the actual `GetSites` call, all you have to do is create an instance of `myEnumerator` and return it.

Before, we created a class and registered it with the component manager. When a client outside of the code wanted to acquire the implementation of an interface, the actual object creation was hidden in the XPCOM code. Here, however, you instantiate your own implementation of a `nsISimpleEnumerator`. This is a simple thing to do, but it requires that you pay special attention to the `NS_ADDREF`.

```
NS_IMETHODIMP WebLock::GetSites(nsISimpleEnumerator * *aSites)
{
    myEnumerator* enumerator = new myEnumerator(mRootURLNode);
    if(!enumerator) return NS_ERROR_OUT_OF_MEMORY;

    NS_ADDREF(*aSites = enumerator);
    return NS_OK;
}
```

AddRef, Releasing, and Deleting Objects

Never forget to `AddRef` an XPCOM object which you instantiate via `new`. All code that uses or is based on XPCOM requires objects that are alive to have a reference count of at least one. Ignoring this fact can cause real trouble.

A related warning is that you should never delete an XPCOM object with `delete`. It can take hours to find the source of crashes that are caused when one part of a system is deleting XPCOM objects instead of releasing them.

Note that in the implementation above, `myEnumerator` may become invalid if another thread concurrently accesses the linked list. The enumeration represents only one way to walk the linked listed of URL strings. If you require that the enumeration be a snapshot of the list of URL strings, then you have to rework this implementation so that the enumerator owns a copy of the linked list.

At component shutdown, you also need to write the linked list to disk and release the memory occupied by the linked list. We leave these as exercises for the reader.

CHAPTER 7

Tutorial: Finishing the Component

At this point you have created most of the infrastructure of the component. The component will be recognized by XPCOM and registered with the Category Manager so that it starts up when XPCOM initializes. When the component starts up, it populates a list of URLs read in from a file stored next to the gecko binary on the local system.

Using Frozen Interfaces

The core functionality of blocking sites is still missing, however. The interfaces needed to block certain URLs from loading are not frozen, and there is still some debate about how exactly this functionality should be exposed to embedders and component developers, so the APIs are not ready to be published. This puts you in the same situation as many developers using Mozilla—you want to use some specific functionality, but the interfaces seem to change on a daily basis.

All of the Mozilla source code is publicly available, and interfaces can be used easily enough. Grab the right headers, use the Component or Service Manager to access the interface you want, and the XPCOM object(s) that implement that interface will do your bidding. With this huge amount of flexibility, however, you lose compatibility. If you use ‘stuff’ that isn’t frozen, that stuff is subject to change in future versions of Gecko.

If you want to be protected against changes in Gecko, you must only use interfaces and APIs that are clearly marked as FROZEN. The marking is made in the comments above the interface declaration. For example, take a look at the `nsIServiceManager`:

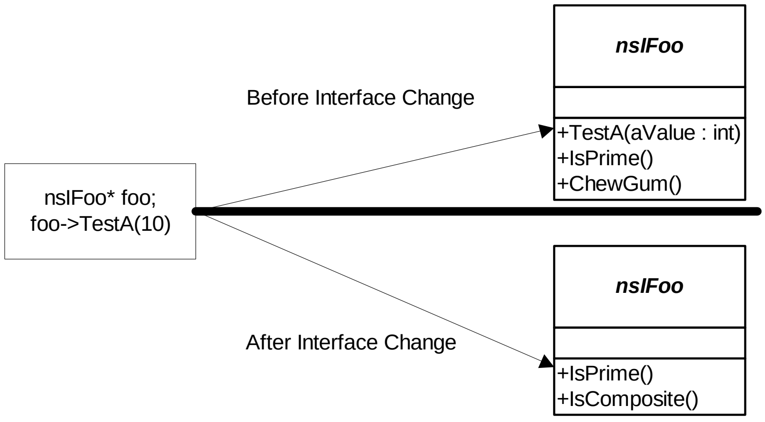
```
/**
 * The nsIServiceManager manager interface provides a means to obtain
 * global services in an application. The service manager depends
 * on the repository to find and instantiate factories to obtain
 * services.
 *
 * Users of the service manager must first obtain a pointer to the
 * global service manager by calling NS_GetServiceManager. After that,
 * they can request specific services by calling GetService.
 * When they are finished they can NS_RELEASE() the service as usual.
 *
 * A user of a service may keep references to particular services
 * indefinitely and only must call Release when it shuts down.
 *
 * @status FROZEN
 */
```

These frozen interfaces and functions are part of the Gecko SDK. The rule of thumb is that interfaces outside of the SDK are considered “experimental” or unfrozen. See the following sidebar for information about how frozen and unfrozen interfaces can affect your component development, and for technical details about how interface changes beneath your code can cause havoc.

The Danger of Using Unfrozen Interfaces

Suppose that you need to use the interface `nsIFoo` that isn't frozen. You build your component using this interface, and it works great with the version of Gecko that you have tested against. However, some point in the future, the `nsIFoo` interface requires a major change, and methods are reordered, some are added, others are removed. Moreover, since this interface was never supposed to be used by clients other than Gecko or Mozilla, the maintainers of the interface don't know that it's being used, and don't change the IID of the interface. When your component runs in a version of Gecko in which this interface is updated, your method calls will be routed through a different v-table than the one the component expected, most likely resulting in a crash.

Below, the component is compiled against a version of the `nsIFoo` interface that has three methods. The component calls the method `TestA` and passes an integer, 10. This works fine in any Gecko installation where a contract guarantees that the interface that was compiled against has the same signature. However, when this same component is used in a Gecko installation where this interface has changed, the method `TestA` does not exist in the `nsIFoo` interface, where the first entry in the v-table `IsPrime()`. When this method call is made, the code execution treats the `IsPrime` method as `TestA`. Needless to say, this is a bad thing. Furthermore, there is no way easy way to realize this error at runtime.



Gecko developers could change the interface's IID, and some do. This can prevent many errors like this. But unfrozen interfaces are not supported in any formal way, and relying upon a different IID for any change in the interface is not a good idea either.

When using frozen interfaces, you are guaranteed compatibility with future versions of Gecko. The only trouble occurs when the compiler itself changes its v-table layout, which can happen when the compiler changes its ABI. For example, in 2002 the GNU Compiler Collection (GCC), version 3.2 changed the C++ ABI, and this caused problems between libraries compiled with GCC 3.2 and applications compiled with an earlier version and vice versa.

Before attempting to use unfrozen interfaces, you should contact the developers who are responsible for the code you're trying to use (i.e., *module owners*) and ask them how best to do what you are trying to do. Be as precise you possibly can. They may be able to suggest a supported alternative, or they may be able to notify you about pending changes. A complete listing of module owners can be found at *http://www.mozilla.org/owners.html*.

The interface that we need for this project is something called `nsIContentPolicy`. This interface is currently *under review*. An interface reaches this state when a group of module owners and peers are actively engaged in discussion about how best to expose it. Usually there are only minor changes to interfaces marked with such a tag. Even with interfaces marked "under review," however, it's still a good idea to contact the module owners responsible for the interfaces you are interested in using.

Copying Interfaces into Your Build Environment

To get and implement interfaces that are not part of Gecko in your component, simply create a new directory in the Gecko SDK named "unfrozen". Copy the headers and IDL files that you need from the *mozilla/content/base/public* source directory of the Gecko build into this new directory. (For **WebLock**, all you need are the headers for nsIContentPolicy and the *nsIContentPolicy.idl.)* Then, using the same steps you used to create the *Weblock.h*, create a header from this IDL file using the xpidl compiler. Once you have these interface and header files, you can modify the `WebLock` class to implement the `nsIContentPolicy` interface. The `Weblock` class will then support four interfaces: `nsISupports`, `nsIObserver`, `nsIContentPolicy`, and `iWeblock`.

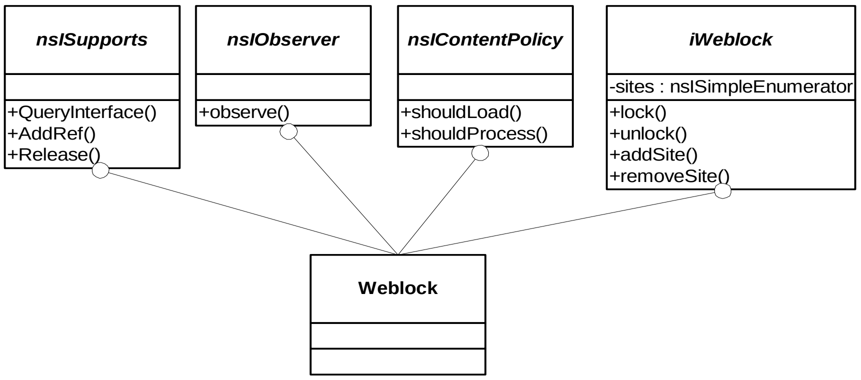


Table 1: WebLock Interfaces

Interface Name	Define by	Status	Summary
`nsISupports`	XPCOM	Frozen	Provides interface discovery, and object reference counting
`nsIObserver`	XPCOM	Frozen	Allows messaging passing between objects
`nsIContentPolicy`	Content	Not Frozen	Interface for policy control mechanism
`iWeblock`	Web Lock	Not Frozen	Enables and disables Weblock. Also, provides access to the URL that are whitelisted.

Implementing the nsIContentPolicy Interface

To implement the new interface, you must `#include` the unfrozen `nsIContentPolicy`, and you must also make sure the build system can find the file you've brought over. The location of the file and the steps for adding that location to the build system vary depending on how you build this component.

Once you have made sure that your component builds with the new header file, you must derive the `Weblock` class from the interface `nsIContentPolicy`, which you can do by simply adding a public declaration when defining the class. At the same time, you can add the macro `NS_DECL_NSICONTENTPOLICY` to the class declaration that provides all of the methods defined in the interface `nsIContentPolicy`. The updated `WebLock` class looks as follows:

```
class WebLock: public nsIObserver,
        public iWeblock,
        public nsIContentPolicy
      {
public:
  WebLock();
  virtual ~WebLock();

  NS_DECL_ISUPPORTS
  NS_DECL_NSIOBSERVER
  NS_DECL_IWEBLOCK
  NS_DECL_NSICONTENTPOLICY

private:
  urlNode* mRootURLNode;
  PRBool   mLocked;
};
```

Remember to change the `nsISupport` implementation macro to include `nsIContentPolicy` so that other parts of Gecko will know **WebLock** supports the `nsIContentPolicy` without modifying this macro.

```
NS_IMPL_ISUPPORTS3(WebLock, nsIObserver, iWeblock, nsIContentPolicy);
```

Receiving Notifications

To receive notifications, you must register as a new category. You have already registered as a category to receive startup notification. This time, the category name to use is “content-policy”. To add the **WebLock** component to this category, modify the `WebLockRegistration` callback function so that it looks like this:

```
static NS_METHOD WebLockRegistration(
  nsIComponentManager *aCompMgr,
  nsIFile *aPath,
  const char *registryLocation,
  const char *componentType,
  const nsModuleComponentInfo *info)
{
    nsresult rv;
    nsCOMPtr<nsIServiceManager> servman =
do_QueryInterface((nsISupports*)aCompMgr, &rv);
    if (NS_FAILED(rv))
        return rv;

    nsCOMPtr<nsICategoryManager> catman;
    servman->GetServiceByContractID(NS_CATEGORYMANAGER_CONTRACTID,
                                    NS_GET_IID(nsICategoryManager),
                                    getter_AddRefs(catman));
    if (NS_FAILED(rv))
        return rv;

    char* previous = nsnull;
    rv = catman->AddCategoryEntry("xpcom-startup",
                                  "WebLock",
                                  WebLock_ContractID,
                                  PR_TRUE,
                                  PR_TRUE,
                                  &previous);
    if (previous)
        nsMemory::Free(previous);

    rv = catman->AddCategoryEntry("content-policy",
                                  "WebLock",
                                  WebLock_ContractID,
                                  PR_TRUE,
                                  PR_TRUE,
                                  &previous);
    if (previous)
        nsMemory::Free(previous);
    return rv;
}
```

This code adds a new category entry under the topic “content-policy,” and it calls `AddCategoryEntry` in the same way we did in “Registering for Notifications” on page 94. A similar step is required for unregistration.

Implementing the nsIContentPolicy

At this point, you can take the **WebLock** component and install it into a Gecko installation. When the component is loaded, Gecko calls the `nsIContentPolicy` implementation in **WebLock** on every page load, and this prevents pages from displaying by returning the proper value when the load method is called.

The web locking policy that we are going to put into place is quite simple: For every load request that comes through, we will ensure that the URI is in the list of "good" URLs on the white list.

> If you care to extend this implementation so that the list of URLs is held remotely on a server somewhere—as might be the case when the **WebLock** component is used in a corporate intranet, for example—there are Networking APIs in Gecko that will support this. Or you could implement the web lock so that instead of blocking any site, the component would simply log all URLs that are loaded. In any case, the process to make an XPCOM component is the same.

The method that handles the check before page loading and the only method we care about in our own implementation of `nsIContentPolicy` is `ShouldLoad()`. The other method on the `nsIContentPolicy` interface is for blocking processing of specific elements in a document, but our policy is more restrictive: if the URL isn't on the white list, the entire page should be blocked. In the **WebLock** component, `ShouldLoad` method looks like this:

```
NS_IMETHODIMP WebLock::ShouldLoad(PRInt32 contentType,
                                  nsIURI *contentLocation,
                                  nsISupports *ctxt,
                                  nsIDOMWindow *window,
                                  PRBool *_retval)
```

Uniform Resource Locators

The method passes in an interface pointer of type `nsIURI`, which is based on the Uniform Resource Identifier, or URI. This type is defined by the World Wide Web Consortium (*http://www.w3.org*) as:

- The naming scheme of the mechanism used to access the resource.

- The name of the machine hosting the resource.
- The name of the resource itself, given as a path.

In this context, URIs are the strings used refer to places or things on the web. This specific form of URI is called a Uniform Resource Locator, or URL. For more information about URIs and URLs, see http://www.w3.org/TR/REC-html40/intro/intro.html

Gecko encapsulates these identifiers into two interfaces, `nsIURI` and the `nsIURL`. You can `QueryInterface` between these two interfaces. The networking library, Necko, deals only with these interfaces when handling requests. When you want to download a file using Necko, for example, all you probably have is a string that represents the URI of the file. But when you pass that string to Necko, it creates an object that implements at least the `nsIURI` interface (and perhaps other interfaces as well).

Currently, the **WebLock** implementation of the `ShouldLoad` method compares the in parameter with each string in the white list. But it only should do this comparison for remote URLs, because we don't want to block the application from loading local content that it requires, like files it gets via the `resource://` protocol. If URIs of this kind are blocked, then Gecko will not be able to start up, so we'll restrict the content policy to the HTTP and FTP protocols.

Instead of extracting the string `spec` out of the `nsIURI` to do a string comparison, which would requre you to do the parsing yourself, you can compare the `nsURI` objects with each other, as in the following section. This ensures that the URLs are canonical before they are compared.

Checking the White List

The **WebLock** implementation of the `ShouldLoad` method starts by extracting the scheme of the incoming `nsIURI`. If the scheme isn't "http", "https", or "ftp", it immediately returns true, which continues the loading process unblocked.

These three are the only kinds of URI that **Weblock** will try to block. When it has one, it walks the linked list and creates a new `nsIURI` object for each string URL in the list. From each object, `ShouldLoad()` extracts the host and compares it to the URI. If they match, the component allows the load to continue by returning true. If these two strings do not match, then the component returns return false and blocks the load.

URI Caching

Caching the URI would make this method implementation much faster by avoiding the need to create and destroy so many objects. This points out an important drawback of XPCOM, which is that you cannot create an object on the stack.

Creating this many objects is OK in a tight loop if the buffer of memory that holds the contents of the URLs is guaranteed to be valid for the lifetime of the object. But regardless of how optimized the implementation is with respect to is memory usage, a heap allocation will be made for every XPCOM object created.

The string comparison with the URL type “http”, “https”, and “ftp” looks like this:

```
nsEmbedCString scheme;
contentLocation->GetScheme(scheme);

if (strcmp("http", scheme.get()) != 0 &&
   strcmp("https", scheme.get()) != 0 &&
   strcmp("ftp",  scheme.get()) != 0 ) {
   // this isn’t a type of URI that we deal with.
   *_retval = PR_TRUE;
   return NS_OK;
}
```

Creating nsIURI Objects

To create an `nsIURI`, use `nsIIOService`. `nsIIOService` is the part of the networking library (“necko”) that’s responsible for kicking off network requests, managing protocols such as http, ftp, or file, and creating `nsIURIs`. Necko offers tremendous network functionality, but all the **WebLock** component needs is to create the `nsIURI` object that can be compared with the URIs on the white list.

Use the Service Manager to acquire the `nsIIOService`. Since this object is going to be used for the life of the component, it can also be cached. A good place to get an `nsIIOService` is in the component’s `Observer()` method, which already has a pointer to the Service Manager. The code for getting the IO service from the Service Manager looks like this:

```
// Get a pointer to the IOService
rv = servMan->GetServiceByContractID(
   "@mozilla.org/network/io-service;1",
   NS_GET_IID(nsIIOService),
   getter_AddRefs(mIOService));
```

Once you have this interface pointer, you can easily create `nsIURI` objects from a string, as in the following snippet:

```
nsCOMPtr<nsIURI> uri;
nsEmbedCString urlString(node->urlString);
mIOService->NewURI(urlString,
  nsnull,  nsnull,
  getter_AddRefs(uri));
```

This code wraps a C-string with a `nsEmbedCString`, which you'll recall is a string class that many of the Gecko APIs require. See "String Classes in XPCOM" on page 84 for more information about strings.

Once the URL string is wrapped in a `nsEmbedCString`, it can be passed to the method `NewURI`. This method expects to parse the incoming string and create an object which implements a `nsIURI` interface. The two `nsnull` parameters passed to `NewURI` are used to specify the charset of the string and any base URI to use, respectively. We are assuming here that the charset of the URL string is UTF8, and also assuming that every URL string is absolute. See *http://www.w3.org/TR/REC-html40/intro/intro.html* for more information about relative URLs.

Here is the complete implementation of the `ShouldLoad()` method:

```
NS_IMETHODIMP WebLock::ShouldLoad(PRInt32 contentType,
                                  nsIURI *contentLocation,
                                  nsISupports *ctxt,
                                  nsIDOMWindow *window,
                                  PRBool *_retval)
{
    if (!contentLocation)
        return NS_ERROR_FAILURE;


   nsEmbedCString scheme;
   contentLocation->GetScheme(scheme);

   if (strcmp("http", scheme.get()) != 0 &&
      strcmp("https", scheme.get()) != 0 &&
      strcmp("ftp",  scheme.get()) != 0 ) {
      // this isn't a type of URI that we deal with.
      *_retval = PR_TRUE;
      return NS_OK;
   }

    nsEmbedCString hostToLoad;
   contentLocation->GetHost(hostToLoad);

   // Assume failure.  Do not allow this nsIURI to load.
   *_retval = PR_FALSE;

   nsresult rv;

    urlNode* node = mRootURLNode;
   PRBool match = PR_FALSE;

   while (node)
    {
      nsCOMPtr<nsIURI> uri;
        nsEmbedCString urlString(node->urlString);
        rv = mIOService->NewURI(urlString, nsnull,  nsnull,
getter_AddRefs(uri));

      // if anything bad happens, just abort.
      if (NS_FAILED(rv))
         return rv;

      nsEmbedCString host;
      uri->GetHost(host);

      if (strcmp(hostToLoad.get(), host.get()) == 0) {
         // match found.  Allow this nsIURI to load.
         *_retval = PR_TRUE;
         return NS_OK;
      }
        node = node->next;
    }
    return NS_OK;
}
```

At this point, all of the backend work is complete. You can of course improve this backend in many ways, but this example presents the basic creation of what is commonly referred to as a “browser helper object” like **WebLock**. The next chapter looks at how to tie this into the front end—specifically how to use XPConnect to access and control this component from Javascript in the user interface.

CHAPTER 8

Tutorial: Building the WebLock UI

Up until now, we've been building a component that can be installed in any Gecko application. The XPCOM interfaces and tools you've used have been general, cross-platform, and available in the Gecko Runtime Environment or in any Gecko-based application after Mozilla 1.2 (when the GRE began to be used).

In this chapter, however, we are going to be building a user interface for the **WebLock** component that's meant to be added to the *existing* Mozilla browser[1]. It uses XUL, which is an XML language that Gecko knows how to render as user interface, but it also interacts with particular parts of the Mozilla user interface, where it must install itself as an extension to the UI. Specifically, the user interface we create in this chapter will be *overlaid* into the statusbar of the browser component, where it will provide a small icon the user can click to access the web lock interface (see *Figure 1*).

1. Or one very much like it. There are Gecko-based browsers such as Beonex and the IBM Web Browser that share a lot of the structure of the Mozilla user interface, into which it may be possible to install both the **WebLock** component and the user interface described in this chapter.

Figure 1. WebLock Indicator in Browser

User Interface Package List

The user interface described in this section is comprised of four files:

- *webLockOverlay.xul* is the file that defines the little status icon in the browser.
- *weblock.xul* defines the web lock manager dialog.
- *weblock.css* provides style rules for both of the XUL files.
- *weblock.js* provides JavaScript functions for both of the XUL files.

Each of these files is described briefly in the sections below. In the following chapter we'll describe how you can take these files and create a *package*, an installable archive that includes the WebLock component and the new UI.

Because this step (particularly the overlay section) is so dependent on Mozilla, the chapter is divided up into a couple of different sections. The second section, "XUL", describes the XML-based User Interface Language (XUL) and how you can use it to create a dialog that provides access to the **WebLock** component and its services. The third section, "Overlaying New User Interface Into Mozilla", describes how to create an overlay into the browser that will make this dialog accessible from a Mozilla build. The overlay section, where we discuss how to import scripts, images, and other resources from Mozilla into your own UI, is by far the more complicated of the two.

If the **WebLock** component is being installed in Mozilla or another Gecko-based browser, then this third section shows you how to create the entry point in the browser for controlling the web locking. If you are planning on deploying the **WebLock** component in some other application, you'll have to devise a different means of access (e.g., native widgetry that instantiates and controls the **WebLock** component).

- "Client Code Overview"

- “XUL”
- “Overlaying New User Interface Into Mozilla”

Client Code Overview

Before we get started on the actual user interface, we should establish how the client code is to access the **WebLock** component and use its interfaces to control the web locking of the browser.

First of all, it’s important to be able to represent the basic state of the lock as soon as it’s loaded. Like the secure page icon, the weblock icon that appears in the lower right corner of the browser should indicate whether the browser is currently locked or unlocked. Since the **WebLock** component is always initialized as unlocked, we can have the client code—the JavaScript code in the interface—represent this state and track it as the user manipulates the `iWebLock` interface. A boolean `wLocked` variable can do this:

```
// initialize the wLocked variable as unlocked
var wLocked = 0;
```

Then the functions that get called from the interface and call through to the `lock` and `unlock` methods of the **WebLock** component must also adjust this variable accordingly:

```
function wLock() {
   // check to see if locking is on or off
   weblock.lock()
   wLocked = 1;
}

function wUnLock() {
   // check to see if locking is on or off
   weblock.unlock()
   wLocked = 0;
}
```

An important preliminary of these functions is that the **WebLock** component be made available to the JavaScript in the form of the `weblock` object being used in the snippets above. As you can see, `weblock` is initialized as a global JavaScript variable, available in the scope of these functions and others:

```
var weblock = Components.classes[
      "@dougt/weblock"].getService();
weblock = weblock.QueryInterface
     (Components.interfaces.iWebLock);
```

In addition to this basic setup, you must also write JavaScript that uses the `AddSite` method to add new sites to the white list. This is a bit more complicated, because it requires that you work with the currently loaded page or provide other UI (e.g., a textfield where you can enter an arbitrary URL) for specifying URLs. In the "XUL" section below, we'll go into how the user interface is defined. This section describes the functions that are called from the interface and how they interact with the **WebLock** component.

The URL that the `AddSite` method expects is a string, so we can pass a string directly in from the user interface, or we can do a check on the string and verify that it's a valid URL. In this tutorial, focusing as it is on the WebLock functionality (rather than the UI), we'll assume the strings we get from the UI itself are URLs we actually want to write to the white list:

```
function addThisSite() {
   tf = document.getElementById(“dialog.input”);
   // weblock is global and declared above
   weblock.AddSite(tf.value);
}
```

This JavaScript function can be called directly from the XUL widget, where the input string is retrieved as the `value` attribute of the `textbox` element.

You also need to create the function that displays the **WebLock** window itself when the user clicks the weblock icon in the statusbar. That function uses the `openDialog` method from the `window` object and takes the URL to the XUL file in which the dialog is defined:

```
function loadWebLock() {
   openDialog(“chrome://weblock/weblock.xul”);
}
```

XUL

The entire user interface of the Mozilla browser and all of the applications that go with it, including the mail client, the IRC client, and others, have been defined in an XML language called XUL. Elements in the XUL markup map to widgets in the interface that Gecko renders in a fairly straightforward way—so, for instance, the root element of an application window is the element `<window>`, the root element of the dialog we'll be creating here is `<dialog>`, and so forth. Within a XUL application file, elements like button, menu, checkbox can be hooked up to an event model, to scripts, and to the XPCOM interfaces that carry out a lot of the browser functionality in Mozilla.

In the chapter *Using Components*, you saw how XPCOM objects are reflected into the interface layer as JavaScript objects. In this chapter, now that we've created the **WebLock** component and made it available to XPCOM, we create the UI that actually instantiates the **WebLock** component and uses its methods to control page loading in the browser.

In the previous section, we outlined the JavaScript that interacts with the WebLock component. In this section, we are going to create the XUL interface that calls the JavaScript methods when the user interacts with it.

The XUL Document

The first thing to do is create the actual XUL document in which the user interface for the dialog and the events that initiate interaction with the web locking are defined. At the top of all XUL documents, an XML declaration is followed by the root element for the document, which is usually `<window>` but for dialog boxes can also be the element `<dialog>`. The "shell" for the XUL file, then, looks like this:

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<dialog id="weblock_ui"
   xmlns="http://www.mozilla.org/keymaster/gatekeeper/
      there.is.only.xul"
   title="Web Lock Manager"
   persist="screenX screenY"
   screenX="24" screenY="24">

</dialog>
```

Note that this part of the XUL file also contains a stylesheet declaration, which imports CSS rules and applies them to particular parts of the interface. In Gecko, CSS is used to do virtually all of the presentation of the XUL interface—its color, position, style, and to some extent its behavior as well. The web lock manager dialog does not deviate from the look of a standard dialog, so it can use a single declaration to import the “global” skin from the browser and make it available to the widgets you define in weblock.xul.

You can save this first, outermost part of the web lock dialog in a file called *weblock.xul*, which you’ll be adding to an installer archive in Appendix B in this book.

> Note that this file defines the dialog that displays when the user/administrator clicks the web locking icon in the bottom right corner of the browser. That piece of UI—which needs to be dynamically inserted into the browser at run-time—is described in the following section, “Overlaying New User Interface Into Mozilla”.

The finished dialog appears in *Figure 2* below.

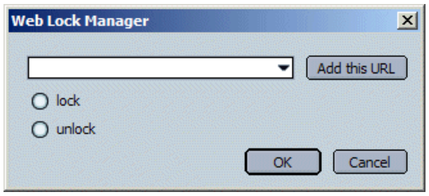


Figure 2. Web Lock Manager Dialog

As you can see, it’s a simple interface, providing just enough widgetry to lock and unlock the browser, and to add new sites to the list. The entire XUL file for the web lock manager dialog is defined in the “weblock.xul” subsection below.

The Locking UI

Once you have the basic XUL wrapper set up for your interface, the next step is to define that part of the interface that locks and unlocks the browser. One of the most efficient ways to expose this is to use radio buttons, which allow the user to toggle a particulart state, as the figure above illustrates.

In XUL, individual radio elements are contained within a parent element called radiogroup. Grouping radio elements together creates the toggling UI by requiring that one or another of the elements be selected, but not both.

The XUL that defines the radiogroup in the web lock manager dialog is this:

```
<radiogroup>
   <radio label="lock" />
   <radio label="unlock" selected=”true” />
</radiogroup>
```

Since the **WebLock** component always starts up in the unlocked position, you can set the default `selected=”true”` property on the unlock radio button and reset it dynamically as the user takes action.

Site Adding UI

The next step is to create that part of the user interface that lets you add a new site to the white list. There are other, more sophisticated ways to do this; you may also want to include some UI that lets you view the whitelist or edit it as a list. In this part of the tutorial, however, we only provide the means of adding an URL provided as a string (and not checked for validity) and passing it through to the AddSite API we defined in the earlier part of the tutorial.

```
<separator class="thin"/>

<hbox align="center">
  <textbox id="url.input" flex="1"/>
  <button label="Add this URL" oncommand="addThisSite();" />
</hbox>
```

This snippet introduces a couple of new general layout widgets in XUL. The separator that appears at the top of this snippet creates a little divider between widgets like the kind you see in menus that divide sets of functionality available there. The parent of the textbox that users enter an URL into is something called an `<hbox>`, which is a layout widget—often invisible—that controls the way its child elements are rendered. In this case, the hbox centers the textbox and the button children, and it orients them horizontally (in contrast to the vbox, which orients its children veritically).

Notice also that when it's clicked, the button executes a JavaScript function called `addThisSite()`, which we've already defined in the *weblock.js* file in the "Client Code Overview" section above.

weblock.xul

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<dialog id="weblock_mgg"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/
there.is.only.xul"
   title="Web Lock Manager"
        style="width: 30em;"
        persist="screenX screenY"
        screenX="24" screenY="24">

<script src="chrome://weblock/content/weblock.js" />

  <hbox>
    <separator orient="vertical" class="thin"/>
    <vbox flex="1">
      <separator class="thin"/>

      <hbox align="center">
        <textbox id="dialog.input" flex="1" />
        <button label="Add this URL"
           oncommand=”addThisSite();”/>
      </hbox>
      <hbox align="center">
        <radiogroup onchange=”toggleLock();”>
           <radio label="lock" />
           <radio label="unlock" />
        </radiogroup>

        <spacer flex="1"/>
      </hbox>
    </vbox>
  </hbox>

 </dialog>
```

Overlaying New User Interface Into Mozilla

You’ve got a dialog that will interact with the **WebLock** component, but how do you install that dialog you’ve created into the browser? And how do you access it once it’s in? Once it’s installed and registered, the **WebLock** component itself is ready to go: XPCOM finds it and adds it to the list of registered components, and then **WebLock** observes the XPCOM start up event and initializes itself.

But you still have to insinuate your new UI into the browser so it can call the component, and the Mozilla *overlay* mechanism is the way to do this. Overlays are XUL files that register themselves to be dynamically inserted into the appropriate parts of the browser UI at runtime.

webLockOverlay.xul

The XUL that defines the new icon is small: it’s a little icon that has an image associated with it, and that calls a JavaScript function to loads the *weblock.xul* file we defined in the previous section. The icon is actually a separate `<statusbar>` element that gets overlaid into the main browser, along with some JavaScript and some CSS to control the behavior and appearance of the element, respectively. Here is that XUL file in its entirety:

```
<?xml version="1.0"?>
<?xml-stylesheet
   href="chrome://navigator/content/weblock.css"
   type="text/css"?>

<overlay id="weblockOverlay"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/
     there.is.only.xul">

  <script type="application/x-javascript"
     src="chrome://weblock/content/weblock.js" />

  <statusbar id="status-bar">
     <statusbarpanel class="statusbarpanel-iconic"
       id="weblock-status"
       insertbefore="offline-status"
       oncommand=”loadWebLock();”
       status="none"/>
  </statusbar>

</overlay>
```

Figure 3. The WebLock Overlay

Note that the root element for this file is not a window but an overlay. In overlays, the ordinarily unique ID attributes that XUL elements use to distinguish themselves are purposely made redundant with UI in the existing browser with which they should be merged. In this case, the weblock `statusbarpanel` appears as a child of the `statusbar` element with ID “status-bar”. This ID is the same one used by the `statusbar` in *navigator.xul*, which means that the overlay mechanism will merge the new UI here (i.e., the weblock `statusbarpanel`) and the UI already defined within that browser `statusbar` at runtime.

Other Resources

This chapter describes the remaining files that must be added to and packaged up with the WebLock component to provide user interface for web locking.

- “weblock.css”
- “Image Resources”

Other Front End Resources

In some UI packages, localization resources are also defined. These include DTDs in which the language in which the UI is labelled can be extracted into external files, which are swapped with DTDs for other languages. For example, user interface packages often include an English DTD that defines labels and strings for button and menus and other elements in the interface. When the user selects a different *language pack*, all of the English that's been externalized in these files is dynamically replaced with the new choice.

In addition to DTDs, the localization parts of a user interface may also include string bundles in which strings that are used in the interface JavaScript can be similarly replaced.

There are also technologies, not discussed here, which may be used in separate, installable files. These include *bindings* in XML files, *metadata* in RDF files, whole collections of CSS files called *skins*, and others.

weblock.css

The following style rules are defined in *weblock.css*, a CSS file that is loaded by the overlay and applied to the icon in the browser that reflects the current status of the web lock and provides access to the web lock manager dialog.

```
statusbarpanel#weblock-status {
   list-style-image: url("chrome://weblock/wlock.gif");
}

statusbarpanel#weblock-status[status="locked"] {
   list-style-image: url("chrome://weblock/wl-lock.gif");
}

statusbarpanel#weblock-status[status="unlocked"] {
   list-style-image: url("chrome://weblock/wl-un.gif");
}
```

The style rules are distinguished by the state of the `status` attribute on the element in the XUL with the ID "weblock-status." As you can see above, when the status of the element is set to "locked", the image *wl-lock.gif* is used to show the state, and when the web locking is unlocked, it uses *wl-un.gif*. (Note: We include three images to represent the state of the weblock, but *wlock.gif* and *wl-lock.gif* are identical, since weblock is presumed to be unlocked when it's loaded. This tutorial makes use of only two different states, but you can further customize the look of the weblock using the three images if you wish.)

Since the presentation of the weblock manager dialog itself doesn't require any special styles, these are all the rules you need in the *weblock.css*. Note that the weblock.xul file in which the manager is defined imports only the global skin:

```
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
```

Save *weblock.css* in your working directory.

You should now have the four files listed at the top of this chapter as the "packing list" for the **WebLock** package (see "User Interface Package List"). Don't worry for now about where these files are. In the next chapter, "Tutorial: Packaging WebLock", we'll describe how to organize and package them so that they can be installed along with the **WebLock** component itself and the other resources.

Image Resources

If you are following along with this tutorial and want to use the images we use here for the states of the **WebLock** component in the statusbar, you can download them and the other resources for **WebLock** from *http://www.brownhen.com/weblock*. The GIF files that represent the various states are:

- *wlock.gif*
- *wl-lock.gif*
- *wl-un.gif*

CHAPTER 9

Tutorial: Packaging WebLock

In this final part of the tutorial, we'll put all of the pieces of the web locking component—the library itself, the type library, the header file, and the user interface resources—into a package that can be installed on other systems. The first section, "Component Installation Overview", describes the general installation process in Mozilla. The following sections describe the steps you can take to organize the **WebLock** component for distribution and installation.

> Note: the emphasis of this tutorial is on the component development itself, so this section on packaging and installing extensions to Gecko is necessarily brief. For more detailed information on packaging and installation of components into Gecko-based applications, see *http://www.mozilla.org/projects/xpinstall*.

- "Component Installation Overview"
- "Archiving Resources"
- "The WebLock Installation Script"
- "The WebLock Trigger Script"

Component Installation Overview

XPInstall is a set of JavaScript APIs for creating installation scripts. Using XPInstall, you can create web-based installations for Gecko-based applications, Mozilla extensions, or individual components. The installation script for the **WebLock** component can also be used to register the component with the browser into which it is installed (see “Registration Methods in XPCOM” for more information on registration).

The sample installation script shown below uses the Mozilla XPInstall technology to manipulate an installer and talk to Mozilla’s *chrome registry* as high-level JavaScript objects.

What Is the Chrome Registry?

Like the Windows registry, the chrome registry is a database of information about applications, skins, and other extensions that have been installed in a Gecko application. Since Mozilla and other Gecko-based applications are cross-platform, this database is abstracted above the operating system or any particular platform’s registry.

The chrome registry lives in a series of RDF/XML files in the application directory of Mozilla and other Gecko-based browsers, where new installs, user configurable data, skins, and other information are related to one another and the application itself.

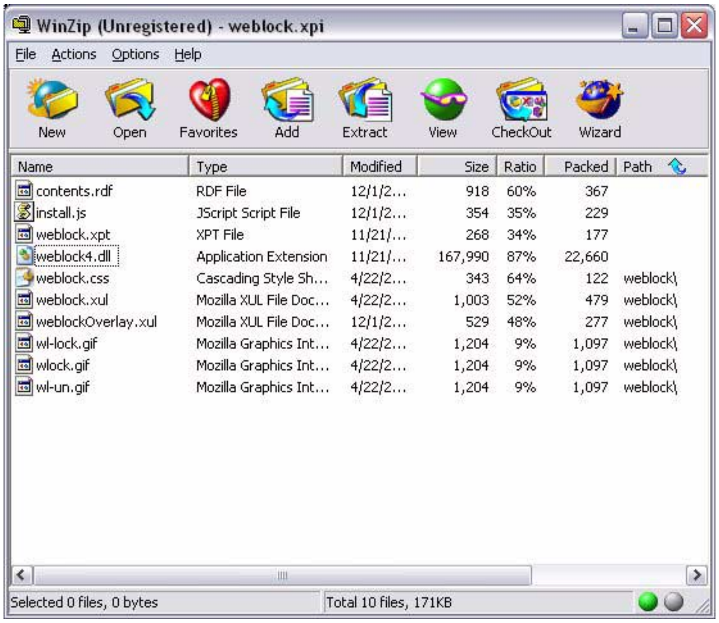
JavaScript APIs from the XPInstall `Install` object download the JAR in which the installable files appear and call registration methods that tell Mozilla about the new component and the UI it uses to access the **WebLock** component. Figure 8 is the complete *trigger installation* script, which can be launched from a web page. The files themselves are stored in the JAR file *weblock.jar,* which is simple a ZIP file with the XPI extension that sometimes also contains an internal installation file called *install.js*.

Once you have the component and the other resources for **Weblock** packaged properly (see the following section, “Archiving Resources”), the installation script for **WebLock** is a simple one (see “The WebLock Installation Script”).

Archiving Resources

Once you have compiled all the resources that make up the **WebLock** component and the files that make up the user interface that will be added to the browser, you can place these within a subdirectory called *weblock*.

Place the entire subdirectory into a ZIP archive and name the archive *weblock.xpi*. The archive, its subdirectory structure, and its contents should look like this:



Name	Type	Modified	Size	Ratio	Packed	Path
contents.rdf	RDF File	12/1/2...	918	60%	367	
install.js	JScript Script File	12/1/2...	354	35%	229	
weblock.xpt	XPT File	11/21/...	268	34%	177	
weblock4.dll	Application Extension	11/21/...	167,990	87%	22,660	
weblock.css	Cascading Style Sh...	4/22/2...	343	64%	122	weblock\
weblock.xul	Mozilla XUL File Doc...	4/22/2...	1,003	52%	479	weblock\
weblockOverlay.xul	Mozilla XUL File Doc...	12/1/2...	529	48%	277	weblock\
wl-lock.gif	Mozilla Graphics Int...	4/22/2...	1,204	9%	1,097	weblock\
wlock.gif	Mozilla Graphics Int...	4/22/2...	1,204	9%	1,097	weblock\
wl-un.gif	Mozilla Graphics Int...	4/22/2...	1,204	9%	1,097	weblock\

Selected 0 files, 0 bytes   Total 10 files, 171KB

Figure 1. weblock.xpi Archive Viewed in WinZIP

Note that the top level of the archive holds the *install.js* installation file, an RDF manifest for the package as a whole, and the component files (*weblock.xpt* and *weblock4.dll*). The component files are copied to the components directory of the Gecko application, and the weblock subdirectory gets copied over into the chrome subdirectory, where its UI resources can be added dynamically to the XUL-based Gecko application.

The next section shows how this process of downloading, copying and registering the necessary files from the XPI can be achieved with an XPInstall installation script.

The WebLock Installation Script

The installation script is the JavaScript file that is stored within the XPI. It must appear at the top level of the installation archive (i.e., *weblock.xpi*) itself. Once triggered (see the next section, “The WebLock Trigger Script”), the installation script:

- downloads the **WebLock** component and places it in the *components* directory
- copies the *weblock* subdirectory in the Mozilla *chrome* application subdirectory
- registers both the component and the UI

The XPInstall API provides such essential methods[1] as `initInstall`, `registerChrome`, `addFile`, and others.

1. The methods are available on the main `Install` object, which is implied in the script below in the same way that the `window` object is implied in JavaScript manipulation of the DOM of a web page. In other words, the fragment `var initInstall()` from the script is equivalent to `var Install.initInstall()`.

```
// initialize the installation
var err = initInstall("WebLock", "weblock", 1.0);

var componentsDir = getFolder("Components");
var cf = getFolder(“Chrome”);

// add the DLL and say where it’ll go
addFile ("weblock.dll", 1.0,
         "weblock.dll", componentsDir, "");

// add the typelib also
addFile ("weblock.xpt", “1.0",
         "weblock.xpt", componentsDir, "");

// add the weblock subdirectory
// of the XPI and specify that it be
// installed in the chrome application directory
err = addDirectory (“weblock”, “1.0”,
         “”, chromeDir, ““)

// ? have to register component here or with regxpcom?

// register the new UI with the mozilla chrome registry

registerChrome(CONTENT, getFolder(cf,"weblock.xpi"),"weblock");
registerChrome(SKIN, getFolder(cf, “weblock.xpi”),”weblock”);


// perform the installation if there are no errors
if (err==SUCCESS)
    performInstall();
else
    cancelInstall(err);
```

Figure 2. WebLock Installation Script

The WebLock Trigger Script

The *trigger script* is the script placed on a web page that actually initiates an XPInstall installation and calls the installation script that appears in the XPI. *Figure 3* is a complete webpage in which the trigger script is defined as a JavaScript function, `installWebLock`, that gets called when the user clicks the hyperlink.

```
<html>
<title>WebLock Installation</title>
<script>
// trigger function
// that downloads the XPI
// so the install.js file inside can be
// read and executed
function installWebLock() {
  weblock_xpi = {‘WebLock Extension’: ‘weblock.xpi’};
  InstallTrigger.install(weblock_xpi);
</script>

<h1>Install WebLock</h1>

<p><a href=”javascript:void”
  onclick=”installWebLock();”>install weblock</a>

</html>
```

Figure 3. Trigger Script for WebLock

Distributing Your Component

Once you have the component packaged properly and the necessary installation and trigger scripts, you are ready to distribute your component so others can install it in their Gecko applications.

In Mozilla and Netscape browsers, XPInstall makes this process especially easy by providing the file format (XPI) and the necessary installation scripts for doing a web-based installation. As *Figure 2* demonstrates, XPInstall uses special keywords to refer to common installation directories such as *components* in a generalized, cross-platform way.

If you are installing **WebLock** in an Gecko-based application for which XPInstall is not available, then you will have to devise a separate installation scheme. We leave this as an exercise for the reader.

Appendix A: Setting up the Gecko SDK

This chapter provides basic setup information for the Gecko Software Development Kit (SDK) used to build the **WebLock** component in this tutorial. The following four sections tell the developer how to download and organize the Gecko SDK and create a new project in which components like **WebLock** can be created:

- “Downloading and Setting the SDK”
- “Building a Microsoft Visual C++ Project”
- “A Makefile for Unix”

Downloading and Setting the SDK

The Gecko SDK provides all of the tools, headers, and libraries that you need to build XPCOM Components. The SDK is available for Windows and Linux operating systems, and versions for other operating systems are being developed, and can be retrieved from as a single archive from the following platform-specific locations:

- Linux: *http://ftp.mozilla.org/pub/mozilla/releases/mozilla1.4a/gecko-sdk-i686-pc-linux-gnu-1.4a.tar.gz*

- Windows: *http://ftp.mozilla.org/pub/mozilla/releases/mozilla1.4a/gecko-sdk-win32-1.4a.zip*

Note that the version number for the archives above is 1.4a. The WebLock component was built with this version, but you can always check for newer versions at *http://ftp.mozilla.org/pub/mozilla/releases/*.

Once you download the SDK, you can expand it into any convenient location. In this appendix, we set up the Windows Gecko SDK in *c:\gecko-sdk*. If you choose some other location, remember to adjust the settings described here (e.g., in the “Building a Microsoft Visual C++ Project” section below) to point to this new location.

When you extract the SDK, it should have the layout seen in *Figure 1*.

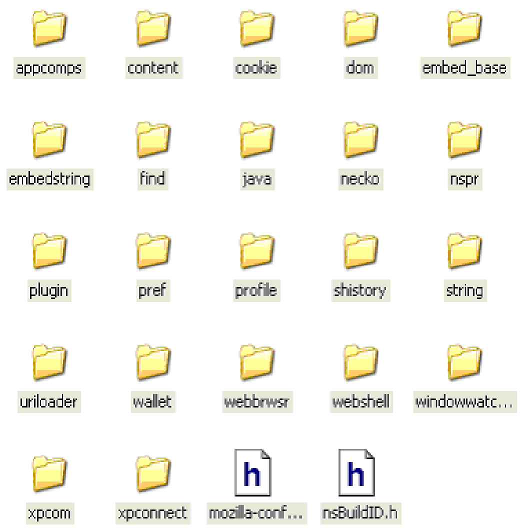


Figure 1. Layout of the Extracted SDK

The directories represent different modules in the SDK. For example, the headers for networking are all located in the "necko" directory, and the headers that XPCOM requires are in the XPCOM directory. This directory structure makes build scripts slightly more complicated (since there will be many different include paths), but it helps to organize the parts of the SDK meaningfully.

The two top level header files are special. The file *mozilla-config.h* lists all of the defines used in the SDK. Including this header file in your project ensures that the component you create uses the same defines as the Gecko libraries themselves. Note that *mozilla-config.h* may be need to be included before other includes in your component's source code.

Each module directory is divided into three subdirectories:

Figure 2. Module Subdirectories

The *bin* directory contains static libraries, dynamic libraries, and in some cases tools that may be useful in development. The *idl* directory contains the public IDL files exported by the module. The *includes* directory contains C++ header files used by your component.

XPCOM exports a number of binaries that should be mentioned at this point. The table below refers to the Windows file names for the executables.

Application Name	**Description of functionality**
regxpcom.exe	Registers or Unregisters components with XPCOM
xpidl.exe	Generates typelib and C++ headers from XPIDL
xpt_dump.exe	Prints out information about a given typelib
xpt_link.exe	Combines multiple typelibs into a single typelib

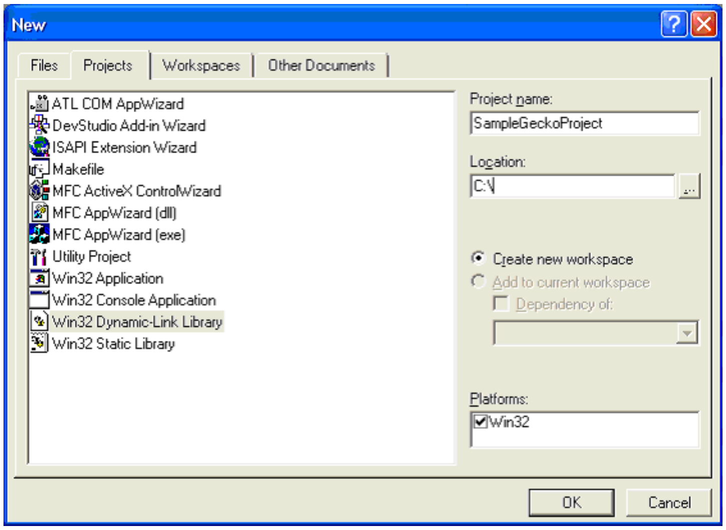
Library Name	**Description of functionality**
xpcomglue.lib	XPCOM Glue library to be used by xpcom components.

Building a Microsoft Visual C++ Project

Once you setup the Gecko SDK, you can create a Microsoft Visual C++ project to handle component development with the SDK.

Creating a New Project

After launching Visual C++, select New from the File menu. Then, from the New dialog, select "Win32 Dynamic-Link Library. Use the fields to the right of the dialog to name your project and set its location (This example uses "SampleGeckoProject" as the Project name and *C:\* as its location.).



Project name: SampleGeckoProject

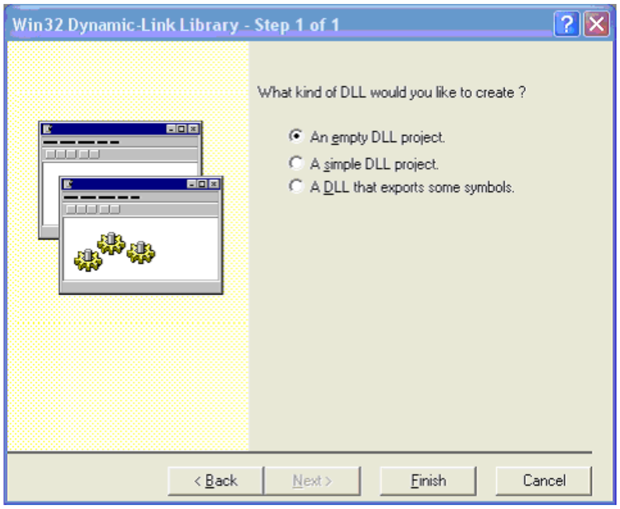
Location: C:\

[✓ Create new workspace]

Platforms: [✓ Win32]

Figure 3. New Dialog

Select OK. In the Win32 Dynamic-Link Library dialog that displays (see *Figure 4*), you can choose the default selection "An Empty DLL Project" as the type of DLL.



What kind of DLL would you like to create ?

- [✓ An empty DLL project.]
- A simple DLL project.
- A DLL that exports some symbols.

Figure 4. Win32 Dynamic-Link Library Dialog

In this dialog, click Finish. Microsoft Studio creates a new project per your specification and presents you with the standard Project view.

Adding the Gecko SDK to the Project Settings

In order to build anything that uses Gecko, you have to further modify the project so that it knows where to find the Gecko SDK on the disk. To edit project settings, select Settings from the Project menu (or press Alt-F7).

Most of the changes you make in the following steps apply to all configurations of the project (both Debug and Optimized), so select “All Configurations” from the Settings For dropdown menu (see *Figure 5*).

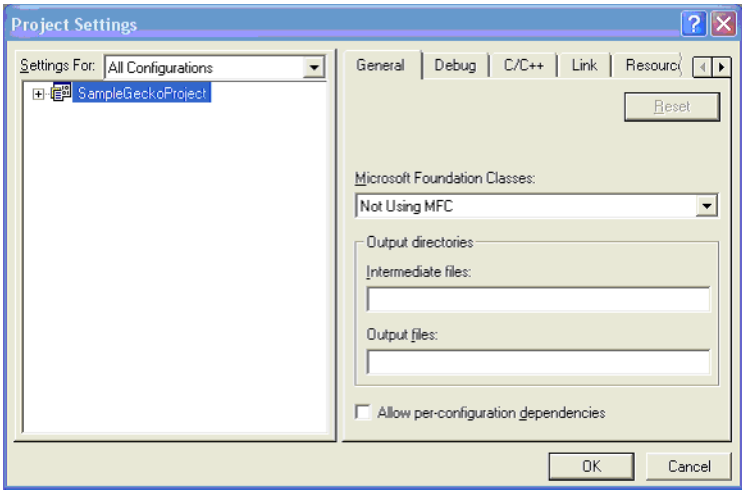


Figure 5. Project Settings Dialog

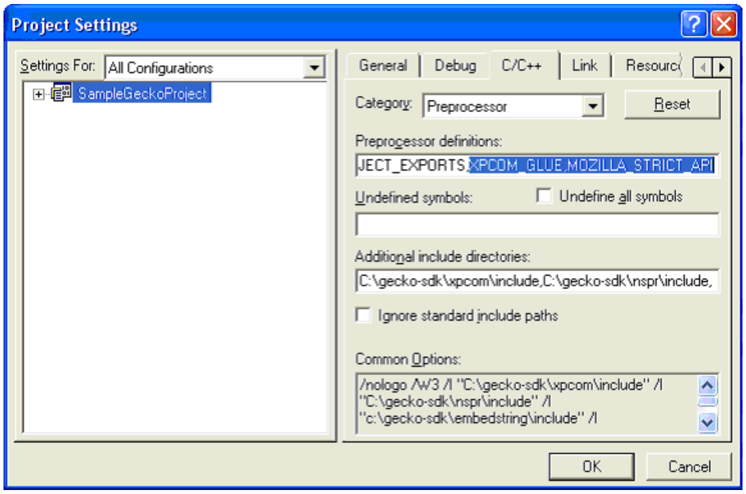
On the C/C++ tab, select the Preprocessor category. This window is where you add the include paths to the Gecko SDK as well as two preprocessor defines:

- `XPCOM_GLUE`
- `MOZILLA_STRICT_API`

At a minimum, you must include the *nspr*, the *embedstring* and *string* include directories, and to the *xpcom* include subdirectories. If your component will use other parts of the SDK (e.g., *necko*), you will have to add these include directories to this field as well.

These paths are the following:

- *c:\gecko-sdk\embedstring\include*
- *c:\gecko-sdk\xpcom\include*
- *c:\gecko-sdk\nspr\include*
- *c:\gecko-sdk\string\include*



Settings For: All Configurations

SampleGeckoProject

Category: Preprocessor

Preprocessor definitions: JECT_EXPORTS,XPCOM_GLUE,MOZILLA_STRICT_API

Undefined symbols:

Undefine all symbols [ ]

Additional include directories: C:\gecko-sdk\xpcom\include,C:\gecko-sdk\nspr\include,

Ignore standard include paths [ ]

Common Options:

```
/nologo /W3 /I "C:\gecko-sdk\xpcom\include" /I
"C:\gecko-sdk\nspr\include" /I
"c:\gecko-sdk\embedstring\include" /I
```

Figure 6. Adding Includes to the Project

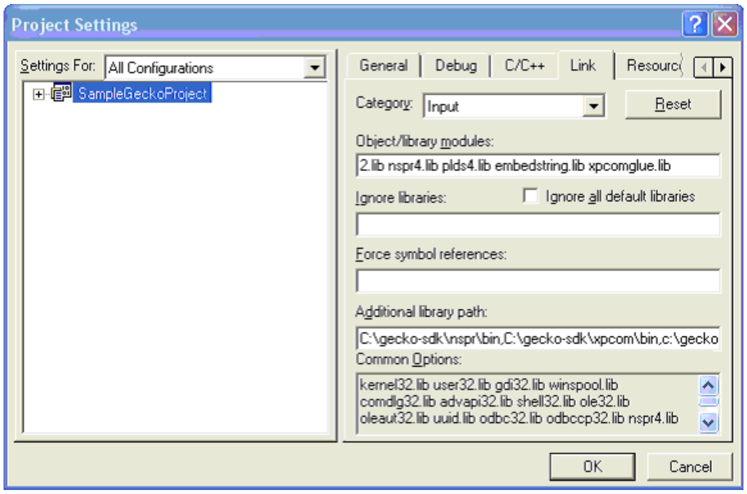
Under the C++ language category, disable exception handling. As described in the section ““Exceptions” in XPCOM” on page 17, exception handling isn't supported across interface boundaries, so setting this option may catch problems during development.

The **WebLock** component needs to link against the appropriate libaries to uses XPCOM Glue. To add these libraries, select the Link tab, then choose the Input category. In this panel, instead of linking to the *include* subdirectories to the *nspr*, *embedstring*, and *xpcom* directories, add the paths to the *bin* subdirectories.

We also link against a number of libraries in the Object/library modules line:

- *nspr4.lib*
- *plds4.lib*
- *plc4.lib*
- *embedstring.lib*
- *xpcomglue.lib*

Both of these settings are shown in *Figure 7*.



Project Settings

Settings For: All Configurations

SampleGeckoProject

General | Debug | C/C++ | Link | Resourc

Category: Input

Object/library modules:
2.lib nspr4.lib plds4.lib embedstring.lib xpcomglue.lib

Ignore libraries:

☐ Ignore all default libraries

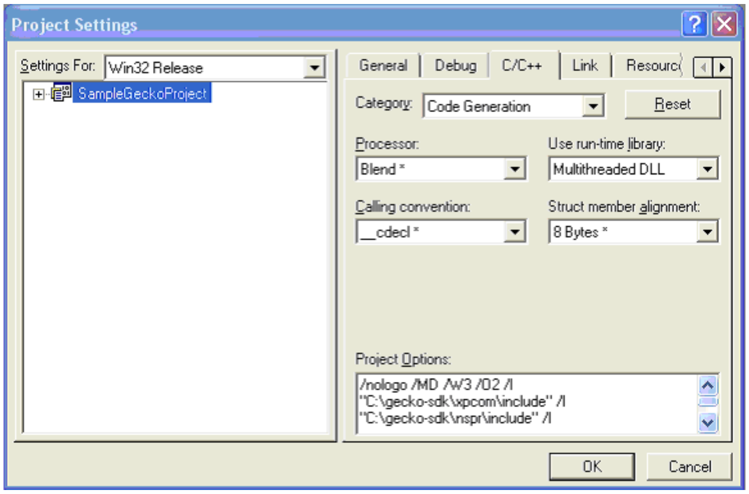
Force symbol references:

Additional library path:
C:\gecko-sdk\nspr\bin,C:\gecko-sdk\xpcom\bin,c:\gecko

Common Options:
kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib ole32.lib
oleaut32.lib uuid.lib odbc32.lib odbccp32.lib nspr4.lib

Figure 7. Bin and Library Settings

The last change you need to make to set up the Gecko SDK in your project is to change the “Use run-time library” setting to “Multithreaded DLL.” Since this change is configuration dependent, you must make set the Release configuration run-time library to the release multithreaded dll runtime and the Debug configuration to the debug multithreaded dll runtime (see *Figure 8*).



Settings For: Win32 Release

SampleGeckoProject

Category: Code Generation

Processor: Blend *

Use run-time library: Multithreaded DLL

Calling convention: __cdecl *

Struct member alignment: 8 Bytes *

Project Options:

```
/nologo /MD /W3 /O2 /I
"C:\gecko-sdk\xpcom\include" /I
"C:\gecko-sdk\nspr\include" /I
```

Figure 8. Run-time Library Settings

After making these changes, press OK. This finalizes the project settings and gives you a project that will hold and compile XPCOM Components.

A Makefile for Unix

On Linux, the equivalent project settings are typically handled in a Makefile. The Makefile allows you to specify any number of options for your build environment, including the path and configuration updates you need to build with the Gecko SDK.

Figure 9 is a listing for a Makefile that configures your compiler to work with the SDK. Explaining the details of the Makefile is outside the scope of this appendix, but it modifies the same properties that are configured in the Visual C++ project (see “Building a Microsoft Visual C++ Project”). For a listing of the commands that appear in this listing, see the Make manual: *http://www.gnu.org/manual/make/*.

```
CXX   = c++

CPPFLAGS +=     -fno-rtti              \
                -fno-exceptions        \
                -shared

# Change this to point at your Gecko SDK directory.
GECKO_SDK_PATH = /home/dougt/gecko-sdk

# GCC only define which allows us to not have to #include mozilla-
config
# in every .cpp file.  If your not using GCC remove this line and
add
# #include "mozilla-config.h" to each of your .cpp files.
GECKO_CONFIG_INCLUDE = -include mozilla-config.h

GECKO_DEFINES  = -DXPCOM_GLUE -DMOZILLA_STRICT_API

GECKO_INCLUDES = -I $(GECKO_SDK_PATH)                    \
                 -I $(GECKO_SDK_PATH)/xpcom/include      \
                 -I $(GECKO_SDK_PATH)/nspr/include       \
                 -I $(GECKO_SDK_PATH)/string/include     \
                 -I $(GECKO_SDK_PATH)/embedstring/include

GECKO_LDFLAGS =  -L $(GECKO_SDK_PATH)/xpcom/bin -lxpcomglue \
                 -L $(GECKO_SDK_PATH)/nspr/bin -lnspr4      \
                 -L $(GECKO_SDK_PATH)/nspr/bin -lplds4      \
                -L $(GECKO_SDK_PATH)/embedstring/bin/ -lembedstring

build:
        $(CXX) -o MozShim.so $(GECKO_CONFIG_INCLUDE)
$(GECKO_DEFINES) $(GECKO_INCLUDES) $(GECK\
O_LDFLAGS) $(CPPFLAGS) $(CXXFLAGS) MozShim.cpp
        chmod +x MozShim.so

clean:
        rm MozShim.so
```

Figure 9. Sample Makefile for the Gecko SDK

Appendix B: XPCOM API Reference

XPCOM Core

XPCOM is Mozilla’s Cross Platform Component Object Model. It is used to unify the creation, ownership, and deletion of objects and other data throughout Mozilla. The following interfaces are the core interfaces associated with the functioning of XPCOM.

Included Interfaces:

- nsISupports
- nsIInterfaceRequestor
- nsIWeakReference
- nsIMemory
- nsIProgrammingLanguage

nsISupports

This is the interface from which all other XPCOM interfaces inherit. It provides the foundation for XPCOM interface discovery and lifetime object management. In most of our examples through the book, we used `nsCOMPtr` which calls these methods. For example, when we made calls to `do_QueryInterface`, it made a call to the method `QueryInterface` on the target object.

Methods:

- **QueryInterface**
- **AddRef**
- **Release**

QueryInterface

Provides a mechanism for requesting interfaces to which a given object might provide access. The semantics of `QueryInterface` dictate that given an interface A that you call `QueryInterfce` on to get to interface B, you must be able to call `QueryInterface` on B to get back to A.

Syntax:

```
nsresult QueryInterface(const nsIID & uuid, void *
*result);
```

Parameters:

`uuid:` The `IID` of the requested interface.

`result:` [out] The reference to return. If this method call was successful, this out parameter will have had its reference count increased by one, effectively making the caller and owner of this object.

Result:

`NS_OK` if the interface was successfully returned.

`NS_NOINTERFACE` if the object does not support the given interface.

Example

```
#include "nsIComponentRegistrar.h"

nsIComponentRegistrar* compReg = nsnull;
nsresult rv =
   aCompMgr->QueryInterface(kIComponentRegistrarIID,(void**)& comp);
```

You can also use `nsCOMPtr` helper method `do_QueryInterface` that calls through to the `QueryInterface` method of the `aCompMgr` object:

```
#include "nsIComponentRegistrar.h"
nsresult rv;
nsCOMPtr<nsIComponentRegistrar> compRef =
  do_QueryInterface(aCompMgr, &rv);
```

AddRef

Increments the internal refcount of the interface.

Syntax:

```
nsrefcnt AddRef()
```

Parameters:

None.

Result:

The refcount.

Release

Decrements the internal refcount of the interface. When the count reaches zero, the interface deletes itself. To prevent objects leaking, every reference count must be accounted for. For example, if you call `QueryInterface` on an object, the result of this must be release at some point before the application shuts down. To release an object you must either use `nsCOMPtr`, a smart pointer which keeps track of references, or you must manually called `Release` on that object.

Syntax:

```
nsrefcnt Release()
```

Parameters:

None.

Result:

The refcount.

nsIInterfaceRequestor

This interface defines a generic interface for requesting interfaces to which a given object might provide access. It is very similar to `QueryInterface` found in `nsISupports`. The main difference is that interfaces returned from `GetInterface` are not required to provide a way back to the object implementing this interface. The semantics of `QI` dictate that given an interface A that you `QI` on to get to interface B, you must be able to `QI` on B to get back to A. This interface, however, allows you to obtain an interface C from A that may or most likely will not have the ability to get back to A.

Methods:

- **GetInterface**

GetInterface

Retrieves the specified interface pointer.

Syntax:

```
nsresult GetInterface(const nsIID & uuid, void *
*result);
```

Parameters:

`uuid:` The IID of the interface being requested.
`result:` [out] The interface pointer to be filled in if the interface is accessible.

Result:

NS_OK if the interface was successfully returned.
NS_NOINTERFACE if the interface is not accessible.
NS_ERROR* if there is method failure.

Example:

The interface that mWebBrowser references is a nsIInterfaceRequestor. We can ask this interface, if it knows anything about the nsIWebBrowser. If it does, it will return that object:

```
nsCOMPtr<nsIWebBrowserFind> finder(do_GetInterface(mWebBrowser));
```

nsIWeakReference

This interface gives access to a proxy object that cooperates with its referent to give clients a non-owning, non-dangling reference. Clients own the proxy, and should generally manage it with an nsCOMPtr as they would any other XPCOM object. The QueryReferent member function provides a owning reference on demand, through which clients can get useful access to the referent, while it still exists.

There are two common usage of this interface. The first is used for breaking a shutdown problems where the implementing object may be deleted without the owning object knowing about it. The second usage of this interface is to break circular dependencies. A circular dependency is when object A refers to object B and at the same time, object B refers to object A. In this case, special measures must be taken to avoid memory leaks.

Methods:

- **QueryReferent**

QueryReferent

Queries the referent, if it exists, and like `QueryInterface`, returns an owning reference to the desired interface.It is designed to look and act exactly like (a proxied) `QueryInterface`. Don't hold on to the produced interface permanently; that would defeat the purpose of using a non-owning `nsIWeakReference` in the first place.

Syntax:

```
nsresult QueryReferent(const nsIID & uuid, void *
*result);
```

Parameters:

`uuid:` The `IID` of the interface being requested.

`result:` [out] The interface pointer to be filled in if the interface is accessible.

Result:

`NS_OK` if successful.

Example

```
nsCOMPtr<nsIWeakReference>
   thisListener(dont_AddRef(NS_GetWeakReference(listener)));
```

nsIMemory

This interface is used to allocate and deallocate memory. It also provides for notifications in low-memory situations. This interface must be used to allocate all memory that is passed between interface bountries. For example, if an interface passes a memory buffer with the expectation that the buffer is now owned by the caller, the assuption is that this memory buffer will be freed by the nsIMemory. This rule need only apply to memory which passes through the interface boundry. Internal component memory usage can use any allocator.

There is a static helper class known as the `nsMemory` which aides in using the `nsIMemory`. `nsMemory` allows you to quickly obtain memory access without having to aquire the pointer to the `nsIMemory` interface.

A client that wishes to be notified of low memory situations (for example, because the client maintains a large memory cache that could be released when memory is tight) should register with the observer service (see `nsIObserverService`) using the topic *memory-pressure*.

There are three specific types of notications that can occur. These types will be passed as the `aData` parameter of the of the "memory-pressure" notification:

low-memory: This will be passed as the extra data when the pressure observer is being asked to flush for low-memory conditions.

heap-minimize: This will be passed as the extra data when the pressure observer is being asked to flush because of a heap minimize call.

alloc-failure: This will be passed as the extra data when the pressure observer has been asked to flush because a `malloc()` or `realloc()` has failed.

Methods:

- **Alloc**
- **Realloc**
- **Free**
- **HeapMinimize**
- **IsLowMemory**

Alloc

Allocates a block of memory of a particular size. If the memory cannot be allocated (because of an out-of-memory condition), null is returned.

Syntax:

```
void * Alloc(size_t size)
```

Parameters:

`size:` The size of the block to allocate.

Returns:

The block of memory.

Realloc

Reallocates a block of memory to a new size.

Syntax:

```
void * Realloc(void * ptr, size_t newSize);
```

Parameters:

`ptr:` The block of memory to reallocate.
`size`: The new size.

Returns:

The reallocated block of memory

Note: If `ptr` is null, this function behaves like `malloc`. If `s` is the size of the block to which `ptr` points, the first `min(s, size)` bytes of `ptr`'s block are copied to the new block. If the allocation succeeds, `ptr` is freed and a pointer to the new block returned. If the allocation fails, `ptr` is not freed and null is returned. The returned value may be the same as `ptr`.

Free

Frees a block of memory. Null is a permissible value, in which case nothing happens.

Syntax:

```
void Free(void * ptr);
```

Parameters:

`ptr:` The block of memory to free.

Returns:

None.

HeapMinimize

Attempts to shrink the heap. This method is scriptable.

Syntax:

```
nsresult HeapMinimize(PRBool immediate);
```

Parameters:

`immediate:` If the value is true, heap minimization will occur immediately if the call was made on the main thread. If the value is false, the flush will be scheduled to happen when the app is idle.

Result:

`NS_ERROR_FAILURE` if 'immediate' is set and the call was not on the application's main thread.

IsLowMemory

Indicates a low-memory situation (what constitutes low-memory is platform dependent). This can be used to trigger the memory pressure observers.

Syntax:

```
nsresult IsLowMemory(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if memory is low.

`FALSE` otherwise.

nsIProgrammingLanguage

Enumeration of programming languages. These values are used by the `nsIClassInfo` interface and indicate in what programming language the component was implemented. Note that the `ZX81_BASIC` value is a joke.

Constants

- `UNKNOWN`
- `CPLUSPLUS`

- `JAVASCRIPT`
- `PYTHON`
- `PERL`
- `JAVA`
- `ZX81_BASIC`
- `JAVASCRIPT2`

This list can grow indefinitely. Existing items, however, must not be changed.

XPCOM Components

These interfaces provide access to XPCOM's mechanisms for creating, managing and destroying objects.

Included Interfaces:

- nsIComponentManager
- nsIFactory
- nsIModule
- nsIComponentRegistrar
- nsIServiceManager
- nsIClassInfo

nsIComponentManager

This interface accesses the mechanism used to organize and create objects in XPCOM.

Methods:

- **GetClassObject**
- **GetClassObjectByContractID**
- **CreateInstance**
- **CreateInstanceByContractID**

GetClassObject

Returns the class object represented by the `CID aClass`. The result is an object that implements the `nsIFactory` interface. The result may also implement the `nsIClassInfo` interface.

Syntax:

```
nsresult GetClassObject(const nsCID & aClass, const
nsIID & aIID, void * *result);
```

Parameters:

`aClass:` The class ID of the class whose factory is being requested.

`aIID:` The interface ID of the interface that the factory's class implements.

`result`: [out] The reference to return.

Result:

`NS_OK` if successful.

GetClassObjectByContractID

This method is exactly the same as `getClassObject` but instead of a `CID` parameter, this method takes a contract ID string.

Syntax:

```
nsresult GetClassObjectByContractID(const char
*aContractID, const nsIID & aIID, void * *result);
```

Parameters:

`aContractID:` The contract ID of the class whose factory is being requested.

`aIID:` The interface ID of the interface that the factory's class implements.

`result`: [out] The reference to return.

Result:

`NS_OK` if successful.

CreateInstance

Creates an instance of the class indicated by the class ID and returns the interface indicated by the interface ID.

Syntax:

```
nsresult CreateInstance(const nsCID & aClass,
nsISupports *aDelegate, const nsIID & aIID, void *
*result);
```

Parameters:

`aClass:` The class ID of the requested class.

`aDelegate:` Used for aggregation.

`aIID:` The ID of the requested interface.

`result:` [out] The reference to return.

Result:

`NS_OK` if successful

Example

```
#include "nsXPCOMCID.h"
#include "nsXPCOM.h"
#include "nsIComponentManager.h"
#include "nsISupportsPrimitives.h"

static NS_DEFINE_CID(kSupportsStringCID, NS_SUPPORTS_CSTRING_CID);

nsCOMPtr<nsIComponentManager> compMgr;
rv = NS_GetComponentManager(getter_AddRefs(compMgr));
if (NS_FAILED(rv)) return rv;

nsISupportsCString* stringSupports;
compMgr->CreateInstance(kSupportsStringCID, nsnull,
                             NS_GET_IID(nsISupportsCString),
                             (void**)&stringSupports);

if (!stringSupports)
        return NS_ERROR_UNEXPECTED;
```

CreateInstanceByContractID

Creates an instance of the class indicated by the contract ID string and returns the interface indicated by the interface ID.

Syntax:

```
nsresult CreateInstanceByContractID(const char
*aContractID, nsISupports *aDelegate, const nsIID &
aIID, void * *result);
```

Parameters:

`aContractID`: The contract ID of the requested class.

`aDelegate`: Used for aggregation.

`aIID`: The ID of the requested interface.

`result`: [out] The reference to return.

Result:

`NS_OK` if successful

Example

```
#include "nsXPCOMCID.h"
#include "nsXPCOM.h"
#include "nsIComponentManager.h"
#include "nsISupportsPrimitives.h"

nsCOMPtr<nsIComponentManager> compMgr;
rv = NS_GetComponentManager(getter_AddRefs(compMgr));
if (NS_FAILED(rv)) return rv;

nsISupportsCString* stringSupports;
compMgr-
>CreateInstanceByContractID(NS_SUPPORTS_CSTRING_CONTRACTID, nsnull,
                             NS_GET_IID(nsISupportsCString),
                             (void**)&stringSupports);

if (!stringSupports)
        return NS_ERROR_UNEXPECTED;
```

nsIFactory

A class factory allows the creation of `nsISupports` derived components without specifying a concrete base class.

See “webLock1.cpp” on page 68 for a complete listing of a sample `nsIModule` implementation.

Methods:

- **CreateInstance**
- **LockFactory**

CreateInstance

Creates an instance of an object that implements the specified `IID`.

Syntax:

```
nsresult CreateInstance(nsISupports *aOuter, const
nsIID & iid, void * *result)
```

Parameters:

`aOuter`: Pointer to a component that wishes to be aggregated in the resulting instance. This will be nsnull if no aggregation is requested.

`iid:` The IID of the interface being requested in the component which is being currently created.

`result:` [out] Pointer to the newly created instance, if successful.

Result:

`NS_OK` if the component was successfully created and the interface being requested was successfully returned in result.

`NS_NOINTERFACE` if the interface not accessible.

`NS_ERROR_NO_AGGREGATION` if an ’outer’ object is supplied, but the component is not aggregatable.

`NS_ERROR*` if there is a method failure.

LockFactory

Provides the client a way to keep the component in memory until the client is finished with it. The client can call LockFactory(`PR_TRUE`) to lock the factory and LockFactory(`PR_FALSE`) to release the factory.

In the generic factory code we used thorough our examples, this method is a noop.

Syntax:

```
nsresult LockFactory(PRBool lock);
```

Parameters:

`lock` must be `PR_TRUE` or `PR_FALSE`

Result:

`NS_OK` if the lock operation was successful.
`NS_ERROR*` if there is a method failure.

nsIModule

This interface handles module registration and management. A module is a non empty set of factories. The main entry point of all XPCOM component librares are expected to return a `nsIModule` implementation. This interface is meant to be implemented by a component and called only from inside XPCOM. Developers are not encouraged to call on this interface directly, but instead use the Component and Service Manager for component access and control.

See page “webLock1.cpp” on page 68 for a complete listing of a sample `nsIModule` implementation.

Methods:

- **GetClassObject**
- **RegisterSelf**
- **UnregisterSelf**
- **CanUnload**

GetClassObject

Obtains a factory object from an `nsIModule` for a given `CID` and `IID` pair. This method is called from XPCOM when XPCOM wants to discover a class object for CID implementing a given IID in a component library.

Syntax:

```
nsresult GetClassObject(nsIComponentManager *aCompMgr,
const nsCID & aClass, const nsIID & aIID, void *
*aResult)
```

Parameters:

`aCompMgr:` The component manager.

`aClass:` The class ID of the class for which aResult is the factory.

`aIID:` The interface ID of the requested interface.

`aResult:` [out] The reference to return.

Result:

`NS_OK` if successful.

RegisterSelf

registerSelf is called by XPCOM on a component giving the component time to register with the Component Manager.

Syntax:

```
nsresult RegisterSelf(nsIComponentManager *aCompMgr,
nsIFile *aLocation, const char *aLoaderStr, const char
*aType)
```

Parameters:

`aCompMgr`: The Component Manager. This interface may be queried to the nsIComponentRegistrar for registration needs.

`aLocation` : The location of the module on disk .

`aLoaderStr`: Opaque loader specific string. This value is meant to be passed into the registration methods of nsIComponentRegistrar unmodified.

`aType`: Loader Type being used to load this module. This value is meant to be passed into the registration methods of nsIComponentRegistrar unmodified.

Result:

`NS_OK` if successful.

UnregisterSelf

unregisterSelf is called by XPCOM on a component giving the component time to unregister itself from the Component Manager.

Syntax:

```
nsresult UnregisterSelf(nsIComponentManager *aCompMgr,
nsIFile *aLocation, const char *aLoaderStr)
```

Parameters:

`aCompMgr` : The component manager. This interface may be queried to the nsIComponentRegistrar for registration needs.

`aLocation` : The location of the module on disk. This value is meant to be passed into the registration methods of nsIComponentRegistrar unmodified.

`aLoaderStr`: Opaque loader specific string.This value is meant to be passed into the registration methods of nsIComponentRegistrar unmodified.

Result:

`NS_OK` if successful.

CanUnload

Indicates that the module is willing to be unloaded. This does not guarantee that the module will be unloaded. Unless you know that you component can be unloaded safely, you must return FALSE.

The generic module always returns false.

Syntax:

```
nsresult CanUnload(nsIComponentManager *aCompMgr,
PRBool *_retval)
```

Parameters:

`aCompMgr:` The component manager.

Return:

`PR_TRUE` if the module is willing to be unloaded. It is very important to check that no outstanding references to the module's code/data exist before returning true.

`PR_FALSE` guarantees that the module will not be unloaded.

nsIComponentRegistrar

This interface handles all component registration and management in XPCOM. There are basically four conceptual parts to this interface: (1) queries to see what has been registered, (2) register methods that take an in-memory instance of nsIFactory objects, (3) register components that exist at a specific place, and finally (4) callbacks that allow components to register themselves during nsIModule method RegisterSelf.

Methods:

- **AutoRegister**
- **AutoUnregister**
- **RegisterFactory**
- **UnregisterFactory**
- **RegisterFactoryLocation**

- **UnregisterFactoryLocation**
- **IsCIDRegistered**
- **IsContractIDRegistered**
- **EnumerateCIDs**
- **EnumerateContractIDs**
- **CIDToContractID**
- **ContractIDToCID**

AutoRegister

Registers a component file or all component files in a directory. This method is usually called by the application to register a component or all components in a directory. If a directory is pass, the directory will be recursively traversed. Each component file must be valid as defined by acomponent's loader. For example, if the given file is a native library, it must export the symbol `NSGetModule`. Other loaders may have different semantics.

This method may only be called from the main thread.

Syntax:

```
nsrsult AutoRegister(nsIFile *aSpec);
```

Parameters:

> `aSpec:` Filename spec for component file's location. If `aSpec` is a directory, then every component file in the directory will be registered. If the `aSpec` is null, then the application component's directory and the GRE components directory, if one exists, will be registered. (see *nsIDirectoryService.idl*)

Result:

`NS_OK` if registration was successful.

`NS_ERROR` if not.

```
#include "nsXPCOMCID.h"
#include "nsXPCOM.h"
#include "nsIComponentRegistrar.h"

nsCOMPtr<nsIComponentRegistrar> registrar;
nsresult rv = NS_GetComponentRegistrar(getter_AddRefs(registrar));
if (NS_FAILED(rv)) return rv;

rv = registrar->AutoRegister(nsnull);
```

AutoUnregister

Similar to `autoRegister`, this method registers a component file or all component files in a directory. This method is usually called by the application to unregister a component or all components in a directory.

This method may only be called from the main thread.

Syntax:

```
nsresult AutoUnregister(nsIFile *aSpec)
```

Parameters:

`aSpec:` Filename spec for component file's location. If `aSpec` is a directory, then every component file in the directory will be registered. If the `aSpec` is null, then the application component's directory and the GRE components directory, if one exists, will be registered. (see *nsIDirectoryService.idl*)

Result:

`NS_OK` if unregistration was successful.

`NS_ERROR` if not.

RegisterFactory

Registers a instantiated `nsIFactory` factory object with a given ContractID, CID, and Class Name. This `nsIFactory` object will only be registered until XPCOM shuts down.

Syntax:

```
nsresult RegisterFactory(const nsCID & aClass, const
char *aClassName, const char *aContractID, nsIFactory
*aFactory)
```

Parameters:

`aClass:` The CID.

`aClassName:` The Class Name associated with `aClass`.

`aContractID:` The ContractID associated with `aClass.`

`aFactory:` The factory that will be registered for `aClass.`

Result:

`NS_OK` if registration was successful.

`NS_ERROR` if not.

UnregisterFactory

Unregisters a factory associated with CID `aClass`.

Syntax:

```
nsresult UnregisterFactory(const nsCID & aClass,
nsIFactory *aFactory)
```

Parameters:

`aClass:` The CID being unregistered.

`aFactory:` The factory that will be unregistered for `aClass.`

Result:

`NS_OK` if unregistration was successful.

`NS_ERROR` if not.

RegisterFactoryLocation

This is a low level method that allows registration of a factory object given ContractID, CID and Class Name, location, and so forth. This call is usually maded from the `nsIModule` implementations of a component library to register its factories.

Syntax:

```
nsresult RegisterFactoryLocation(const nsCID & aClass,
const char *aClassName, const char *aContractID, nsIFile
*aFile, const char *aLoaderStr, const char *aType)
```

Parameters:

`aClass:` The CID of the class.

`aClassName:` The Class Name of `aClass.`

`aContractID:` The ContractID associated with `aClass.`

`aFile: The` Component File. This file must have an associated loader and export the required symbols which this loader specifies.

`aLoaderStr:` An opaque loader specific string. This value is passed into the `nsIModule`'s `registerSelf` callback and must be fowarded unmodified when registering factories via their location.

`aType:` The Component Type of `aClass`. This value is passed into the `nsIModule`'s `registerSelf` callback and must be fowarded unmodified when registering factories via their location.

Result:

`NS_OK` if registration was successful.

`NS_ERROR` if not.

See page “webLock1.cpp” on page 68 for a complete listing of a sample `nsIModule` implementation which calls `registerFactoryLocation`.

UnregisterFactoryLocation

This is a low level method that allows unregistering a factory associated with `aClass.` This call is usually maded from the nsIModule implementations of a component library to unregister its factories.

Syntax:

```
nsresult UnregisterFactoryLocation(const nsCID &
aClass, nsIFile *aFile)
```

Parameters:

`aClass:` The CID being unregistered.

`aFile: The` Component File previously registered.

Result:

`NS_OK` if unregistration was successful.

`NS_ERROR` if not.

See page “webLock1.cpp” on page 68 for a complete listing of a sample `nsIModule` implementation which calls `unregisterFactoryLocation`.

IsCIDRegistered

Returns true if a factory is registered for a CID.

Syntax:

```
nsresult IsCIDRegistered(const nsCID & aClass, PRBool
*_retval)
```

Parameters:

`aClass:` The CID queried for registration.

Returns:

`TRUE` if a factory is registered for the CID

FALSE if not.

IsContractIDRegistered

Returns true if a factory is registered for a ContractID.

Syntax:

```
nsresult IsContractIDRegistered(const char
*aContractID, PRBool *_retval)
```

Parameters:

`aContractID:` The ContractID queried for registration.

Returns:

`TRUE` if a factory is registered for the ContractID.

`FALSE` if not.

EnumerateCIDs

Enumerates the list of all registered CIDs. Elements of the enumeration can be `QueryInterface`'d for the `nsISupportsID` interface. From the `nsISupportsID`, you can obtain the actual CID.

Syntax:

```
nsresult EnumerateCIDs(nsISimpleEnumerator **_retval)
```

Parameters:

None.

Returns:

An enumerator for CIDs

EnumerateContractIDs

Enumerates the list of all registered ContractIDs. Elements of the enumeration can be `QueryInterface`'d for the `nsISupportsCString` interface. From the `nsISupportsCString` interface, you can obtain the actual Contract ID string.

Syntax:

```
EnumerateContractIDs(nsISimpleEnumerator **_retval)
```

Parameters:

None.

Returns:

An enumerator for ContractIDs.

CIDToContractID

Gets the ContractID for a given CID, if one exists and is registered.

Syntax:

```
nsresult CIDToContractID(const nsCID & aClass, char
**_retval)
```

Parameters:

`aClass`: The CID whose ContractID is being sought.

Returns:

The ContractID.

ContractIDToCID

Gets the CID for a given Contract ID, if one exists and is registered.

Syntax:

```
nsresult ContractIDToCID(const char *aContractID, nsCID
* *_retval)
```

Parameters:

`aContractID`: The ContractID whose CID is being sought.

Returns:

The CID.

nsIServiceManager

This interface provides a means to obtain global services (ie, access to a singleton object) in an application. . Users of the service manager must first obtain a pointer to the global service manager by calling `NS_GetServiceManager`. After that, they can request specific services by calling `GetService`. When they are finished they should NS_RELEASE the service as usual. A user of a service may keep references to particular services indefinitely and must call Release only when XPCOM shuts down.

Methods:

- **GetService**
- **GetServiceByContractID**
- **IsServiceInstantiated**
- **IsServiceInstantiatedByContractID**

GetService

Returns the object that implements `aClass` and the interface `aIID`. This may result in the object being created.

Syntax:

```
nsresult GetService(const nsCID & aClass, const nsIID &
aIID, void * *result)
```

Parameters:

`aClass:` The class ID of the requested class.
`aIID` : The interface ID of the requested interface.
`result` :[out] The resulting service

Result:

`NS_OK` if successful.

Example

```
nsCOMPtr<nsIServiceManager> mgr;
NS_GetServiceManager(getter_AddRefs(mgr));
if (mgr)
rv = mgr->GetService(mCID, aIID, (void**)aInstancePtr);
```

GetServiceByContractID

Returns the object that implements `aContractID` and the interface `aIID`. This may result in the instance being created.

Syntax:

```
void nsIServiceManager::GetServiceByContractID(in
string aContractID, in nsIIDRef
aIID,[iid_is(aIID),retval] out nsQIResult result)
```

Parameters:

`aContractID:` The contract ID of the requested class.

`aIID` : The interface ID of the requested interface.

`result` :[out] The reference to return.

Result:

`NS_OK` if successful.

Example

```
nsCOMPtr<nsIServiceManager> mgr;
NS_GetServiceManager(getter_AddRefs(mgr));
if (mgr)
rv = mgr->GetServiceByContractID(
  mContractID, aIID, (void**)aServicePtr);
```

IsServiceInstantiated

Returns TRUE if the singleton service object has already been created.

Syntax:

```
nsresult IsServiceInstantiated(const nsCID & aClass,
const nsIID & aIID, PRBool *_retval)
```

Parameters:

`aClass`: The class ID of the requested class.
`aIID` : The interface ID of the requested interface.

Return:

`PR_TRUE` if the object has already been created.
`PR_FALSE` if the object has not been created.

IsServiceInstantiatedByContractID

ReturnsTRUE if the singleton service object has already been created.

Syntax:

```
nsresult IsServiceInstantiatedByContractID(const char
*aContractID, const nsIID & aIID, PRBool *_retval)
```

Parameters:

`aContractID`: The contract ID of the requested class.
`aIID` : The interface ID of the requested interface.

Result:

`PR_TRUE` if the object has already been created.
`PR_FALSE` if the object has not been created.

nsIClassInfo

This interface provides information about a specific implementation class.

Methods:

- **GetInterfaces**
- **GetHelperForLanguage**
- **GetContractID**

- **GetClassDescription**
- **GetClassID**
- **GetImplementationLanguage**
- **GetClassIDNoAlloc**
- **GetFlags**

GetInterfaces

Returns an ordered list of the interface IDs that instances of the class promise to implement. Note that `nsISupports` is an implicit member of any such list and need not be included. Should set `*count = 0` and `*array = null` and return `NS_OK` if getting the list is not supported.

Syntax:

```
nsresult GetInterfaces(PRUint32 *count, nsIID * **array)
```

Parameters:

`count:`	[out]The number of implemented interfaces.
`array:`	[out] The list of implemented interfaces.

Result:

`NS_OK` if successful, or if getting list is not supported.

GetHelperForLanguage

Gets a language mapping specific helper object that may assist in using objects of this class in a specific lanaguage. For instance, if asked for the helper for `nsIProgrammingLanguage::JAVASCRIPT` this might return an object that can be QI'd into the `nsIXPCScriptable` interface to assist XPConnect in supplying JavaScript specific behavior to callers of the instance object. Should return null if no helper available for given language.

Syntax:

```
nsresult GetHelperForLanguage(PRUint32 language,
nsISupports **_retval)
```

Parameters:

`language`: An integer representing the requested language.

Returns:

The helper object.

See also: `nsIProgrammingLanguage` for language constants.

GetContractID

Returnes a string contract ID through which an instance of this class can be created (or accessed as a service, if `flags & SINGLETON`), or null.

Syntax:

```
nsresult GetContractID(char * *aContractID)
```

GetClassDescription

Gets a human readable string naming the class, or null.

Syntax:

```
nsresult GetClassDescription(char * *aClassDescription)
```

GetClassID

Gets the class ID through which an instance of this class can be created (or accessed as a service, if `flags & SINGLETON`), or null.

Syntax:

```
nsresult GetClassID(nsCID * *aClassID);
```

GetImplementationLanguage

Gets the language type from list in `nsIProgrammingLanguage`.

Syntax:

```
nsresult GetImplementationLanguage(PRUint32
*aImplementationLanguage);
```

GetClassIDNoAlloc

Gets a class ID through which an instance of this class can be created (or accessed as a service, if `flags & SINGLETON`). If the class does not have a CID, it should return NS_ERROR_NOT_AVAILABLE. This attribute exists so C++ callers can avoid allocating and freeing a CID, as would happen if they used class ID.

Syntax:

```
nsresult GetClassIDNoAlloc(nsCID *aClassIDNoAlloc)
```

GetFlags

Returns implementation flags which made be ORed together of the values below.

Syntax:

```
nsresult GetFlags(PRUint32 *aFlags)
```

Flags:

Note: The high order bit is RESERVED for consumers of these flags. No implementor of this interface should ever return flags with this bit set.

SINGLETON

Flag specifies that the object is a singleton or service.

THREADSAFE

Flag specifies that the object is a threadsafe.

MAIN_THREAD_ONLY

Flag specifies that the object may only be called from the main (UI) thread.

DOM_OBJECT

Flag specifies that the object is part of the DOM.

PLUGIN_OBJECT

Flag specifies that the object is a plugin object.

CONTENT_NODE

Flag specifies that the object is a content node.

EAGER_CLASSINFO

Flag tells the generic module code to construct a factory object during initialization.

RESERVED

Flags saved for future use.

XPCOM Data Structures

These interfaces provide access to various utility mechanisms in XPCOM.

Included Interfaces:

- nsICategoryManager
- nsIObserver
- nsIObserverService
- nsIProperties

- nsISimpleEnumerator
- nsISupportsPrimitives

nsICategoryManager

This interface is implemented by an object that wants to observe an event corresponding to a topic.

Methods:

- **GetCategoryEntry**
- **AddCategoryEntry**
- **DeleteCategoryEntry**
- **DeleteCategory**
- **EnumerateCategory**
- **EnumerateCategories**

GetCategoryEntry

Get the value for the given category's entry.

Syntax:

```
nsresult GetCategoryEntry(const char *aCategory, const
char *aEntry, char **_retval)
```

Parameters:

`aCategory`: The name of the category (e.g. "protocol").
`aEntry`: The entry you're looking for (e.g. "http").
`_retval`: [out] returns the category entry corresponding to the category and entry.

Result:

`NS_OK` if successful.

AddCategoryEntry

Add an entry to a category.

Syntax:

```
nsresult AddCategoryEntry(const char *aCategory, const
char *aEntry, const char *aValue, PRBool aPersist,
PRBool aReplace, char **_retval)
```

Parameters:

`aCategory:` The name of the category (e.g. "protocol").

`aEntry:` The entry you're looking for (e.g. "http").

`aValue:` The value for the entry ("moz.httprulez.1")

`aPersist:` Should we persist between invocations? This value is ignored in some implementations.

`aReplace:` Should we replace an existing entry? This value is ignored in some implementations.

`_retval:` [out] Returns the previous category entry corresponding to the category and entry Previous entry, if any..

Result:

`NS_OK` if successful.

DeleteCategoryEntry

Delete an entry from the category.

Syntax:

```
nsresult DeleteCategoryEntry(const char *aCategory,
const char *aEntry, PRBool aPersist)
```

Parameters:

`aCategory:` The name of the category (e.g. "protocol").

`aEntry:` The entry you're looking for (e.g. "http").

`aPersist:` Should we persist between invocations? This value is ignored in some implementations.

Result:

`NS_OK` if successful.

DeleteCategory

Delete a category and all entries.

Syntax:

```
nsresult DeleteCategory(const char *aCategory)
```

Parameters:

`aCategory:` The name of the category (e.g. "protocol").

Result:

`NS_OK` if successful.

EnumerateCategory

Enumerate the entries in a category.

Syntax:

```
nsresult EnumerateCategory(const char *aCategory,
nsISimpleEnumerator **_retval)
```

Parameters:

`aCategory:` The name of the category (e.g. "protocol").

`_retval:` [out] returns an enumeration of all of the entries in the given category. The elements of the enumeration can be QueryInterface'd for the nsISupportsCString interface. From the nsISupportsCString, you can obtain the actual string of the entry.

Result:

`NS_OK` if successful.

EnumerateCategories

Enumerate the entries in a category.

Syntax:

```
nsresult EnumerateCategories(nsISimpleEnumerator
**_retval)
```

Parameters:

> `_retval:` [out] returns an enumeration of all of the category. The elements of the enumeration can be QueryInterface'd for the nsISupportsCString interface. From the nsISupportsCString, you can obtain the actual string of the category name.

Result:

> `NS_OK` if successful.

nsIObserver

This interface is implemented by an object that wants to observe an event corresponding to a topic.

See also: `nsIObserverService.`

Methods:

- **Observe**

Observe

Called when there is a notification for the topic `aTopic`. The object implementing this interface must have been registered with an observer service such as the `nsIObserverService`. If you expect multiple topics/subjects, your implementation is responsible for filtering. You should not modify, add, remove, or enumerate notifications in the implemention of observe.

Syntax:

```
nsresult Observe(nsISupports *aSubject, const char
*aTopic, const PRUnichar *aData)
```

Parameters:

`aSubject:` Notification specific interface pointer.

`aTopic:` The notification topic or subject.

`aData:` Notification specific wide string, depending on the subject event.

Result:

`NS_OK` if successful.

See page “Getting Called at Startup” on page 91, where this interface is implemented.

nsIObserverService

Service allows a client listener (`nsIObserver`) to register and unregister for notifications of a specific string referenced topic. Service also provides a way to notify registered listeners and a way to enumerate registered client listeners

Methods:

- **AddObserver**
- **RemoveObserver**
- **NotifyObservers**
- **EnumerateObservers**

AddObserver

Registers a given listener for a notifications regarding the specified topic.

Syntax:

```
nsresult AddObserver(nsIObserver *anObserver, const
char *aTopic, PRBool ownsWeak)
```

Parameters:

`anObserver:` The interface pointer to the object which will receive notifications.

`aTopic`: The notification topic or subject.

`ownsWeak:` If set to false, the `nsIObserverService` will hold a strong reference to `anObserver`. If set to true and `anObserver` supports the `nsIWeakReference` interface, a weak reference will be held. Otherwise an error will be returned.

Result:

`NS_OK` if successful.

Example:

```
nsCOMPtr<nsIObserverService> observerService(
  do_GetService(/"@mozilla.org/observer-service;1"/, &rv));
if (NS_SUCCEEDED(rv))
  rv = observerService->AddObserver(
     this,
     NS_XPCOM_SHUTDOWN_OBSERVER_ID, PR_TRUE);
```

RemoveObserver

Unregisters a given listener from notifications regarding the specified topic.

Syntax:

`nsresult RemoveObserver(nsIObserver *anObserver, const char *aTopic)`

Parameters:

`anObserver:` The interface pointer to the object which should stop receiving notifications.

`aTopic:` The notification topic or subject.

Result:

`NS_OK` if successful.

Example:

```
nsCOMPtr<nsIObserverService> observerService(
   do_GetService(/"@mozilla.org/observer-service;1"/, &rv));
if (NS_SUCCEEDED(rv))
  rv = observerService->RemoveObserver(
    this,
    NS_XPCOM_SHUTDOWN_OBSERVER_ID);
```

NotifyObservers

Notifies all registered listeners of the given topic.

Syntax:

`nsresult NotifyObservers(nsISupports *aSubject, const char *aTopic, const PRUnichar *someData)`

Parameters:

`aSubject:` Notification specific interface pointer.
`aTopic:` The notification topic or subject.
`someData:` Notification specific wide string.

Result:

`NS_OK` if successful.

Example:

```
nsCOMPtr<nsIServiceManager> mgr;
NS_GetServiceManager(getter_AddRefs(mgr));

if (!mgr)
    return NS_ERROR_FAILURE;

nsCOMPtr<nsIObserverService> observerService;
rv = mgr->GetServiceByContractID("@mozilla.org/observer-service;1",
                                 NS_GET_IID(nsIObserverService),
                                 getter_AddRefs(observerService));

if (!observerService)
    return NS_ERROR_FAILURE;

char* string = "My Topic";
PRUnichar* wstring = nsnull;
nsISupports* context;

observerService->NotifyObservers( context, string, wstring );
```

EnumerateObservers

Returns an enumeration of all registered listeners.

See also: `nsISimpleEnumerator`.

Syntax:

```
nsresult EnumerateObservers(const char *aTopic,
nsISimpleEnumerator **_retval)
```

Parameters:

`aTopic:` The notification topic or subject.

Result:

`NS_OK` if successful.

Example

```
nsCOMPtr<nsIServiceManager> mgr;
NS_GetServiceManager(getter_AddRefs(mgr));
if (!mgr)
    return NS_ERROR_FAILURE;

nsCOMPtr<nsIObserverService> observerService;
mgr->GetServiceByContractID("@mozilla.org/observer-service;1",
                            NS_GET_IID(nsIObserverService),
                            getter_AddRefs(observerService));

if (!observerService)
    return NS_ERROR_FAILURE;

nsCOMPtr<nsISimpleEnumerator> theEnum;
rv = observerService->EnumerateObservers( <sometopic>,
                                         getter_AddRefs(theEnum));
if (theEnum)
{
    PRBool loop = PR_TRUE
    while (NS_SUCCEEDED(theEnum->HasMoreElements(&loop)) && loop)
    {
        nsCOMPtr<nsISupports> inst;
        theEnum->GetNext(getter_AddRefs(inst));

        if (inst) { do something useful }
    }
}
```

nsIProperties

This interface is .

See “The Directory Service” on page 112 for both an implementation of this interface and a client usage example.

Methods:

- **Get**
- **Set**
- **Has**

- **Undefine**
- **GetKeys**

Get

Gets a property with a given name.

Syntax:

```
nsresult Get(const char *prop, const nsIID & iid, void *
*result)
```

Parameters:

`prop:` A property name string key.
`iid:` The IID to QueryInterface any result of the Get.
`result:` [out] The property with the given name, if any.

Result:

`NS_OK` if successful.
`NS_ERROR_FAILURE` if a property with that name doesn't exist.
`NS_ERROR_NO_INTERFACE` if a property with that name doesn't exist.

Set

Sets a property with a given name to a given value.

Syntax:

```
nsresult Set(const char *prop, nsISupports *result)
```

Parameters:

`prop:` A property name string key.
`result:` The property with the given name.

Result:

`NS_OK` if successful.

Has

Returns true if the property with the given name exists.

Syntax:

```
nsresult Has(const char *prop, PRBool *_retval)
```

Parameters:

`prop:` A property name string key.
`_retval:` True if the propery exists with the given string key.

Result:

`NS_OK` if successful.

Undefine

Undefines a property.

Syntax:

```
nsresult Undefine(const char *prop)
```

Parameters:

`prop:` A property name string key.

Result:

`NS_OK` if successful.
`NS_ERROR_FAILURE` if a property with that name doesn't exist.

GetKeys

Returns an array of the keys.

Syntax:

```
nsresult GetKeys(PRUint32 *count, char ***keys)
```

Parameters:

`count:` If successful, count will contain the number of keys.

`keys:` An array of all keys in know by the object implementing the nsIProperties interface.

Result:

`NS_OK` if successful.

`NS_ERROR_FAILURE` if a property with that name doesn't exist.

nsISimpleEnumerator

This interface is used to enumerate over elements defined by its implementor. Although `hasMoreElements()` can be called independently of `getNext()`, `getNext()` must be preceded by a call to `hasMoreElements()`. There is no way to "reset" an enumerator, once you obtain one.

The `nsISimpleEnumerator` interface is shown in the section “The Web Locking Interface” on page 105.

Methods:

- **HasMoreElements**
- **GetNext**
- **GetType**

HasMoreElements

Determines whether or not the enumerator has any elements that can be returned via `getNext()`. This method is generally used to determine whether or not to initiate or continue iteration over the enumerator, though it can be called without subsequent `getNext()` calls. This does not affect internal state of enumerator.

Syntax:

```
nsresult HasMoreElements(PRBool *_retval)
```

Parameters:

None.

Result:

`PR_TRUE` if there are remaining elements in the enumerator.

`PR_FALSE` if there are no more elements in the enumerator.

GetNext

Called to retrieve the next element in the enumerator. The "next" element is the first element upon the first call. Must be preceded by a call to `hasMoreElements()` which returns `PR_TRUE.` This method is generally called within a loop to iterate over the elements in the enumerator.

Syntax:

```
nsresult GetNext(nsISupports **_retval)
```

Parameters:

None.

Result:

`NS_OK` if the call succeeded in returning a non-null value.

`NS_ERROR_FAILURE` if there are no more elements to enumerate.

nsISupportsPrimitives

This group of interfaces provide ways to wrap primiatives such as integers, floats, doubles, chars, strings, etc. so that these values may be passed between interface boundries using a `nsISupports` parameter. The base class of these primitive wrappers is the `nsISupportsPrimitive`.

These objects can be constructed by calling the Component Manager with the appropriate Class ID or Contract ID listed in *nsXPCOMCID.h*.

Also see “SetSites” on page 124, where we create and use an `nsISupportsCString` object.

GetType

Determines what kind of primative the nsISupportsPrimimitive subclass supports..

Syntax:

```
nsresult GetType(PRUint16 *aType)
```

Parameters:

`aType:` [out] The type of the primative. This value will be one of the following constants:

`TYPE_ID`	`TYPE_CSTRING`
`TYPE_STRING`	`TYPE_PRBOOL`
`TYPE_PRUINT8`	`TYPE_PRUINT16`
`TYPE_PRUINT32`	`TYPE_PRUINT64`
`TYPE_PRTIME`	`TYPE_CHAR`
`TYPE_PRINT16`	`TYPE_PRINT32`
`TYPE_PRINT64`	`TYPE_FLOAT`
`TYPE_DOUBLE`	`TYPE_INTERFACE_POINTER`
`TYPE_VOID`	

These first three are pointer types and do data copying using the `nsIMemory`.

The derived classes implement a getter and setter for its data type and a ToString method that returns a `nsIMemory` allocated string representative of the data. These interfaces are:

`nsISupportsID`	`nsISupportsString`
`nsISupportsCString`	`nsISupportsPRBool`
`nsISupportsPRUint8`	`nsISupportsPRUint16`
`nsISupportsPRUint32`	`nsISupportsPRUint64`
`nsISupportsPRTime`	`nsISupportsChar`

```
nsISupportsPRInt16          nsISupportsPRInt32
nsISupportsPRInt64          nsISupportsFloat
nsISupportsDouble           nsISupportsVoid
nsISupportsInterfacePointer
```

XPCOM I/O

These interfaces provide access to various XPCOM I/O related services.

Included Interfaces:

- nsIDirectoryServiceProvider
- nsIDirectoryServiceProvider2
- nsIDirectoryService
- Well Known Locations
- nsIFile
- nsIInputStream
- nsILocalFile
- nsIOutputStream

nsIDirectoryServiceProvider

This interface is used by Directory Service to get file locations.

Methods:

- **GetFile**
-

GetFile

Provides DirectoryService with a prop on the first request or on every request if the prop is not persistent.

Syntax:

```
nsresult GetFile(const char *prop, PRBool *persistent,
nsIFile **_retval)
```

Parameters:

`prop`: The symbolic name of the file.

`persistent`: [out] If true, the returned file will be cached by Directory Service. Subsequent requests for this prop will bypass the provider and use the cache. If false, the provider will be asked for this prop each time it is requested.

Returns:

The file represented by the property.

Example:

```
nsCOMPtr<nsIProperties> dirService;
rv = servMgr->GetServiceByContractID(
  NS_DIRECTORY_SERVICE_CONTRACTID,
  NS_GET_IID(nsIProperties),
  getter_AddRefs(dirService));

if (NS_FAILED(rv))
        return rv;

nsCOMPtr<nsIFile> xpcomDll;
rv = dirService->Get(
  NS_XPCOM_LIBRARY_FILE, NS_GET_IID(nsIFile),
  getter_AddRefs(xpcomDll));

if (NS_FAILED(rv))
        return rv;
```

nsIDirectoryServiceProvider2

This interface is an extension of `nsIDirectoryServiceProvider` which allows multiple files to be returned for a given key.

Methods:

- **GetFiles**

GetFiles

Provides Directory Service with a prop when it gets a request for it and the requested type is `nsISimpleEnumerator`.

Syntax:

```
nsresult GetFiles(const char *prop, nsISimpleEnumerator
**_retval)
```

Parameters:

`prop:` The symbolic name of the file list.

Returns:

An enumerator for a list of file locations. The elements in the enumeration are of the type `nsIFile`.

nsIDirectoryService

This interface provides XPCOM's directory service.

Methods:

- **Init**
- **RegisterProvider**
- **UnregisterProvider**

Init

Must be called. This function is used internally by XPCOM initialization and developers are not required nor expected to call this method.

Syntax:

```
nsresult Init(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

RegisterProvider

Registers a provider with the service.

Syntax:

```
nsresult RegisterProvider(nsIDirectoryServiceProvider
*prov)
```

Parameters:

`prov:` The provider to be registered. The service will keep a strong reference to this object. It will be released when the service is released.

Result:

`NS_OK` if successful.

UnregisterProvider

Unregisters a provider with the service.

Syntax:

```
nsresult UnregisterProvider(nsIDirectoryServiceProvider
*prov)
```

Parameters:

`prov:` The provider to be unregistered.

Result:

`NS_OK` if successful.

nsIFile

There are many built in location which are known to XPCOM and are accessible from the `nsIDirectoryService`. These values are defined in the *nsDirectoryServiceDefs.h*.

This interface is the only correct cross-platform way to specify a file. Strings are not such a way. Despite the fact that they work on Windows or Unix, they will not work here.

All methods with string parameters have two forms. The preferred form operates on UCS-2 encoded characters strings. An alternate form operates on characters strings encoded in the "native" charset. A string containing characters encoded in the native charset cannot be safely passed to javascript via xpconnect. Therefore, the UCS-2 forms are scriptable, but the "native methods" are not.

Methods:

- **Append**
- **AppendNative**
- **Normalize**
- **Create**
- **CopyTo**
- **CopyToNative**
- **CopyToFollowingLinks**
- **CopyToFollowingLinksNative**
- **MoveTo**
- **MoveToNative**
- **Remove**
- **Exists**
- **IsWritable**
- **IsReadable**
- **IsExecutable**
- **IsHidden**
- **IsDirectory**

- **IsFile**
- **IsSymlink**
- **IsSpecial**
- **CreateUnique**
- **Clone**
- **Equals**
- **Contains**

Attributes and Constants:

- **LeafName**
- **Permissions**
- **LastModificationTime**
- **FileSize**
- **Target**
- **Parent**
- **DirectoryEntries**

Append

AppendNative

Makes a descendent of the current `nsIFile`.

Syntax:

```
nsresult Append(const nsAString & node)
nsresult AppendNative(const nsACString & node)
```

Parameters:

`node`: A string which is intended to be a child node of the `nsIFile`. For the `appendNative` method, the node must be in the native filesystem charset.

Result:

`NS_OK` if successful.

Normalize

Normalizes the pathName (e.g. removing .. and . components on Unix).

Syntax:

```
nsresult Normalize(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

Create

Creates a new file or directory in the file system. Any nodes that have not been created or resolved, will be. If the file or directory already exists, `create()` returns `NS_ERROR_FILE_ALREADY_EXISTS`.

The type flag must be either `NORMAL_FILE_TYPE` or `DIRECTORY_TYPE`

Syntax:

```
nsresult Create(PRUint32 type, PRUint32 permissions)
```

Parameters:

`type`: This specifies the type of file system object to be made. The only two types at this time are file and directory which are defined below. If the type is unrecognized, returns (`NS_ERROR_FILE_UNKNOWN_TYPE`).

`permissions`: Unix style octal permissions. This may be ignored on systems that do not need to do permissions.

Result:

`NS_OK` if successful.

CopyTo

CopyToNative

Copies this file to the specified `newParentDir`. If a `newName` is specified, the file will be renamed. If 'this' is not created, returns the error (`NS_ERROR_FILE_TARGET_DOES_NOT_EXIST`). `copyTo` may fail if the file already exists in the destination directory. `copyTo` will NOT resolve aliases/ shortcuts during the copy.

Syntax:

```
nsresult CopyTo(nsIFile *newParentDir, const nsAString &
newName)
```

```
nsresult CopyToNative(nsIFile *newParentDir, const
nsACString & newName)
```

Parameters:

`newParentDir`: The destination directory. If the `newParentDir` is empty, `copyTo()` will use the parent directory of this file. If the `newParentDir` is not empty and is not a directory, an error will be returned (`NS_ERROR_FILE_DESTINATION_NOT_DIR`). For the `CopyToNative` method, the `newName` must be in the native filesystem charset.

`newName`: This param allows you to specify a new name for the file to be copied. This param may be empty, in which case the current leaf name will be used.

Result:

`NS_OK` if successful.

CopyToFollowingLinks

CopyToFollowingLinksNative

Is identical to `copyTo` except, as the name implies, it follows symbolic links. The XP_UNIX implementation always follows symbolic links when copying.

Syntax:

```
nsresult CopyToFollowingLinks(nsIFile *newParentDir,
const nsAString & newName)
```

```
  nsresult CopyToFollowingLinksNative(nsIFile
*newParentDir, const nsACString & newName)
```

Parameters:

`newParentDir`: The destination directory. If the `newParentDir` is empty, `copyTo()` will use the parent directory of this file. If the `newParentDir` is not empty and is not a directory, an error will be returned (`NS_ERROR_FILE_DESTINATION_NOT_DIR`).

`newName`: This param allows you to specify a new name for the file to be copied. This param may be empty, in which case the current leaf name will be used. For the `copyToFollowingLinksNative` method, the newName must be in the native filesystem charset.

Result:

`NS_OK` if successful.

MoveTo

MoveToNative

Moves this file to the specified `newParentDir`. If a `newName` is specified, the file will be renamed. If 'this' is not created, returns an error (`NS_ERROR_FILE_TARGET_DOES_NOT_EXIST`). `moveTo` will NOT resolve aliases/shortcuts during the copy. `moveTo` will do the right thing and allow copies across volumes.

Syntax:

```
nsresult MoveTo(nsIFile *newParentDir, const nsAString &
newName)
```

```
nsresult MoveToNative(nsIFile *newParentDir, const
nsACString & newName)
```

Parameters:

`newParentDir`: This param is the destination directory. If the newParentDir is empty, `moveTo()` will rename the file within its current directory. If the newParentDir is not empty and does not name a directory, an error will be returned (`NS_ERROR_FILE_DESTINATION_NOT_DIR`). For the `moveToNative` method, the newName must be in the native filesystem charset.

`newName`: This param allows you to specify a new name for the file to be moved. This param may be empty, in which case the current leaf name will be used.For the `moveToNative` method, the newName must be in the native filesystem charset.

Result:

`NS_OK` if successful.

Remove

Tries to delete this file. The 'recursive' flag must be `PR_TRUE` to delete directories which are not empty. It will not resolve any symlinks.

Syntax:

```
nsresult Remove(PRBool recursive)
```

Parameters:

`recursive`: A boolean indicating whether or not to delete directories recursively.

Result:

`NS_OK` if successful.

Exists

Determines if this file exists.

Syntax:

```
nsresult Exists(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it exists.
`FALSE` otherwise.

IsWritable

Determines if this file is writable.

Syntax:

```
nsresult IsWritable(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is writable.
`FALSE` otherwise.

IsReadable

Determines if this file is readable.

Syntax:

```
nsresult IsReadable(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is readable.
`FALSE` otherwise.

IsExecutable

Determines if this file is executable.

Syntax:

```
nsresult IsExecutable(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is executable.
`FALSE` otherwise.

IsHidden

Determines if this file is hidden.

Syntax:

```
nsresult IsHidden(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is hidden.
`FALSE` otherwise.

IsDirectory

Determines if this file is a directory.

Syntax:

```
nsresult IsDirectory(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is a directory.
`FALSE` otherwise.

IsFile

Determines if this file is a file.

Syntax:

```
nsresult IsFile(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is a file.
`FALSE` otherwise.

IsSymlink

Determines if this file is a symbolic link.

Syntax:

```
nsresult IsSymlink(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is a symlink.
`FALSE` otherwise.

IsSpecial

Determines if this file is other than a regular file, a directory, or a symbolic link.

Syntax:

```
nsresult IsSpecial(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if it is not a file, a directory, or a symlink.

`FALSE` otherwise.

CreateUnique

Create a new file or directory in the file system. Any nodes that have not been created or resolved, will be. If this file already exists, this tries variations on the leaf name "suggestedName" until it finds one that does not already exist. If the search for nonexistent files takes too long (thousands of the variants already exist), it gives up and return `NS_ERROR_FILE_TOO_BIG`.

The type flag must be either `NORMAL_FILE_TYPE` or `DIRECTORY_TYPE`

Syntax:

```
nsresult CreateUnique(PRUint32 type, PRUint32
permissions)
```

Parameters:

`type`: The type of file system object to be made. The only two types at this time are file and directory which are defined below. If the type is unrecongnized, returns an error (`NS_ERROR_FILE_UNKNOWN_TYPE`).

`permissions`: Unix style octal permissions. This may be ignored on systems that do not need to do permissions.

Result:

`NS_OK` if successful.

Clone

Allocates and initializes an nsIFile object to the exact location of `this` nsIFile

Syntax:

```
nsresult Clone(nsIFile **_retval)
```

Parameters:

None.

Returns:

An `nsIFile` with which this object will be initialized.

Equals

Determines if `inFile` equals this.

Syntax:

```
Equals(nsIFile *inFile, PRBool *_retval)
```

Parameters:

`inFile`: The comparison object.

Returns:

`TRUE` if the files are equal.

`FALSE` otherwise.

Contains

Determines if `inFile` is a descendant of this file. If `recur` is true, it will also look in subdirectories.

Syntax:

```
nsresult Contains(nsIFile *inFile, PRBool recur, PRBool
*_retval)
```

Parameters:

`inFile`: The file to be evaluated.

`recur`: A boolean indicating whether subdirectories should be searched.

Returns:

`TRUE` if `infile` is a descendent of this file.
`FALSE` otherwise.

LeafName

Getters and setters for the leaf name of the file itself. `nativeLeafName` must be in the native filesystem charset.

```
nsresult GetLeafName(nsAString & aLeafName)
```

```
nsresult SetLeafName(const nsAString & aLeafName)
```

```
nsresult GetNativeLeafName(nsACString &
aNativeLeafName)
```

```
nsresult SetNativeLeafName(const nsACString &
aNativeLeafName)
```

Permissions

Getters and setters for the permssion of the file or link.

```
nsresult GetPermissions(PRUint32 *aPermissions)
```

```
nsresult SetPermissions(PRUint32 aPermissions)
```

```
nsresult GetPermissionsOfLink(PRUint32
*aPermissionsOfLink)
```

```
nsresult SetPermissionsOfLink(PRUint32
aPermissionsOfLink)
```

LastModificationTime

Gets and sets time of the last file modification. Times are stored as milliseconds from midnight (00:00:00), January 1, 1970 Greenwich Mean Time (GMT).

Syntax

```
nsresult GetLastModifiedTime(PRInt64
*aLastModifiedTime)
```

```
nsresult SetLastModifiedTime(PRInt64 aLastModifiedTime)
```

```
nsresult GetLastModifiedTimeOfLink(PRInt64
*aLastModifiedTimeOfLink)
```

```
nsresult SetLastModifiedTimeOfLink(PRInt64
aLastModifiedTimeOfLink)
```

FileSize

Getters and setters for the file size.

Warning: On the Mac, getting/setting the file size with `nsIFile` only deals with the size of the data fork. If you need to know the size of the combined data and resource forks use the `GetFileSizeWithResFork()` method defined on `nsILocalFileMac`.

Syntax

```
nsresult GetFileSize(PRInt64 *aFileSize)
```

```
nsresult SetFileSize(PRInt64 aFileSize)
```

```
nsresult GetFileSizeOfLink(PRInt64 *aFileSizeOfLink)
```

Target

Gets the target, ie, what the symlink points at. Gives an error (`NS_ERROR_FILE_INVALID_PATH`) if not a symlink. Note that the `ACString` attribute is returned in the native filesystem charset.

Warning:The native version of these strings are not guaranteed to be a usable path to pass to NSPR or the C stdlib. There are problems that affect platforms on which a path does not fully specify a file because two volumes can have the same name (e.g., XP_MAC). This is solved by holding "private", native data in the `nsIFile` implementation. This native data is lost when you convert to a string.

Note: NOT USE TO PASS TO NSPR OR STDLIB!

Syntax

```
nsresult GetTarget(nsAString & aTarget)

nsresult GetNativeTarget(nsACString & aNativeTarget)
```

Parent

Gets the parent of this file. Parent will be null when this file is at the top of the volume.

Syntax

```
nsresult GetParent(nsIFile * *aParent);
```

DirectoryEntries

Gets an enumeration of the elements in a directory. Each element in the enumerator is an nsIFile. If the current `nsIFile` does not specify a dirctory, returns an error `NS_ERROR_FILE_NOT_DIRECTORY`.

Syntax

```
nsresult GetDirectoryEntries(nsISimpleEnumerator *
*aDirectoryEntries)
```

Example

```
PRBool RecursiveDirectories(nsIFile* file)
{
    nsresult rv;
    nsCOMPtr<nsISimpleEnumerator> entries;
    rv = file->GetDirectoryEntries(getter_AddRefs(entries));
    if(NS_FAILED(rv) || !entries)
        return PR_FALSE;

    PRUint32 count = 0;
    PRBool hasMore;
    while(NS_SUCCEEDED(entries->HasMoreElements(&hasMore)) && hasMore)
    {
        nsCOMPtr<nsISupports> sup;
        entries->GetNext(getter_AddRefs(sup));
        if(!sup)
            return PR_FALSE;

        nsCOMPtr<nsIFile> file = do_QueryInterface(sup);
        if(!file)
            return PR_FALSE;

        nsEmbedCString name;
        if(NS_FAILED(file->GetNativeLeafName(name)))
            return PR_FALSE;

        PRBool isDir;
        printf("%s\n", name.get());
        rv = file->IsDirectory(&isDir);
        if (NS_FAILED(rv))
        {
            printf("IsDirectory Failed!!!\n");
            return PR_FALSE;
        }

        if (isDir == PR_TRUE)
        {
           RecursiveDirectories(file);
        }
    }
    return PR_TRUE;
}
```

nsIInputStream

This interface manages reading data in from an input stream. It is partially scriptable.

Methods:

- **close**
- **Available**
- **Read**
- **ReadSegments**
- **IsNonBlocking**

close

Closes the stream.

Syntax:

```
nsresult Close(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

Available

Gets number of bytes currently available in the stream.

Syntax:

```
nsresult Available(PRUint32 *_retval)
```

Parameters:

None.

Returns:

The number of bytes.

Read

Reads data from the stream.

Syntax:

```
nsresult Read(char * aBuf, PRUint32 aCount, PRUint32
*_retval)
```

Parameters:

`aBuf:` The buffer into which the data is to be read.
`aCount:` The maximum number of bytes to be read.

Returns:

The number of bytes read. Returns 0 if end of file has been reached.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if reading from the input stream would block the calling thread (non-blocking mode only).

Throws <other-error> on failure.

ReadSegments

Low-level read method that has access to the stream's underlying buffer. The writer function may be called multiple times for segmented buffers.

Syntax:

```
nsresult ReadSegments(nsWriteSegmentFun aWriter, void *
aClosure, PRUint32 aCount, PRUint32 *_retval)
```

Parameters:

`aWriter:` The "consumer" of the data to be read. The type is described below.
`aClosure:` Opaque parameter passed to writer.
`aCount:` The maximum number of bytes to be read.

```
typedef NS_CALLBACK(nsWriteSegmentFun)(nsIInputStream
*aInStream, void *aClosure, const char *aFromSegment,
PRUint32 aToOffset, PRUint32 aCount, PRUint32
*aWriteCount);
```

`aInStream:` The stream being read.

`aClosure:` Opaque parameter passed to `ReadSegments`.

`aFromSegment:` A pointer to memory owned by the input stream.

`aToOffset:` The amount already read (since `ReadSegments` was called).

`aCount:` The length of fromSegment.

`aWriteCount:` The number of bytes read.

Note: Implementers should return the following: `NS_OK` and `(*aWriteCount > 0)` if consumed some data; `NS_BASE_STREAM_WOULD_BLOCK` if not interested in consuming any data; <other-error> on failure. Errors are passed to the caller of ReadSegments, unless `aToOffset` is greater than zero.

Returning `NS_OK` and `(*aWriteCount = 0)` has undefined behavior.

Returns:

The number of bytes read. Returns 0 if end of file has been reached.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if reading from the input stream would block the calling thread (non-blocking mode only).

Throws <other-error> on failure.

Note: This method may be unimplemented if a stream has no underlying buffer (e.g., socket input stream).

IsNonBlocking

Returns `TRUE` if stream is non-blocking.

Syntax:

```
nsresult IsNonBlocking(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if stream is non-blocking.
`FALSE` otherwise.

nsILocalFile

This interface adds methods to `nsIFile` that are particular to a file that is accessible via the local file system. It follows the same string conventions as `nsIFile`.

Methods:

- **InitWithPath**
- **InitWithNativePath**
- **InitWithFile**
- **OpenNSPRFileDesc**
- **OpenANSIFileDesc**
- **Load**
- **appendRelativePath**
- **appendRelativeNativePath**
- **Reveal**
- **Launch**
- **GetRelativeDescriptor**
- **SetRelativeDescriptor**
- **FollowingLinks**
- **GetDiskSpaceAvailable**

InitWithPath

InitWithNativePath

Initializes the `nsILocalFile` object. Any internal state information will be reset.

Note: This function has a known bug on the macintosh and other operating systems which do not represent file locations as paths. If you do use this function, be very aware of this problem.

Syntax:

```
nsresult InitWithPath(const nsAString & filePath)
```

```
nsresult InitWithNativePath(const nsACString &
filePath)
```

Parameters:

> `filePath`: A string which specifies a full file path to a location. Relative paths will be treated as an error (`NS_ERROR_FILE_UNRECOGNIZED_PATH`). For `InitWithNativePath`, `filePath` must be in the native filesystem charset.

Result:

> `NS_OK` if successful.

InitWithFile

Initializes this object with another file.

Syntax:

```
nsresult InitWithFile(nsILocalFile *aFile)
```

Parameters:

> `aFile`: The file to which `this` becomes equivalent.

Result:

> `NS_OK` if successful.

OpenNSPRFileDesc

Opens the NSPR file descriptor.

Syntax:

```
nsresult OpenNSPRFileDesc(PRInt32 flags, PRInt32 mode,
PRFileDesc * *_retval)
```

Parameters:

`flags`: The appropriate flags.
`mode`: The appropriate mode.

Returns:

A pointer to the descriptor.

See NSPR's documentation regarding PRFileDesc at *http://www.mozilla.org/projects/nspr*.

OpenANSIFileDesc

Opens the ANSI file descriptor.

Syntax:

```
nsresult OpenANSIFileDesc(const char *mode, FILE *
*_retval)
```

Parameters:

`mode`: The appropriate mode.

Returns:

A pointer to the file.

Load

Loads `this` file (a library).

Syntax:

```
nsresult Load(PRLibrary * *_retval)
```

Parameters:

None.

Returns:

A pointer to the library.

See NSPR’s documentation regarding PRLibrary at *http://www/mozilla.org/projects/nspr.*

appendRelativePath

appendRelativeNativePath

Appends a relative path to the current path of the nsILocalFile object.

Syntax:

```
nsresult AppendRelativePath(const nsAString &
relativeFilePath)
```

```
nsresult AppendRelativeNativePath(const nsACString &
relativeFilePath)
```

Parameters:

`relativeFilePath`: The relativeFilePath. It is a native relative path. For security reasons, it cannot contain .. or start with a directory separator. For the `appendRelativeNativePath` method, the relativeFilePath must be in the native filesystem charset.

Result:

`NS_OK` if successful.

Reveal

Asks the operating system to open the folder which contains this file or folder. This routine only works on platforms which support the ability to open a folder.

Syntax:

```
nsresult Reveal(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

Launch

Asks the operating system to attempt to open the file. This really just simulates "double clicking" the file on your platform and thus only works on platforms which support this functionality.

Syntax:

```
nsresult Launch(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

GetRelativeDescriptor

Gets a relative file path in an opaque, XP format. It is therefore not a native path.

Note: The character set of the string returned from this function is undefined.

DO NOT TRY TO INTERPRET IT AS HUMAN READABLE TEXT!

Syntax:

```
nsresult GetPersistentDescriptor(nsACString &
aPersistentDescriptor)
```

Parameters:

`fromFile`: The file from which the descriptor is relative.

Returns:

The descriptor.

SetRelativeDescriptor

Initializes the file to the location relative to fromFile using a string returned by `getRelativeDescriptor`.

Syntax:

```
nsresult SetPersistentDescriptor(const nsACString &
aPersistentDescriptor)
```

Parameters:

`fromFile`: The file to which the descriptor is relative.
`relativeDesc`: The relative descriptor obtained from getRelativeDescriptor.

Result:

`NS_OK` if successful.

FollowingLinks

Getter and Setter for FollowingLinks attribute. Gets and sets whether the `nsLocalFile` will auto resolve symbolic links. By default, this value will be false on all non unix systems. On Unix, this attribute is effectively a no-op.

Be aware that changing this attribute from true to false after the nsILocalFile has been initialized may lead to errors. This could happen if there were resolved symlink in the initialized path. For example if you had /a/b/c where |b| was a symlink, and you change this attribute to false, the next usage would mostlikely fail.

Syntax:

```
nsresult GetFollowLinks(PRBool *aFollowLinks)
```

```
nsresult SetFollowLinks(PRBool aFollowLinks)
```

Result:

`NS_OK` if successful.

GetDiskSpaceAvailable

Return the available disk space on the volume or drive reference by the `nsILocalFile`.

Syntax:

```
nsresult GetDiskSpaceAvailable(PRInt64
*aDiskSpaceAvailable)
```

Result:

`NS_OK` if successful.

nsIOutputStream

This interface manages writing data to an output stream. It is partially scriptable.

Methods:

- **Close**
- **Flush**
- **Write**
- **WriteFrom**
- **WriteSegments**
- **IsNonBlocking**

Close

Closes the stream. Forces the output stream to flush any buffered data.

Syntax:

```
nsresult Close(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if unable to flush without blocking the calling thread (non-blocking mode only)

Flush

Flushes the stream.

Syntax:

```
nsresult Flush(void)
```

Parameters:

None.

Result:

`NS_OK` if successful.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if unable to flush without blocking the calling thread (non-blocking mode only).

Write

Writes data into the stream from an input stream.

Syntax:

```
nsresult Write(const char *aBuf, PRUint32 aCount,
PRUint32 *_retval)
```

Parameters:

`aBuf:` The buffer containing the data to be written.

`aCount:` The maximum number of bytes to be written.

Returns:

The number of bytes written.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if unable to flush without blocking the calling thread (non-blocking mode only)

Throws <other-error> on failure.

WriteFrom

Writes data into the stream from an input stream.

Syntax:

```
nsresult WriteSegments(nsReadSegmentFun aReader, void *
aClosure, PRUint32 aCount, PRUint32 *_retval)
```

Parameters:

`aFromStream:`The stream containing the data to be written.

`aCount:` The maximum number of bytes to be written.

Returns:

The number of bytes written.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if unable to flush without blocking the calling thread (non-blocking mode only)

Throws <other-error> on failure

Note: This method is defined by this interface in order to allow the output stream to efficiently copy the data from the input stream into its internal buffer (if any). If this method was provided as an external facility, a separate `char*` buffer would need to be used in order to call the output stream's other `Write` method.

WriteSegments

Low-level write method that has access to the stream’s underlying buffer. The reader function may be called multiple times for segmented buffers.

Syntax:

```
nsresult WriteSegments(nsReadSegmentFun aReader, void *
aClosure, PRUint32 aCount, PRUint32 *_retval)
```

Parameters:

`aReader:` The "provider" of the data to be written. The type is described below.

`aClosure:` Opaque parameter passed to reader.

`aCount:` The maximum number of bytes to be written.

```
typedef NS_CALLBACK(nsReadSegmentFun)(nsIOutputStream
*aOutStream, void *aClosure, char *aToSegment, PRUint32
aFromOffset, PRUint32 aCount, PRUint32 *aReadCount)
```

`aOutStream:` The stream being written to.

`aClosure:` Opaque parameter passed to WriteSegments.

`aToSegment:` A pointer to memory owned by the output stream.

`aFromOffset:` The amount already written (since WriteSegments was called).

`aCount:` The length of toSegment.

`aReadCount:` The number of bytes written.

Note: Implementers should return the following: `NS_OK` and `(*aReadCount > 0)` if successfully provided some data; `NS_OK` and `(*aReadCount = 0);` or `NS_BASE_STREAM_WOULD_BLOCK` if not interested in providing any data;<other-error> on failure

Errors are passed to the caller of `WriteSegments`, unless `aFromOffset` is greater than zero.

Returns:

The number of bytes written.

Throws `NS_BASE_STREAM_WOULD_BLOCK` if writing to the output stream would block the calling thread (non-blocking mode only).

Throws <other-error> on failure.

Note: this function may be unimplemented if a stream has no underlying buffer (e.g., socket output stream).

IsNonBlocking

Returns `TRUE` if stream is non-blocking.

Syntax:

```
nsresult IsNonBlocking(PRBool *_retval)
```

Parameters:

None.

Returns:

`TRUE` if stream is non-blocking.

`FALSE` otherwise.

XPCOM Startup/Shutdown

These C++ functions serve to initialize and terminate XPCOM. (In an embedding situation, this is usually taken care of by embedding initialization.) Also included are a number of global functions that provide access to the main XPCOM components.

Note: These are C++ functions only, and are therefore not scriptable.

- `NS_InitXPCOM2`
- `NS_ShutdownXPCOM`
- `NS_GetServiceManager`

- `NS_GetComponentManager`
- `NS_GetComponentRegistrar`
- `NS_GetMemoryManager`
- `NS_NewLocalFile`
- `NS_NewNativeLocalFile`

NS_InitXPCOM2

Initializes XPCOM. This function must be called by the application before using XPCOM. Components should not call this function. The one exception is that you may call `NS_NewLocalFile` to create an `nsIFile` object to supply as the bin directory path in this call.

Syntax:

```
nsresult NS_InitXPCOM2(nsIServiceManager**
result,nsIFile*
binDirectory,nsIDirectoryServiceProvider*
appFileLocationProvider)
```

Parameters:

`result:` The service manager. You may pass null.

`binDirectory:` The directory containing the component registry and runtime libraries. You can use `nsnull` to use the working directory.

`appFileLocProvider:` The object to be used by Gecko that specifies to Gecko where to find profiles, the component registry preferences and so on. You can use `nsnull` for the default behaviour.

nsresult

`NS_OK` if successful.

Other error codes indicate failure during initialization.

See also: see `NS_NewLocalFile`, `nsILocalFile`, and `nsIDirectoryServiceProvider`.

NS_ShutdownXPCOM

Shuts down XPCOM. This function must be called by the application when you are finished using XPCOM.

Syntax:

```
nsresult NS_ShutdownXPCOM(nsIServiceManager* servMgr)
```

Parameters:

`servMgr:` The service manager which was returned by `NS_InitXPCOM2`. This will release the service manager. You may pass null.

Result:

`NS_OK` if successful.
Other error codes indicate failure.

NS_GetServiceManager

Accesses the Service Manager.

Syntax:

```
nsresult NS_GetServiceManager(nsIServiceManager**
result)
```

Parameters:

`result:` An interface pointer to the service manager.

Result:

`NS_OK` if successful.

NS_GetComponentManager

Accesses the Component Manager.

Syntax:

```
NS_COM nsresult
NS_GetComponentManager(nsIComponentManager** result)
```

Parameters:

`result:` `An` interface pointer to the component manager.

Result:

`NS_OK` if successful.

NS_GetComponentRegistrar

Accesses the Component Registration Manager.

Syntax:

```
NS_COM nsresult
NS_GetComponentRegistrar(nsIComponentRegistrar**
result)
```

Parameters:

`result:` An interface pointer to the component registration manager.

Result:

`NS_OK` if successful.

NS_GetMemoryManager

Accesses the memory manager.

Syntax:

```
NS_COM nsresult NS_GetMemoryManager(nsIMemory** result)
```

Parameters:

`result:` An interface pointer to the memory manager

Result:

`NS_OK` if successful.

NS_NewLocalFile

NS_NewNativeLocalFile

Creates a new instanstance of an object implementating the nsIFile and `nsILocalFile` interfaces. (On the Macintosh platform, this object also implments the `nsILocalFileMac` interface). If using the native form of this call, `path` must be in the native filesystem charset.

Syntax:

```
nsresult

NS_NewLocalFile(const nsAString &path,

                PRBool followLinks,

                nsILocalFile* *result);

nsresult

NS_NewNativeLocalFile(const nsACString &path,

                      PRBool followLinks,

                      nsILocalFile* *result);
```

Parameters:

`path:` A string which specifies a full file path to a location. Relative paths will be treated as an error (`NS_ERROR_FILE_UNRECOGNIZED_PATH`). For `initWithNativePath`, the `filePath` must be in the native filesystem charset.

`followLinks:` Sets whether the `nsLocalFile` will auto resolve symbolic links. On Unix, this parameter is ignored

`result:` [out] An interface pointer to the new `nsILocalFile`.

Result:

`NS_OK` if successful.

Appendix C: Resources

This last section of the book provides a list of resources referred to in the tutorial and other links that may be useful to the Gecko developer. The resources are divided into the following categories:

- **WebLock Resources**
- **Gecko Resources**
- **XPCOM Resources**
- **General Development Resources**

WebLock Resources

- **WebLock** installer and information: *http://www.brownhen.com/weblock*
- The SDK download:

  Linux: *http://ftp.mozilla.org/pub/mozilla/releases/mozilla1.4a/gecko-sdk-i686-pc-linux-gnu-1.4a.tar.gz*

  Windows: *http://ftp.mozilla.org/pub/mozilla/releases/mozilla1.4a/gecko-sdkwin32-1.4a.zip*

- Other Mozilla downloads: *http://ftp.mozilla.org/pub/mozilla/releases/*.

Gecko Resources

- Guide to the Mozilla string classes: *www.mozilla.org/projects/xpcom/string-guide.html*
- The Gecko networking library ("necko"): *www.mozilla.org/projects/netlib*
- The Netscape Portable Runtime Environment: *www.mozilla.org/projects/nspr*
- Embedding Mozilla: *www.mozilla.org/projects/embedding*
- A list of module owners: *www.mozilla.org/owners.html*
- XPInstall: *www.mozilla.org/projects/xpinstall*
- XUL: *www.mozilla.org/projects/xul*

XPCOM Resources

- The XPCOM project page: *www.mozilla.org/projects/xpcom*
- XULPlanet's online XPCOM reference: *http://www.xulplanet.com/references/xpcomref/*
- Information on XPConnect and scriptable components: *http://www.mozilla.org/scriptable*
- The Smart Pointer Guide: *http://www.mozilla.org/projects/xpcom/nsCOMPtr/*.
- XPIDL Reference: *http://www.mozilla.org/scriptable/xpidl/*
-

General Development Resources

- The World Wide Web Consoritum: *www.w3.org*
- URL specification at the W3: *www.w3.org/TR/REC-html40/intro/intro.html*
- Make: *http://www.gnu.org/manual/make/*

Index