Slice for a Simple File System

For this manual, we use a file system application to illustrate various aspects of Ice. Throughout, we progressively improve and modify the application such that it evolves into an application that is realistic and illustrates the architectural and coding aspects of Ice. This allows us to explore the capabilities of the platform to a realistic degree of complexity without overwhelming you with an inordinate amount of detail early on.

In this section:

- #File System Application outlines the file system functionality
- #Slice Definitions for the File System develops the data types and interfaces that are required for the file system
- #Complete Definition presents the complete Slice definition for the application.

File System Application

Our file system application implements a simple hierarchical file system, similar to the file systems we find in Windows or Unix. To keep code examples to manageable size, we ignore many aspects of a real file system, such as ownership, permissions, symbolic links, and a number of other features. However, we build enough functionality to illustrate how you could implement a fully-featured file system, and we pay attention to things such as performance and scalability. In this way, we can create an application that presents us with real-world complexity without getting buried in large amounts of code.

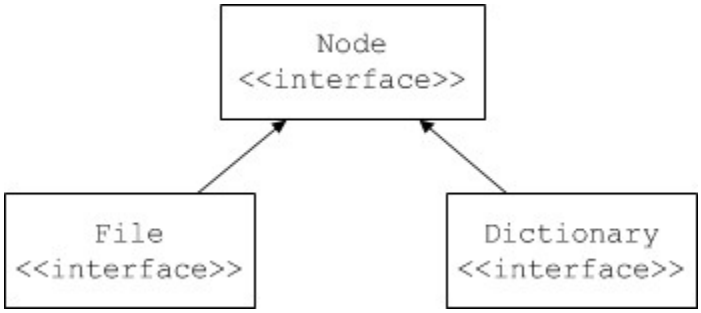
Our file system consists of directories and files. Directories are containers that can contain either directories or files, meaning that the file system is hierarchical. A dedicated directory is at the root of the file system. Each directory and file has a name. Files and directories with a common parent directory must have different names (but files and directories with different parent directories can have the same name). In other words, directories form a naming scope, and entries with a single directory must have unique names. Directories allow you to list their contents.

For now, we do not have a concept of pathnames, or the creation and destruction of files and directories. Instead, the server provides a fixed number of directories and files. (We will address the creation and destruction of files and directories in Object Life Cycle.)

Files can be read and written but, for now, reading and writing always replace the entire contents of a file; it is impossible to read or write only parts of a file.

Slice Definitions for the File System

Given the very simple requirements we just outlined, we can start designing interfaces for the system. Files and directories have something in common: they have a name and both files and directories can be contained in directories. This suggests a design that uses a base type that provides the common functionality, and derived types that provide the functionality specific to directories and files, as shown:



Inheritance Diagram of the File System.

The Slice definitions for this look as follows:

Slice

```
interface Node {
    // ...
};

interface File extends Node {
    // ...
};

interface Directory extends Node {
    // ...
};
```

Next, we need to think about what operations should be provided by each interface. Seeing that directories and files have names, we can add an operation to obtain the name of a directory or file to the `Node` base interface:

Slice

```
interface Node {
    idempotent string name();
};
```

The `File` interface provides operations to read and write a file. For simplicity, we limit ourselves to text files and we assume that `read` operations never fail and that only `write` operations can encounter error conditions. This leads to the following definitions:

Slice

```
exception GenericError {
    string reason;
};

sequence<string> Lines;

interface File extends Node {
    idempotent Lines read();
    idempotent void write (Lines text) throws GenericError;
};
```

Note that `read` and `write` are marked idempotent because either operation can safely be invoked with the same parameter value twice in a row: the net result of doing so is the same has having (successfully) called the operation only once.

The `write` operation can raise an exception of type `GenericError`. The exception contains a single `reason` data member, of type `string`. If a `write` operation fails for some reason (such as running out of file system space), the operation throws a `GenericError` exception, with an explanation of the cause of the failure provided in the `reason` data member.

Directories provide an operation to list their contents. Because directories can contain both directories and files, we take advantage of the polymorphism provided by the `Node` base interface:

Slice

```
sequence<Node*> NodeSeq;

interface Directory extends Node {
    idempotent NodeSeq list();
};
```

The `NodeSeq` sequence contains elements of type `Node*`. Because `Node` is a base interface of both `Directory` and `File`, the `NodeSeq` sequence can contain proxies of either type. (Obviously, the receiver of a `NodeSeq` must down-cast each element to either `File` or `Directory` in order to get at the operations provided by the derived interfaces; only the `name` operation in the `Node` base interface can be invoked directly, without doing a down-cast first. Note that, because the elements of `NodeSeq` are of type `Node*` (not `Node`), we are using pass-by-reference semantics: the values returned by the `list` operation are proxies that each point to a remote node on the server.

These definitions are sufficient to build a simple (but functional) file system. Obviously, there are still some unanswered questions, such as how a client obtains the proxy for the root directory. We will address these questions in the relevant implementation chapter.

Complete Definition

We wrap our definitions in a module, resulting in the final definition as follows:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        idempotent NodeSeq list();
    };
};
```

See Also

- Object Life Cycle