Services

Introduction

The Yellow Box services facility allows an application to offer its functionality to other applications, without requiring the other applications to know in advance what's offered. A service-providing application advertises an operation that it can perform on a particular type of data—for example, encrypting text, performing optical character recognition on a bitmapped image, or providing text such as a message of the day (with no input data). Any application that uses the services facility then automatically has access to that functionality through its Services menu, or through certain other mechanisms. It doesn't need to know what the operations are in advance; it merely indicates what types of data it has, and the Services menu makes available the operations that apply to those types. The services facility thus gives applications an open-ended means of extending each others' functionality.

This document describes the four available types of service: standard services, which the user chooses from the Services menu; filter services, which the developer invokes through the NSPasteboard class; print filter services, which the user chooses when saving a printout as a PostScript file; and spell checker services, which the user chooses from the standard spelling checker panel. The first section, "Standard Services," describes the general structure of all the services and the details of standard services. The second section, "Variations on Standard Services," describes ways that the other three types of service differ from standard services.

Standard Services

In general terms, the standard services facility works as though the user copies data from one application and pastes it into another, modifies the data, then copies the result and pastes it back into the original application. The standard services facility does in fact use the pasteboard to transfer data, automatically copying the selection from the service requestor and pasting the altered data back—though the data transfer doesn't have to be two-way, as the examples in the introduction indicate. You should be familiar with the Application Kit's NSPasteboard class before working with the standard services facility.

This section describes how to provide a service in your application, and how to make sure your application can also request appropriate services in any situation. "Providing a Standard Service" covers everything you need to know as the implementor of a service. "Using Services" shows you what you need to make your custom classes work as requestors of services.

Providing a Standard Service

Suppose you’re working on a program to read USENET news, and have an object with a method to encrypt and decrypt articles, such as the one below. News articles containing offensive material are often encrypted with this algorithm, called “rot13,” in which letters are shifted halfway through the alphabet.

```
- (NSString *)rotateLettersInString:(NSString *)aString
{
    NSString *newString;
    unsigned length;
    unichar *buf;
    unsigned i;

    length = [aString length];
    buf = malloc( (length + 1) * sizeof(unichar) );
    [aString getCharacters:buf];
    buf[length] = (unichar)0;  // not really needed....

    for (i = 0; i < length; i++) {
        if (buf[i] >= (unichar)'a' && buf[i] <= (unichar) 'z') {
            buf[i] += 13;
            if (buf[i] > 'z') buf[i] -= 26;
        }
         else if (buf[i] >= (unichar)'A' &&
               buf[i] <= (unichar) 'Z') {
            buf[i] += 13;
            if (buf[i] > 'Z') buf[i] -= 26;
        }
    }

    newString = [NSString stringWithCharacters:buf length:length];
    free(buf);

    return newString;
}
```

Since this feature is generally useful as a simple encryption scheme, it can be exported to other applications. To offer this functionality as a service, write a method such as this:

```
- (void)simpleEncrypt:(NSPasteboard *)pboard
    userData:(NSString *)data
    error:(NSString **)error
{
    NSString *pboardString;
    NSString *newString;
    NSArray *types;

    types = [pboard types];
```

```
        if (![types containsObject:NSStringPboardType]) {
            *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                       @"pboard couldn't give string.");
            return;
         }

        pboardString = [pboard stringForType:NSStringPboardType];
        if (!pboardString) {
            *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                       @"pboard couldn't give string.");
            return;
        }

        newString = [self rotateLettersInString:pboardString];

        if (!newString) {
            *error = NSLocalizedString(@"Error: couldn't encrypt text.",
                       @"self couldn't rotate letters.");
            return;
        }

        types = [NSArray arrayWithObject:NSStringPboardType];
        [pboard declareTypes:types owner:nil];
        [pboard setString:newString forType:NSStringPboardType];

        return;
    }
```

A method for providing a standard service is of the form *serviceName*:**userData:error:** and takes arguments as shown in the example. The method itself takes data from the pasteboard as needed, operates on it, and writes any results back to the pasteboard. In case of an error, the method simply sets the pointer given by the *error* argument to a non-**nil** NSString and returns. The *userData* argument isn't used here; see "Entries in a Service Specification" and "Add-on Services" for some suggestions on how to use it.

Making a Service Available

Now you have an object with methods that allow it to perform a service for another application. There are two things remaining to do: register the object at run time so the services facility knows which object to have perform the service, and advertise the service to the services facility. You create and register your object in the **applicationDidFinishLaunching:** application delegate method (or equivalent) with NSApplication's **setServicesProvider:** method. If your object is called **encryptor** you create and register it with this code fragment:

```
EncryptoClass *encryptor;

encryptor = [[EncryptoClass alloc] init];
[NSApp setServicesProvider:encryptor];
```

You can register only one service provider per application. If you have more than one service to provide, a single object must be able to provide all of the services.

In order for the system to know that your application provides a service, you must advertise that fact. You do this by adding an entry to your application project's **CustomInfo.plist** file, which is incorporated into the application's **Info.plist** file when you build your project. The entry you add is called the *service specification*. In our example, the service specification looks like this:

```
{
    NSServices = (
      { NSPortName = NewsReader;
        NSMessage = simpleEncrypt;
        NSSendTypes = (NSStringPboardType);
        NSReturnTypes = (NSStringPboardType);
        NSMenuItem = {
            default = "Encrypt Text";
            English = "Encrypt Text";
            French = "Encoder le texte";
            German = "Text verschlüsseln";
        };
        NSKeyEquivalent = {
            default = E;
            German = S;
        };
      }
    );
}
```

Note: CustomInfo.plist should be saved as Unicode, not NEXSTTEP encoding.

The meaning of each of the subfields is explained further in the section below, "Entries in a Service Specification."

Note: If you've just built an application with a service and you want to test the service, log out and log back in again. The application must be in one of the standard directories: **~/Apps**, **/NextApps**, or **/LocalApps**.

Entries in a Service Specification

This template shows all possible fields in a standard service specification:

NSServices = (
{ **NSMessage =** *messageName***;**
NSPortName = *programName***;**
NSSendTypes = (*type1* [, *type2*] ... **);**
NSReturnTypes = (*type1* [, *type2*] ... **);**
NSMenuItem = { default = *item***;** [*language* **=** *item***;**] **};**
NSKeyEquivalent = { default = *character***;** [*language* **=** *character***;**] **};**
NSUserData = *string***;**

```
        NSTimeout = milliseconds;
        NSHost = hostName;
        NSExecutable = pathname;
    }
    [, { another service entry } ] ...
);
```

Filter, print filter, and spell checker services differ slightly. Their service specifications are described in “Variations on Standard Services.”

NSMessage indicates the name of the Objective-C method to invoke. Its value is the first part of the method name, which follows the form *messageName***:userData:error:**. This is a required entry.

NSPortName is the name of the port the application should use to listen for service requests. Its value depends on how you registered the service provider. If you used the NSApplication method **setServicesProvider:**, **NSPortName** is the application name. If you used the **NSRegisterServicesProvider()** function (which should only be used for filter services), **NSPortName** is the value passed to that function for its *name* argument. See “Filter Services” for more information on **NSRegisterServicesProvider()**. This is a required entry.

NSSendTypes and **NSReturnTypes** are arrays of names for data types, such as **NSStringPboardType**. Send types are the types sent from the service requestor; return types are the types returned to the service requestor. See the NSPasteboard class specification for a list of standard data types. A service provider must specify one or both of these entries.

NSMenuItem and **NSKeyEquivalent** indicate the text of the Services menu item and its key equivalent (if any). Both of these entries take the form of dictionaries, with language names as keys and the text as values. In addition to actual language names, you can define a value for the key **default**, which is used when no languages in the user’s preferences match the languages named in the service specification. The text of a menu item can indicate a single submenu with a slash; for example, “Mail/Send Selection” appears in the Services menu as a submenu named “Mail” with an item named “Send Selection”. **NSMenuItem** is required, but **NSKeyEquivalent** is optional.

NSUserData is a string containing a value of your choice. You can use this string to control the behavior of your service method; this entry is useful for applications that provide open-ended services (see “Add-on Services”). **NSUserData** is an optional entry.

NSTimeout is a string indicating the number of milliseconds the Services facility should wait for a response from the service provider when a response is required. If this time is exceeded, the services facility opens an attention panel informing the user that an error has occurred. This is an optional entry. If you don't specify this entry, the timeout value is 3000 milliseconds (30 seconds).

NSHost is a string containing the name of a host on the network. The executable is launched on this host instead of on the host of the application requesting the service. This is an optional entry.

NSExecutable is the path of the application that performs this service. This can either be a full or relative path. If it is a relative path, the application must be located in the same bundle as this service declaration. This entry is most useful for filter services. This entry is optional.

Add-on Services

You typically define services when you create your application and advertise them in the **Info.plist** file of the application's bundle. The services facility also allows you to advertise services outside of the application bundle, enabling you to create "add-on" services after the fact. This is where the **NSUserData** entry becomes truly useful: You can define a single message in your application that performs actions based on the user data provided, such as running the user data string as a UNIX command (which the Terminal application does) or treating it as a special argument in addition to the selected data that gets sent through the pasteboard.

To define an add-on service, you create a bundle with a **.service** extension that contains an **Info.plist** file, which in turn contains the add-on service specification. You then put this bundle into a **Services** directory in the library search path (**~/Library**, **/LocalLibrary**, **/NextLibrary**). The services facility scans these directories when the user logs in and takes note of which services are defined; you can force this scanning by running the **make_services** UNIX command. If your application creates a service at run time and needs it to be available immediately, it calls this function to force scanning:

```
void NSUpdateDynamicServices(void)
```

Using Services

If you add a Services menu to your application in Interface Builder, there's nothing else you need to do for your application to work with the standard services facility; your application automatically has access to all appropriate services provided by other applications. If you need to construct menus programmatically or if you subclass NSView or NSWindow (or any other subclass of NSResponder), however, you need to do a little work to tie things into the standard services facility. Setting a Services menu programmatically is straightforward. You simply designate the NSMenu that you want as your Services menu with NSApplication's **setServicesMenu:** method. Tying custom NSViews or NSWindows into the standard services facility falls into three steps, in which you invoke or implement these methods:

registerServicesMenuSendTypes:returnTypes:
validRequestorForSendType:returnType:
writeSelectionToPasteboard:types:
readSelectionFromPasteboard:

The following sections cover each of these methods. A final section, “Invoking a Standard Service Programmatically,” shows how to invoke a standard service in your code.

Registering User-Interface Objects for Standard Services

The Services menu doesn’t contain *every* standard service offered by other applications. For example, in a text editor a service to invert a bitmapped image is of no use and shouldn’t be offered. Which services appear in the Services menu is determined by the data types that the objects in the application—specifically the NSResponder objects—can send and receive through the pasteboard.

An NSResponder registers these data types using NSApplication’s **registerServicesMenuSendTypes:returnTypes:** method. Application Kit objects already do this, but your custom NSResponder subclass must do this in its **initialize** class method. All types used by instances of the class must be registered, even if they’re not always available; Services menu items are enabled and disabled dynamically based on what’s available at the moment, as described in “Enabling Services Menu Items Based on the Selection”.

An object doesn’t have to register the same types for both sending and receiving. Suppose you’re writing a rich text editor that can send unformatted and rich text, but can only receive unformatted text. Here’s a portion of the initialization method for the text-editor NSView subclass:

```
+ (void)initialize
{
    static BOOL initialized = NO;

    /* Make sure code only gets executed once. */
    if (initialized == YES) return;
    initialized = YES;

    sendTypes = [NSArray arrayWithObjects:NSStringPboardType,
        NSRTFPboardType, nil];
    returnTypes = [NSArray arrayWithObjects:NSStringPboardType,
        nil];
    [NSApp registerServicesMenuSendTypes:sendTypes
        returnTypes:returnTypes];

    return;
}
```

Your NSResponder object can register any pasteboard data type, public or proprietary, common or rare. If it handles the public and common types, of course, it will have access to more services. See the NSPasteboard class specification for a list of standard pasteboard data types.

Enabling Services Menu Items Based on the Selection

While your application is running, various types of data can be selected and available for transfer on the pasteboard. If a service doesn't apply to the type of the selected data, its menu item needs to be disabled. To check whether a service applies, the application object sends **validRequestorForSendType:returnType:** messages to objects in the responder chain to see whether they have data of the type used by that service. While the Services menu is visible, this method is invoked frequently—typically many times per event—to ensure that the menu items for all service providers are properly enabled: It's sent for each service and possibly for many objects in the responder chain. Because this method is invoked so frequently, it must be fast so that event handling doesn't fall behind the user's actions.

The following example shows how this method can be implemented for an object that handles unformatted text:

```
- (id)validRequestorForSendType:(NSString *)sendType
    returnType:(NSString *)returnType;
{
    if ( (!sendType || [sendType isEqual:NSStringPboardType]) &&
        (!returnType || [returnType isEqual:NSStringPboardType]) ) {
        if ( ([self selection] || !sendType) &&
             ([self isEditable] || !returnType) ) {
            return self;
        }
    }

    return [super validRequestorForSendType:sendType
        returnType:returnType];
}
```

This implementation checks both the types indicated and the state of the object. The object is a valid requestor if the send and return types are unformatted text or simply aren't specified, and if the object has a selection and is editable (when send and return types are given). If this object can't handle the service request in its current state, it invokes its superclass' implementation.

validRequestorForSendType:returnType: is sent along an abridged responder chain, comprising only the responder chain for the key window and the application object. The main window is excluded.

Sending and Receiving Data

When the user chooses a Services menu command, the responder chain is checked with **validRequestorForSendType:returnType:** and the first object that returns a value other than **nil** is called upon to handle the service request by providing data (if any is required) with a **writeSelectionToPasteboard:types:** message. You can implement this method to provide the data immediately or to provide the data only when it's actually requested. Here's an implementation for an object that writes unformatted text immediately:

```
- (BOOL)writeSelectionToPasteboard:(NSPasteboard *)pboard
    types:(NSArray *)types
{
    NSArray *typesDeclared;

    if ([types containsObject:NSStringPboardType] == NO) {
        return NO;
    }

    typesDeclared = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:typesDeclared owner:nil];
    return [pboard setString:[self selection]
        forType:NSStringPboardType];
}
```

This method returns **YES** if it successfully writes or declares any data and **NO** if it fails. If you want to provide the data only on demand—which makes sense for large amounts—you have to declare an object as the owner for the data and then make sure that object responds to **pasteboard:provideDataForType:** (as described in the NSPasteboard class specification). In such a case, the two methods look like this:

```
- (BOOL)writeSelectionToPasteboard:(NSPasteboard *)pboard
    types:(NSArray *)types
{
    NSArray *typesDeclared;

    if ([types containsObject:NSStringPboardType] == NO) {
        return NO;
    }

    typesDeclared = [NSArray arrayWithObject:NSStringPboardType];
    [pboard declareTypes:typesDeclared owner:self];
    return YES;
}

- (void)pasteboard:(NSPasteboard *)pboard
    provideDataForType:(NSString *)type
{
    [pboard setString:[self selection] forType:NSStringPboardType];
    return;
}
```

You can even write some types in **writeSelectionToPasteboard:types:** and offer the rest on demand only via **pasteboard:provideDataForType:**. Remember that the owner of a pasteboard must exist when the data is finally requested. To be safe, you should make sure the owner is an object that will never be deallocated.

Once the service requestor writes data to the pasteboard, it waits for a response as the service provider is invoked to perform the operation; if the service doesn't return data, of course, the requesting application simply continues running and none of the following applies. The service provider reads the data from the pasteboard, works on it, and then returns the result. At this point the service requestor is sent a **readSelectionFromPasteboard:** message telling it to replace the selection with whatever data came back. Our simple text object can implement this method as follows:

```
- (BOOL)readSelectionFromPasteboard:(NSPasteboard *)pboard;
{
    NSArray *types;
    unsigned index;
    NSString *theText;

    types = [pboard types];
    index = [types indexOfObject:NSStringPboardType];
    if ([types containsObject:NSStringPBoardType] == NO) {
        return NO;
    }
    theText = [pboard stringForType:NSStringPboardType];
    [self replaceSelectionWithString:theText];
    return YES;
}
```

This method returns **YES** if it successfully reads the data from the pasteboard, **NO** otherwise.

Invoking a Standard Service Programmatically

Though the user typically invokes a standard service by choosing an item in the Services menu, you can invoke it in code using this function:

BOOL **NSPerformService(**NSString **serviceItem*, NSPasteboard **pboard***)**

This function returns **YES** if the service is successfully performed, **NO** otherwise. *serviceItem* is the name of a Services menu item (in any language). It must be the full name of the service, including the submenu and slash; for example, "Mail/Selection". *pboard* contains the data to be used for the service, and when the function returns contains the data resulting from the service. You can then do with the data what you wish.

Variations on Standard Services

The three other types of services—filter, print filter, and spell checker—all share the use of a service specification, but they're each implemented in different ways. The following sections describe how the service specification for each type of service differs from that for a standard service, and how you take advantage of that type of service.

Filter Services

The NSPasteboard class automatically uses a filter service when you invoke a method for filtering data, such as:

+ (NSArray *)**typesFilterableTo:**(NSString *)*type*
+ (NSPasteboard *)**pasteboardByFilteringFile:**(NSString *)*filename*
+ (NSPasteboard *)**pasteboardByFilteringData:**(NSData *)data
  ofType:(NSString *)*type*
+ (NSPasteboard *)**pasteboardByFilteringTypesInPasteboard:**
  (NSPasteboard *)*pboard*

Because filter services commonly translate data from unknown file formats into known formats, you need a way of dynamically specifying pasteboard types. The filter services and pasteboard facilities define types based on file extensions with these functions:

NSString ***NSCreateFilenamePboardType(**NSString **fileExtension***)**
NSString ***NSCreateFileContentsPboardType(**NSString **fileExtension***)**
NSString ***NSGetFileType(**NSString **pboardType***)**
NSArray ***NSGetFileTypes(**NSArray **pboardTypes***)**

The *fileExtension* argument is a file extension, minus the period (for example, "eps" or "tiff"). You create pasteboard type strings with the first two functions, and get file types (extensions) from pasteboard type strings with the second two functions. In a service specification (in the **CustomInfo.plist** file), you can indicate a file type based on the extension as **NSTypedFilenamesPboardType:***fileExtension* and a file contents type as **NSTypedFileContentsPboardType:***fileExtension*; for example:

```
NSSendTypes = (NSTypedFilenamesPboardType:tiff);
NSSendTypes = (NSTypedFileContentsPboardType:tiff);
```

You implement a filter service exactly like a standard service, with a *filterName***:userData:error:** method that accepts a pasteboard containing a file path, converts the contents of the file to the requested type or types, and returns the converted data on the pasteboard. There are two major differences between filter services and standard services. The first major difference is in the way you register the service provider. With filter services, you typically don't have an NSApplication object to register the service provider with. Instead, you use the function **NSRegisterServicesProvider()**. This function's declaration is:

(void)**NSRegisterServicesProvider(**id *provider*, NSString **name***)**

provider is the object that provides the services, and *name* is the same value you specify for the **NSPortName** entry in the services specification. After making this function call, the filter service must enter the run loop in order to respond to service requests as shown:

```
while(1) {
    NS_DURING
        [[NSRunLoop currentRunLoop] run];
    NS_HANDLER
        NSLog(@"Received exception: %@", localException);
    NS_ENDHANDLER
}
```

The second major difference is in the service specification: Instead of an **NSMessage** entry you define an **NSFilter** entry with *filterName* as the value; you must define both send and return types; and the **NSMenuItem** and **NSKeyEquivalent** entries are ignored.

A filter service can use data-transfer mechanisms other than the pasteboard, indicated by an optional entry in the filter service specification. The key is **NSInputMechanism**, and it can have a value of **NSUnixStdio**, **NSMapFile**, or **NSIdentity.** If you specify an input mechanism, the value for the **NSFilter** entry is ignored (though it's still required).

NSUnixStdio allows you to turn nearly any UNIX command-line program into a filter service. Instead of sending an Objective-C message to an object in your filter service program, the services facility simply runs the executable specified in the service specification with the contents of the pasteboard as the argument (which must be of **NSFilenamesPboardType** or **NSTypedFilenamesPboardType**). If there is more than one filename on the pasteboard, only the first is used. The output of the filter program (on **stdout**) is captured by the services facility and put on a pasteboard for use by the requestor of the filter. Note that the UNIX program must be relaunched every time the service is invoked; if you're creating a filter service from scratch it's more efficient to package it as an application that can remain running. Here's a sample service specification for a UNIX program that converts GIF images to TIFF:

```
{
    NSServices = (
      { NSFilter = "";
        NSPortName = gif2tiff;
        NSInputMechanism = NSUnixStdio;
        NSSendTypes = (NSTypedFilenamesPboardType:gif);
        NSReturnTypes = (NSTIFFPboardType);
      }
    );
}
```

NSMapFile defines an “empty” service for data in files, used when you invoke NSPasteboard’s **pasteboardByFilteringFile:** class method. Its value must be an **NSFilenamesPboardType** or an **NSTypedFilenamesPboardType**. When the filter service is invoked for a file, the services facility merely puts the contents of the file on the pasteboard. This input mechanism is useful for file types with nonstandard or special extensions whose format is nonetheless the same as a standard type. For example, if you’ve defined an image format based on a subset of TIFF and given it a file extension of **stif**, you can define a service that maps the **stif** file extension to **NSTIFFPboardType**:

```
{
    NSServices = (
      { NSFilter = "";
        NSInputMechanism = NSMapFile;
        NSSendTypes = (NSTypedFilenamesPboardType:stif);
        NSReturnTypes = (NSTIFFPboardType);
      }
    );
}
```

NSIdentity defines an empty service for data in memory, used when you invoke NSPasteboard’s **pasteboardByFilteringData:ofType:** class method. It declares that the send type is effectively identical to the return type—though the reverse isn’t necessarily true. For example, you can define a service that filters your custom image format in memory with this service specification:

```
{
    NSServices = (
      { NSFilter = "";
        NSInputMechanism = NSIdentity;
        NSSendTypes = (MyCustomImagePboardType);
        NSReturnTypes = (NSTIFFPboardType);
      }
    );
}
```

Neither **NSMapFile** nor **NSIdentity** result in any program being executed, so their services specifications lack the **NSPortName** entry.

Print Filter Services

A print filter service is invoked when the user saves a file as a PostScript file through the Print panel. When the user clicks the Save... button on the Print panel a Save panel opens with a pop-up list near the bottom. This pop-up list contains special types of PostScript that the user can choose from. A print filter service adds an entry to this list.

You implement a print filter service as a UNIX command line program that reads PostScript on the standard input stream and writes it to a file specified on the command line by a **-o** option; for example:

```
ps2superps -o outputfile.ps
```

Instead of an **NSMessage** entry, the service specification for a print filter service contains a **NSPrintFilter** entry, whose value is the extension used for the output file. If it's empty "ps" is used by default. The **NSPortName** entry is the name of the UNIX program—**ps2superps** in the example. **NSMenuItem** gives the string that appears in the pop-up list. The following entries are ignored in a print filter service specification:

NSKeyEquivalent
NSSendTypes
NSReturnTypes
NSUserData

A print filter service specification adds one entry: **NSDeviceDependent**. Its value is "YES" or "NO" (the default). If you specify "YES" for this entry the PostScript code sent through your print filter is specific to the type of printer chosen in the Print panel.

Here's a sample print filter service specification:

```
{
    NSServices = (
      { NSPrintFilter = "superps";
        NSPortName = ps2superps;
        NSMenuItem = {
            default = "Super PostScript for Chosen Printer";
            English = "Super PostScript for Chosen Printer";
            French =
                "Super PostScript pour l'imprimante sélectionnée";
            German = "SuperPostScript für ausgewählten Drucker";
        };
        NSDeviceDependent = "YES";
      }
    );
}
```

Spell Checker Services

A Spell checker service is made available in the Application Kit's standard spell checker panel. You implement a spell checker service by creating a program that uses an NSSpellServer object. See the NSSpellServer class specification for full information on creating a spell checker service. You'll want to create the spell check service as an add-on service as described in "Add-on Services." Instead of a **NSMessage** entry, the service specification for a spell checker service contains a **NSSpellChecker** entry, whose value is the text that should be used to identify the spell checker in the spelling panel's pop-up list. A spell checker service specification should also contain a **NSLanguages** entry whose value is the language for which the spell checker applies. The spell checker won't be advertised unless one of its values for **NSLanguages** matches one of the user's preferred languages.

As an example, here's the service specification for the NeXT spell checker:

```
{
    NSServices = (
      {NSExecutable = NeXTspell;
       NSLanguages = (English);
       NSSpellChecker = NeXT;
      },
    );
}
```