Paper 017-2011

Integrating an iOS Application with SAS® Platform Services

Zachary Marshall, SAS Institute Inc., Cary, NC, USA

ABSTRACT

Apple's iOS operating system for mobiles is available on over 120 million devices and is increasingly being used as a platform for in-house enterprise and business applications. iOS offers an unmatched user experience among handhelds, and is spurring an ever-growing enterprise adoption rate. This paper details the process of creating an iOS application that integrates with the SAS® Analytics Platform. Principally, it will examine the interaction between an iOS application and SAS® BI Web Services, specifically leveraging the new transport types in SAS BI Web Services 9.3, but also offering tips and tricks for integrating with other SAS SOAP Services.

INTRODUCTION

Apple’s iOS was originally released in 2007 and was available only on the Apple iPhone. It was immediately hailed by many technology pundits as a revolution in the mobile communication user experience (although Apple was not without its detractors). For the most part, critics hailed its ability to enable users to easily browse the internet, read and respond to e-mail, and also perform the normal duties of a cellular phone. Apple’s typical attention to detail meant that the original iPhone was a pleasure to use for those who chose the handheld.

Five years later, Apple now enjoys over 125 million iOS users across both the iPhone and iPad (and even more undisclosed users of the iOS-based Apple TV, which currently does not support third-party applications). What was originally simply referred to as the iPhone OS is now called iOS to reflect its presence on both the iPhone and Apple’s new tablet, the iPad. Perhaps more important, iOS has evolved to support native third-party applications beyond the simple Web application support that the iPhone originally shipped with. The Apple iOS Application Software Development Kit (SDK) has allowed independent software vendors as well as in-house application writers to deliver functionality beyond what Apple alone could ever offer. As a result, iOS is now widely used in enterprise for communication and productivity.

In-house applications for interacting with business systems are alluring because they allow the customer complete control over the user experience. Custom applications can leverage the core functionality of a software product assuming the product has well established integration points. The SAS Analytics Platform offers numerous integration points for customers including but not limited to Web services, .NET libraries, Java libraries, message queues, and file system access. Web services are a particularly attractive integration point because they offer a platform and technology agnostic means of connecting disparate systems.

WHAT ARE WEB SERVICES?

“Web services” is a general term that describes an interoperability application programming interface (API) that commonly leverages the Hypertext Transfer Protocol (HTTP) to exchange documents or messages between remote hosts[1]. Since they are designed to enable two completely different computer systems[2] to communicate with one another, Web services typically exchange documents composed of Extensible Markup Language (XML) since XML is an open standard. Also, since XML is not a binary representation, it has the added benefit of being readable by humans which can be helpful when debugging Web service applications.

The most common types of Web services are plain XML over HTTP and Simple Object Access Protocol (SOAP) services. Although both types use XML, SOAP messages must adhere to a specific format, including the usage of specific SOAP namespaces. Plain XML services can be as simple or complex as the service author wishes. Typically, plain XML services are architected using Representational State Transfer (REST) principals. REST describes how a service consumer can interact with a service and leverages the full suite of HTTP verbs (GET, POST, PUT, DELETE, and so on) to describe the action of the service invocation. Since the messages are typically very simple, RESTful services with simple message formats are used frequently when a service consumer is deployed on a device with constrained resources or where SOAP libraries are not available. On the other hand, SOAP supports additional optional standards that prescribe specific interaction patterns and formats for activities such as authentication, authorization, message routing, and more. Therefore, SOAP is typically favored in enterprise settings where these

[1] This definition is intentionally narrow. Certain Web service standards such as SOAP support other transfer formats such as message queues and the Simple Mail Transfer Protocol (SMTP). Further, communication does not need to be between remote parties— Web services can be used for inter-application or even inter-process communication.

[2] A computer system might simply be two different physical computers, or a computer system might be an entire computer network or host.

issues are important. However, REST usage in the enterprise continues to gain momentum and there are fewer and fewer circumstances where one protocol is demonstrably preferable to the other.

SAS BI WEB SERVICES

OVERVIEW

SAS BI Web Services has existed since release 9.13 of the SAS Platform and provides a means for application developers to leverage SAS Stored Processes in custom applications. SAS Stored Processes are natural complements to Web services as they are a reusable unit of SAS code that can be authored and executed in many other SAS products such as SAS Enterprise Guide and SAS Web Report Studio. SAS BI Web Services exists as an application in the SAS middle tier that offers Web service endpoints for executing SAS Stored Processes, including providing input to and obtaining results from stored process execution.

Through release 9.2, SAS BI Web Services supported only the SOAP protocol. However, starting in 9.3, SAS BI Web Services supports plain XML and JavaScript Simple Object Notation (JSON) message formats in addition to supporting RESTful access to stored processes.

SAS BI WEB SERVICES AND SOAP

Support for the SOAP protocol has been a part of SAS BI Web Services since its inception. Initial support for the XML for Analysis (XMLA) standard in 9.13 was augmented by support for fully-featured SOAP endpoints for stored processes in 9.2. These Web service endpoints can be generated by clients using SAS Management Console and have the benefit of exposing Web Service Description Language (WSDL) documents that fully describe the inputs and outputs of the Web service using XML. WSDL files enable clients to generate proxies for invoking Web services in the programming language of their choice using popular Web service libraries. For example, there are tools available from Microsoft to generate Windows Communication Foundation Web service proxies and from the Apache Axis project to generate Java Web service proxies. Countless other programming languages and libraries offer similar tools.

SAS BI Web Services accessed using the SOAP protocol enables you to send and retrieve binary content in the form of attachments. Attachments can be sent using either the SOAP with Attachments (SWA) standard or the Message Transmission Optimization Mechanism (MTOM) standard. When using SOAP, you can also secure your Web service conversations using the WS-Security extension. WS-Security provides a means to both authenticate yourself to a Web service endpoint and also to encrypt the Web service message. WS-Security is part of the WS-* collection of SOAP standards, many of which you can configure to work with SAS BI Web Services using SOAP.

BIWS AND REST

In 9.3, SAS added support for plain XML messages to SAS BI Web Services. This opens the door to Web services for many feature-limited libraries and devices. SOAP typically requires a heavy runtime library to support all of its features. With the burgeoning popularity of connected devices such as smartphones and tablet computers, combined with support for third-party application development environments for these devices, the potential for Web services is larger than ever.

You can continue to generate Web services using SAS Management Console and communicate with them using plain XML messages in 9.3, but you can save yourself time and effort by using the RESTful features of SAS BI Web Services. SAS BI Web Services now exposes both SOAP and REST Web service endpoints as URLs for each stored process stored in a metadata folder. For example, the SAS Platform ships with a set of example stored processes in the metadata folder `/Products/SAS Intelligence Platform/Samples`. To access the REST endpoint for the stored process named `Sample: Hello World` when your SAS middle tier is deployed on the host `http://myhost.com/`, simply use the URL `http://myhost.com/SASBIWS/rest/storedProcesses/Products/SAS%20Intelligence%20Platform/Samples/Sample%3A%20Hello%20World`. You must replace any spaces in the SAS folder path to the stored process with the URL-encoded version of a space, `%20`, and you must replace the colon with the URL-encoded value, `%3A`. In fact, you must replace all special characters that can appear in a SAS folder path with their URL-encoded equivalent when accessing these REST endpoints. Some programming libraries will replace these characters automatically.

Since REST endpoints do not expose a WSDL file to describe their inputs and outputs, it can be a bit harder to write clients to communicate with these endpoints. However, SAS BI Web Services establishes the convention of using the same format for plain XML REST messages as that which appears in the body of SOAP messages. For example, Listing 1 shows a fragment of typical WSDL file containing only the `types` and `message` portions of the document. The `types` section describes the XML schema types and elements that appear in the payload of the message. The `message` section describes explicitly which element from the types section will appear in the body of the SOAP message. This particular Web service simply takes two numbers and adds them together.

```
<wsdl:types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="unqualified" elementFormDefault="qualified"
targetNamespace="http://www.sas.com/xml/namespace/biwebservices"
xmlns:tns="http://www.sas.com/xml/namespace/biwebservices">
    <xsd:element name="addfloats">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="parameters" type="tns:addfloatsParameters"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:complexType name="addfloatsParameters">
      <xsd:sequence>
        <xsd:element name="num1" type="xsd:double"/>
        <xsd:element name="num2" type="xsd:double"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="addfloatsResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="addfloatsResult">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="Parameters">
                  <xsd:complexType>
                    <xsd:sequence>
                      <xsd:element name="Sum" type="xsd:double” />
                    </xsd:sequence>
                  </xsd:complexType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
<wsdl:message name="addfloats">
  <wsdl:part element="tns:addfloats" name="parameters" />
</wsdl:message>
```

Listing 1: A fragment of a WSDL file

Listing 2 shows the corresponding SOAP message a client would send to communicate with this SOAP endpoint. Here we can see the parameter values 2.3 and 5.6 being passed in for summation.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:biw="http://www.sas.com/xml/namespace/biwebservices">
   <soapenv:Header/>
   <soapenv:Body>
      <biw:addfloats>
         <biw:parameters>
            <biw:num1>2.3</biw:num1>
            <biw:num2>5.6</biw:num2>
         </biw:parameters>
      </biw:addfloats>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 2: A typical SOAP message

To obtain the full REST message that you need to communicate with the REST endpoint, simply copy the first child of the `soapenv:Body` element that you would use when communicating with the SOAP endpoint. The use of namespaces in the REST messages is optional. Listing 3 shows the simplified REST payload.

```
<addfloats>
  <parameters>
    <num1>2.3</num1>
    <num2>5.6</num2>
  </parameters>
</addfloats>
```

Listing 3: A plain-XML REST message

As you can see, this message is much simpler than the full SOAP message and is far easier to construct in mobile client libraries, as you'll see later when we explore how to write iOS application code that uses SAS BI Web Service REST endpoints.

Using the new REST endpoints in SAS BI Web Services does not mean that you cannot send and receive binary content. However, since MTOM and SWA are SOAP-specific standards, when attachments are used by your stored process they appear in REST XML messages as base64-encoded binary data. Base64 is a data type that encodes binary data in a strict subset of printable ASCII characters. This means that binary files that might contain unprintable characters can be returned in XML messages, but this encoding format is necessarily inefficient, so do not use plain XML messages with stored processes when sending or receiving large attachment files.

Since REST-based consumption of Web services emphasizes accessing resources through well-structured URLs, you can also access binary content that is returned by a stored process by specifying the returned resource in a URL. Stored processes can return binary data by either using packages or data targets (also known as streams). If you have a stored process named `Foo` that writes a PDF to a stream named `Bar`, then you can access that PDF directly using a URL similar to `http://myhost.com:8080/SASBIWS/rest/storedProcesses/MyFolder/Foo/streams/Bar`. You can even enter this URL into a Web browser and view the PDF there to verify functionality. SAS BI Web Services writes the content-type specified for the stream as an HTTP header that can help clients, especially browsers, determine how to display binary files.

iOS

OVERVIEW

Apple's custom-built operating system for their mobile devices is, according to Apple, actually a version of its desktop operating system, Mac OS X. More specifically, iOS is a UNIX-based operating system that features custom libraries written by Apple specifically for the platform. iOS runs on a suite of Apple products including the iPhone, the iPod Touch, the iPad, and the Apple TV. To develop applications for the iOS platform, you must become a member of the iOS Developer Program through Apple. There are varying levels of membership that afford different benefits. The free version of the membership enables you to write applications using the iOS SDK to access SDK documentation and to test your programs on simulators included in the SDK. You can also purchase other levels of membership that add capabilities. An enterprise membership adds the ability to create development teams, to download beta versions of iOS for use in testing, to access technical support and forums for help, to test on actual iOS devices, and to distribute your custom applications within your company. You can sign up for free memberships or purchase higher membership levels on http://developer.apple.com/.

Third-party applications for iOS are written in a programming language called Objective-C that is a superset of the standard C programming language. Objective-C adds messaging to the C language. Typically, you interact with an object in an Objective-C program by sending it a message (as opposed to calling a method on that object as you might do in C++). A message dispatch looks like the following in Objective-C:

```
[object methodName: argument];
```

Message arguments are named in Objective-C message dispatches. This allows for greater clarity in written code, but it also means that Objective-C programs are more verbose than their counterparts in other languages. The following code shows a multi-argument message dispatch:

```
[object invokeWithParm: myParm1 andSecondParm: myParm2];
```

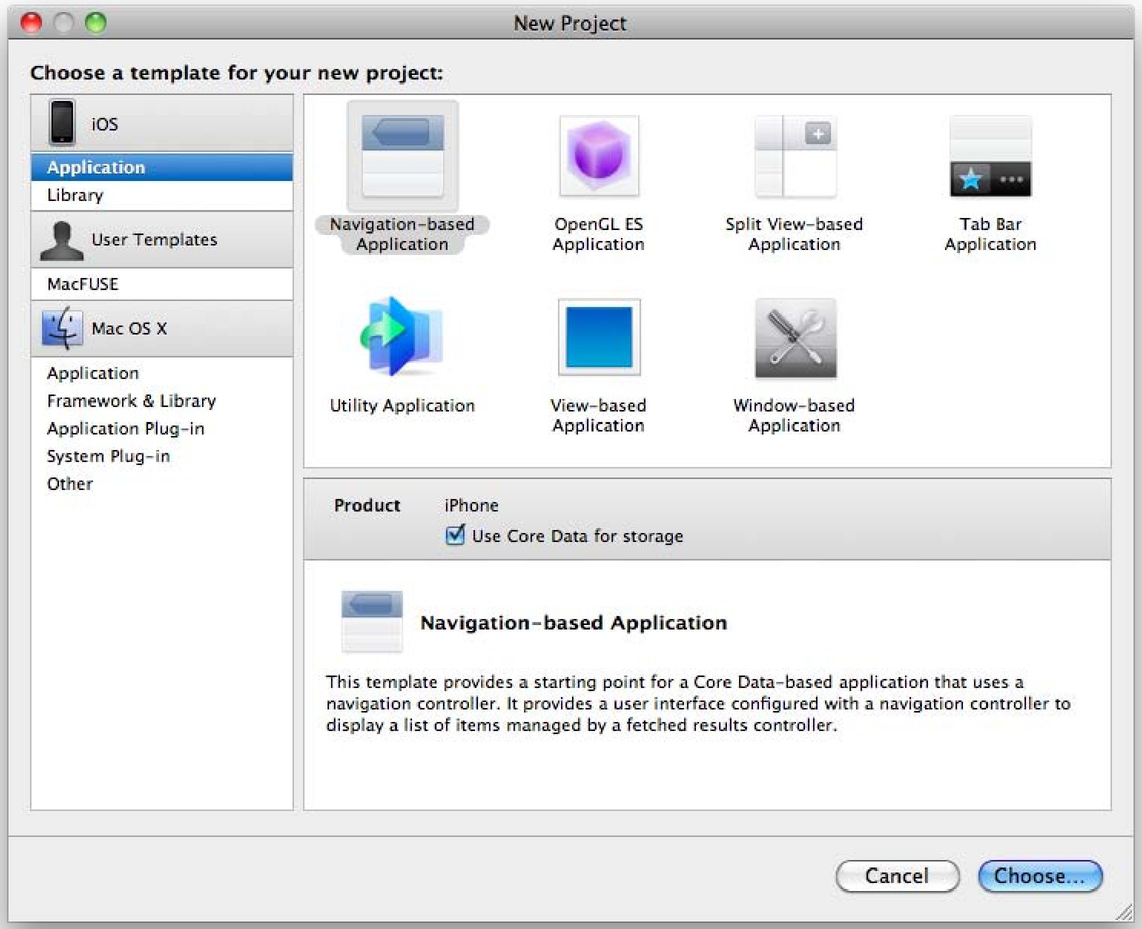
There are many other additions to Objective-C that set it apart from standard C. Apple's *Introduction to The Objective-C Programming Language* is a good, free place to start[3]. It might also be helpful to learn Objective-C in the context of where you will actually use it. To this end, Stephen G. Kochan's *Programming in Objective-C* is recommended for its introduction to Objective-C in the context of Apple's frameworks for iOS and Mac OS X development.

CREATING A PROJECT

The easiest way to write and compile iOS applications is to use Apple's Xcode. Xcode is an Integrated Development Environment (IDE) that assists you in writing Objective-C programs for iOS. Because it includes project templates specifically for iOS applications, getting started with application development using Xcode is fairly quick. Xcode is included as part of Apple's Developer Toolkit and the iOS SDK for free, but is only available for Mac OS X. Therefore, you must have a Mac to write iOS applications.

To create a simple Objective-C project, start Xcode and choose **File > New** from the menu bar. The New Project wizard appears to help you create a project. When you have the iOS SDK installed, the New Project wizard (Figure 1) shows you the different types of iOS Applications you can build.

[3] Available at http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html



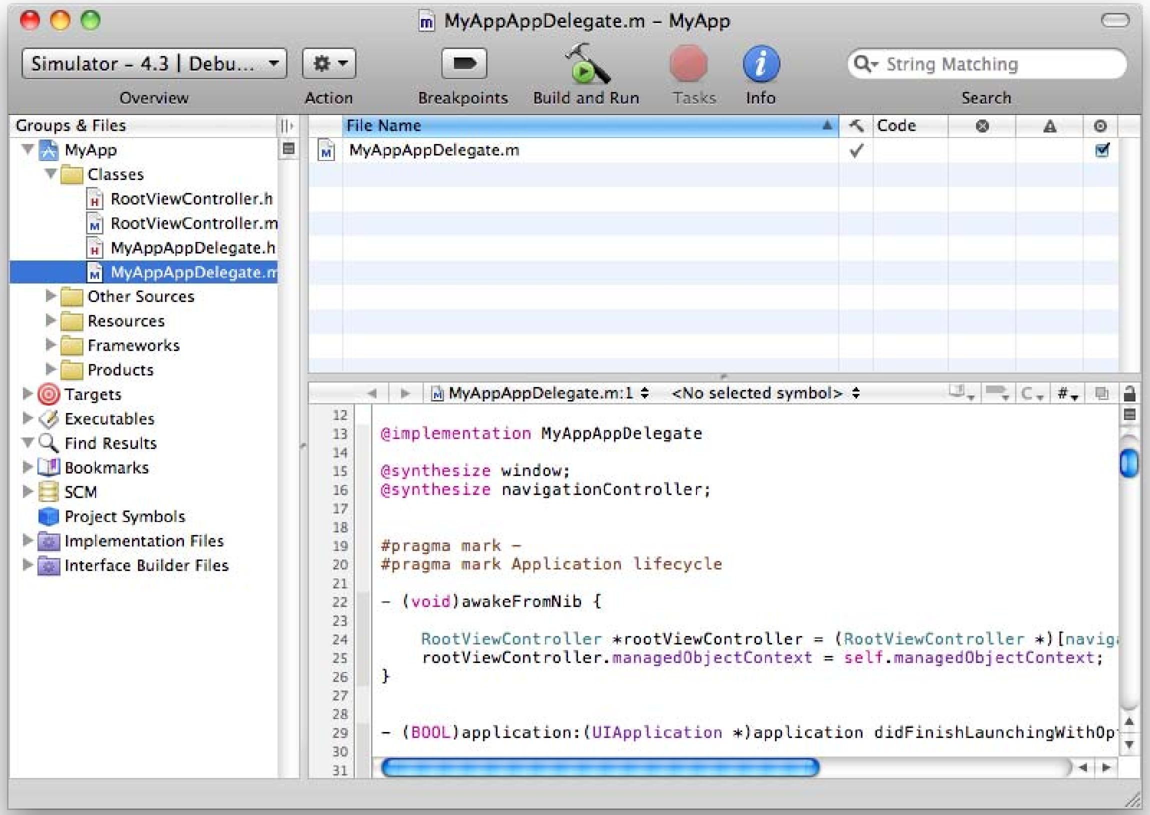
Product iPhone
✓ Use Core Data for storage

Navigation-based Application

This template provides a starting point for a Core Data-based application that uses a navigation controller. It provides a user interface configured with a navigation controller to display a list of items managed by a fetched results controller.

Figure 1: The Xcode New Project wizard.

Once you create your iOS application, Xcode presents an integrated Window (Figure 2) that enables you to manage your iOS application assets, write and compile your Objective-C code, and debug your application when its running in a simulator or on an attached iOS device.



```
@implementation MyAppAppDelegate

@synthesize window;
@synthesize navigationController;


#pragma mark -
#pragma mark Application lifecycle

- (void)awakeFromNib {

    RootViewController *rootViewController = (RootViewController *)[naviga
    rootViewController.managedObjectContext = self.managedObjectContext;
}


- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOp
```

Figure 2: The main Xcode window.

WRITING iOS CODE

When used in iOS applications, Objective-C code is typically split into pairs of files. Files ending with the extension .h are header files and contain declarations of Objective-C classes and methods. Files ending with the extension .m are implementation files containing definitions of classes and methods. Within Xcode, you can edit .h or .m files in the bottom-right pane, or double click on files to edit them in their own window. Xcode provides convenient syntax highlighting and code completion for the Objective-C language. Code completion can be activated in the code editor by pressing the escape key. Doing so brings up a list of context-specific methods to select from, saving you time typing and looking up documentation.

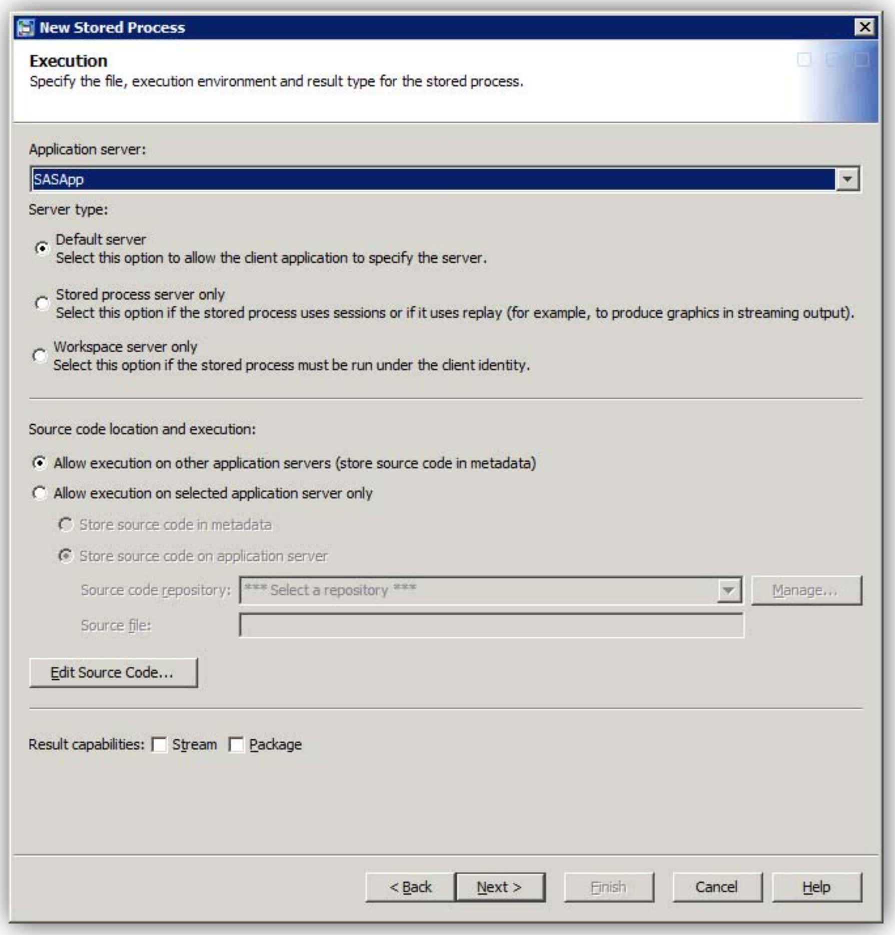
Even the most basic iOS application needs to perform actions such as displaying graphical user interface (GUI) elements on the screen, receiving input from the user, and responding to OS events. Since there are numerous common tasks that all applications must perform, it would not make sense for each application developer to implement these on his own. Therefore, Apple provides a set of libraries called Frameworks that an iOS application can use. Each Framework has a general theme. For instance, many iOS applications will use the UIKit Framework, the Foundation Framework, the CoreGraphics Framework, and the CoreData Framework. The UIKit Framework contains a collection of classes implemented by Apple that provide common GUI widgets. The Foundation Framework provides classes for basic data types and system constructs. The CoreGraphics Framework is responsible for compositing and displaying the fancy graphical transitions that make iOS devices and applications such a pleasure to use. And last, applications that need to manipulate collections of structured data typically use the CoreData Framework for persistence and retrieval. These tasks are so common that these four frameworks are included in many of the iOS project templates by default. There are many other Apple-supplied frameworks that provide additional functionality in iOS such as integration with the messaging system, with the mapping system, with the address book, with any device cameras, and with external accessories. Using these frameworks will allow you take advantage of the full spectrum of capabilities of an iOS device.

When using classes that are part of an Apple-supplied framework, you'll find that many of the class names begin with NS. For example, the string class most often used in Apple frameworks is `NSString`. This is because many of these classes were originally part of the libraries for the NeXTSTEP Operating System in some form. NeXT, the company responsible for NeXTSTEP, was started by Apple's current CEO Steve Jobs and was acquired by Apple in 1996. The collection of frameworks that are available for Mac OS X and for iOS are referred to as Cocoa and Cocoa Touch, respectively.

USING SAS BI WEB SERVICES IN AN iOS APPLICATION

CREATING SAS STORED PROCESSES AND WEB SERVICES

SAS Stored Processes can be created using SAS Management Console. In the past, you had to distribute the source SAS code for a stored process to each computer (that was a member of a logical server) where you wanted to allow users to execute that stored process. Further, each member of a logical server had to store the source code in the same location. Once you distributed this code to each machine, you had to store the path to the code in metadata. Maintaining this code can be tricky since it requires updating multiple files on different machines. Starting in 9.3, you can now store your SAS source code for a stored process in metadata directly in the Stored Process definition. Creating the stored process definition is still similar to how you created it in 9.2.


New Stored Process

Execution
Specify the file, execution environment and result type for the stored process.

Application server: SASApp

Server type:
[✓ Default server] Select this option to allow the client application to specify the server.
Stored process server only — Select this option if the stored process uses sessions or if it uses replay (for example, to produce graphics in streaming output).
Workspace server only — Select this option if the stored process must be run under the client identity.

Source code location and execution:
[✓ Allow execution on other application servers (store source code in metadata)]
Allow execution on selected application server only
Store source code in metadata
[✓ Store source code on application server]
Source code repository: *** Select a repository ***
Source file:

Result capabilities: Stream ☐ Package ☐

Figure 3: Execution options for a stored process

1. In SAS Management Console, navigate to the folder where you wish to create the stored process definition.
2. Right-click on the folder to bring up the context menu and highlight **New** to bring up a submenu then select **Stored Process...**
3. Fill in the general information about the stored process in the first pane of the **New Stored Process Wizard**.
4. The next pane of the wizard specifies how the stored process will execute (Figure 3). Here you can choose

the SAS Server where the stored process will execute in the **Application Server** drop-down list.

5. In the **Server type** section you can indicate whether the stored process should be run on a Workspace Server or a stored process Server depending on the features that you need during execution. You can also indicate that no specific type is necessary, allowing clients to choose the server type at run time.
6. In the **Source code location and execution** section, choose the radio button labeled **Allow execution on other application servers (stored source code in metadata)** then click the **Edit source code** button.
7. In the resulting pop-up window, you can type your SAS code that makes up your stored process. You may find it easier to compose your stored process in an external editor such as Base SAS or SAS Enterprise Guide and then copy and paste the program into the pop-up window in SAS Management Console.
8. The remaining steps in the wizard allow you to specify input prompts, output parameters, data sources, and data targets for your stored process. These steps should be familiar to authors of 9.2 stored process definitions.

Once you have created your stored process definition it will immediately be available for execution using either SOAP or plain XML using REST endpoints in the SAS BI Web Services application in the middle tier. If you store your stored process source code in the stored process definition, you are ready to create Web service clients to leverage your SAS code.

USING THE iOS FOUNDATION FRAMEWORK TO COMMUNICATE WITH SAS BI WEB SERVICES

The Foundation Framework provided by Apple in the iOS SDK includes classes for manipulating XML data. You can use these classes to invoke and view the results of SAS Stored Processes using SAS BI Web Services in 9.3. Here's how to create a simple project that uses a SAS Stored Process to add two numbers and return the result. The result will be displayed in a simple iOS application.

Using SAS Management Console 9.3, create a new stored process in the `Shared Data` folder and name it `addfloats`. In the Execution panel, pick an application server on which to execute the SAS code (SASApp is a good choice if it's present). Pick the default server type and choose to store the following SAS source code in metadata:

```
%let Sum = %sysevalf(&num1 + &num2);
```

On the Parameters panel, create a new prompt and give it a name and displayed text of `num1`. On the **Prompt Type and Values** tab of the New Prompt wizard, change the prompt type to Numeric and deselect the checkbox labeled **Only allow integer values**. Repeat these steps to create another prompt named `num2`. In the Output Parameters section, create a new output parameter named `Sum` with a label of `Sum` and a type of Double. Proceed through the wizard to finish defining the new stored process. You can now verify that the stored process was created successfully and available for execution in SAS BI Web Services 9.3 by accessing the WSDL file of the SOAP endpoint for this service. You won't actually use the SOAP endpoint in this example, but it provides a good test to make sure your application server is up and running and can view your stored process definitions properly. Pull up http://host.com:port/SASBIWS/services/Shared%20Data/addfloats?wsdl in a Web browser, replacing the host and port placeholders in the URL with values that correspond to your SAS middle tier installation.

In Xcode, create a new iOS application using the **New Project** wizard and use the **View-based Application** template. Make sure iPhone is selected as the Product for this example then press the Choose button. Name the new application something meaningful such as addfloats. Your new iOS application will contain the URL for the Web service we just created. When the application starts up, it will access this Web service URL and use the HTTP verb POST to send XML to be evaluated by the SAS Stored Process. SAS BI Web Services will receive your XML input, deserialize it to obtain the parameters `num1` and `num2`, and invoke the SAS Stored Process with these parameters. The stored process will receive the values of `num1` and `num2` as macro variables and use the SAS code you supplied in the stored process definition to sum the values and store the result in the macro variable `Sum`. SAS BI Web Services will then obtain the value of the `Sum` macro variable before the stored process finishes executing and will generate XML that contains this value and return it to your iOS application. Your iOS application will then parse this XML and display the result in the application.

Start by opening the `addfloatsViewController.h` file using the Groups & Files pane on the left of the main Xcode window. This file is responsible for controlling how the main view of your application displays itself and how it handles common OS scenarios. You'll see that this file declares our addfloatsViewController and that it inherits from the UIViewController class. The UIViewController class is a standard class supplied by Apple in the UIKit Framework.

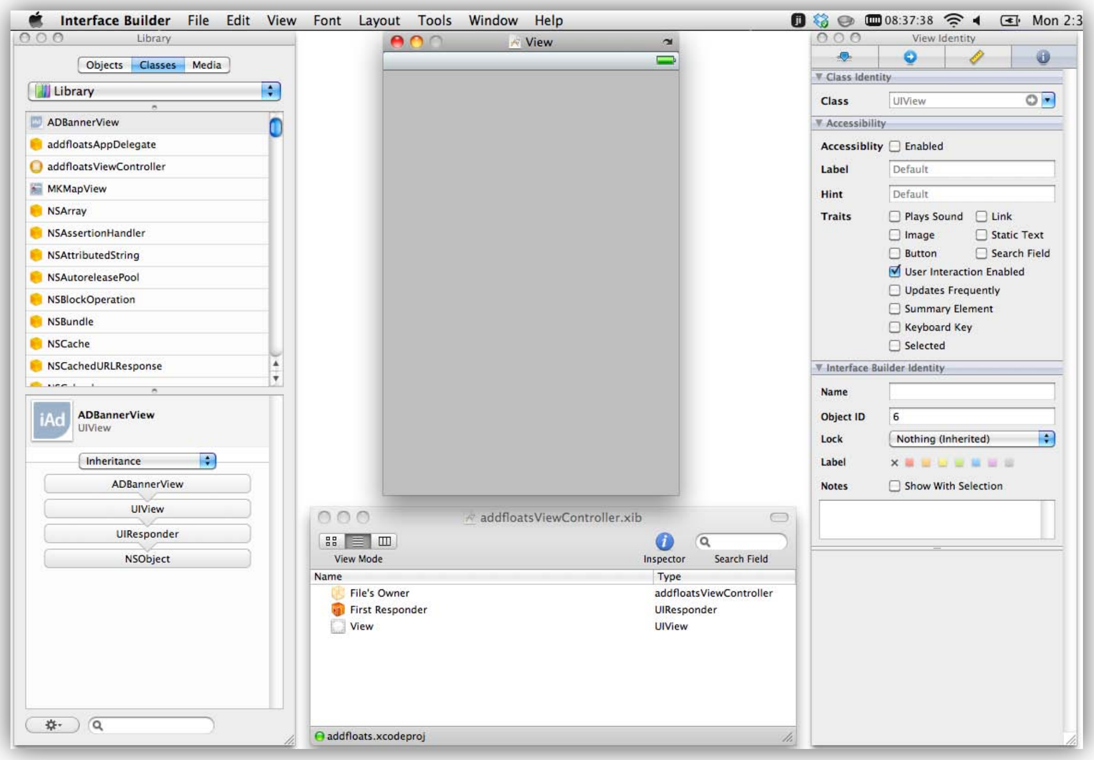
```
#import <UIKit/UIKit.h>
@interface addfloatsViewController : UIViewController {}
@end
```

Since you want to display the output of your stored process in your application, we will need to connect our controller to a GUI element that will actually display the text. To do this, we will add an `IBOutlet` to our controller interface (make sure to save the code after adding the outlet).

```
#import <UIKit/UIKit.h>
@interface addfloatsViewController : UIViewController {
  IBOutlet id label;
}
@end
```

An outlet is simply a hint that we will later connect this object up to our GUI. The type `id` is a generic pointer type in Objective-C that can be used to refer to any type. You'll actually hook this label up to an `NSLabel` so you could replace `id` with `NSLabel` in the `addfloatsViewController.h` file, but using `id` gives you the flexibility to change that implementation detail later. Many Objective-C and Cocoa guides will emphasize the flexible nature of the language, but that flexibility has a cost if you're not careful; using the loose-typing and dynamic dispatching features of Objective-C means that you have to be more careful when you write code since the compiler will not be ensuring type safety.

Your application will display the summation in a view in your iOS application. You could write all the code by hand that creates and displays the necessary GUI components, but Apple provides a convenient GUI editor that makes this task much easier. These GUI components can be arranged using Interface Builder and then packaged up in a nib file. Xcode already included a nib file for your main view of your new application. In the Groups & Files pane on the left of Xcode, select the Resources folder then double-click the `addfloatsViewController.xib` to open the view in Interface Builder.



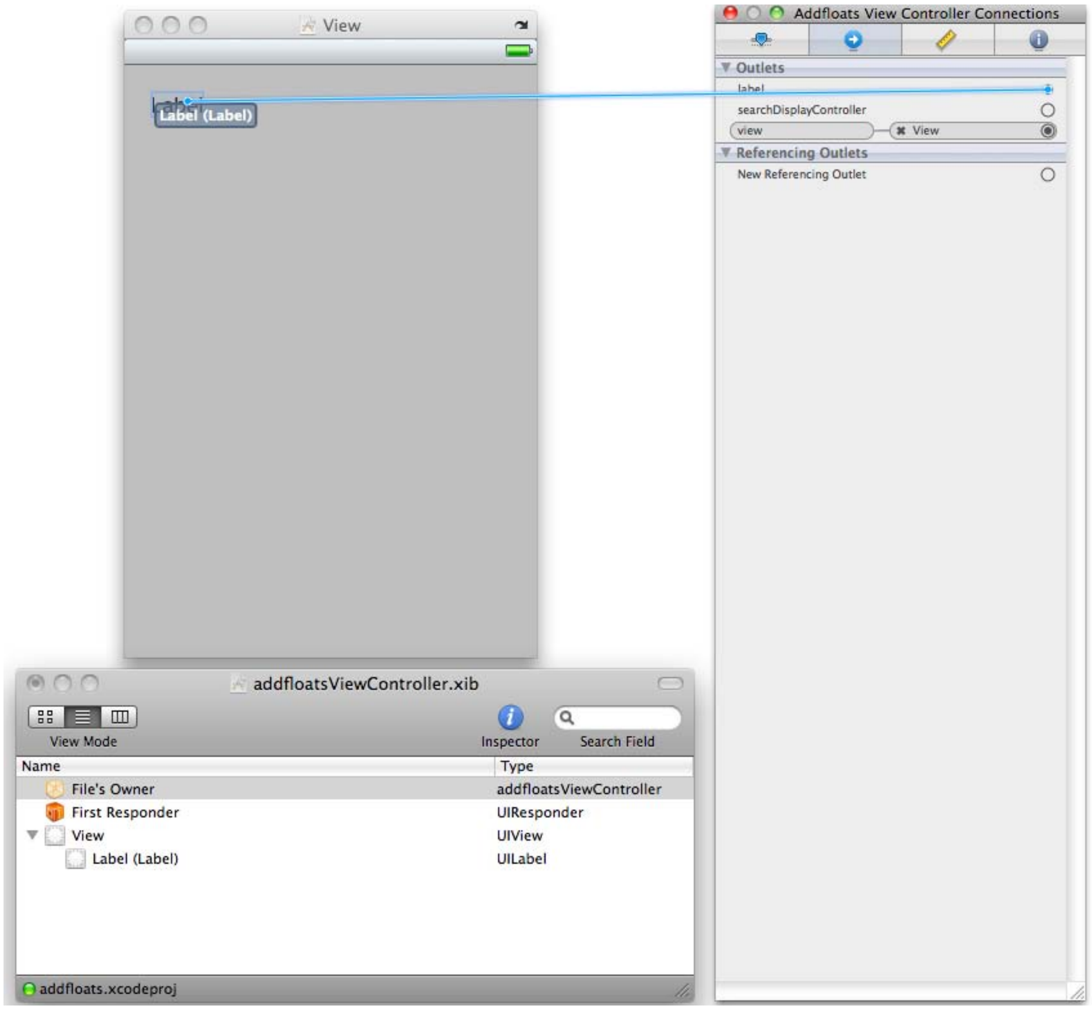
Name	Type
File's Owner	addfloatsViewController
First Responder	UIResponder
View	UIView

Class: UIView
Accessibility: ☐ Enabled
Label: Default
Hint: Default
Traits: ✓ User Interaction Enabled
Object ID: 6
Lock: Nothing (Inherited)

Figure 4: Working in Interface Builder

In Interface Builder, add a `UILabel` to the main `UIView` of your `addfloatsViewController.xib`. First, make sure the Library window is showing by choosing **Tools > Library** from the menubar. The Library is a collection of objects, classes, and media files that you can use in your application's UI. Select the Classes tab and at the bottom of the Library window type `UILabel` in the search box to narrow the selection to just this class. Now drag the `UILabel` class from the upper pane of the Library directly onto the gray view window. Now, you need to connect your new `UILabel` to your view controller. To do that, make sure the Connections Inspector is showing by choosing **Tools > Connections Inspector** from the menubar. Now, select the File's Owner item in the main window of Interface

Builder. In the Outlets section of the Connections Inspector, find the item named `label`[4]. This corresponds to the `IBOutlet` you added earlier to the view controller. Click and drag from the circle to the right of label in the Connections Inspector to the UILabel in the view window to connect the UI element to the label in your controller (Figure 5).



Name	Type
File's Owner	addfloatsViewController
First Responder	UIResponder
View	UIView
Label (Label)	UILabel

Figure 5: Connecting a label

With the UILabel in your nib connected to the outlet in your controller, you can now send messages to the label in the UI to display specific text from your controller. Save the nib in Interface Builder and return to Xcode. You are now ready to write the code that interfaces with SAS BI Web Services.

The NSXMLParser class is included in the iOS Foundation Framework. In order to use the NSXMLParser, you must indicate that your controller can act as a parser delegate. A delegate parser simply receives messages when certain actions occur during the parsing of an XML document. The delegate can then act on these messages to determine what XML element has been found and to store its contents. To indicate that your controller is an NSXMLParser delegate, open addfloatsController.h in Xcode and modify the code to resemble the following:

[4] If the label element is not showing in the Connections Inspector window, choose File > Reload All Class Files from the menu bar in Interface Builder to reload any changes that have been made in Xcode.

```
#import <UIKit/UIKit.h>

@interface addfloatsViewController : UIViewController <NSXMLParserDelegate> {
   IBOutlet id label;
   BOOL inSumElement;
   NSString *sum;
}

- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qualifiedName
  attributes:(NSDictionary *)attributeDict;

- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName;

- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string;

@end
```

The `<NSXMLParserDelegate>` in the interface declaration indicates to the compiler that your controller is a suitable parser delegate and the inclusion of the three parser methods enables you to respond to specific events during parsing. You also added the new `BOOL` instance variable `inSumElement` to use as a control during parsing and the `NSString` instance variable `sum` to hold the string representation of the sum.

To implement these new methods and parsing logic, open the implementation file for your view controller `addfloatsViewController.m` in Xcode. The iOS framework will call certain methods automatically as your program and views load. Find the `viewDidLoad` method and uncomment it. Augment the `viewDidLoad` method so that it resembles the following:

```
- (void)viewDidLoad {
   [super viewDidLoad];

   NSURL *biwsUrl =
     [NSURL URLWithString:
     @"http://host:port/SASBIWS/rest/storedProcesses/Shared%20Data/addfloats"];

   NSString *requestBody =
     @"<addfloats> \
       <parameters> \
           <num1>3.2</num1> \
           <num2>4.7</num2> \
       </parameters> \
   </addfloats>";
   NSData *requestData = [requestBody dataUsingEncoding:NSUTF8StringEncoding];

   NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:biwsUrl];
   [request setHTTPMethod:@"POST"];
   [request setHTTPBody:requestData];
   [request setValue:@"application/xml" forHTTPHeaderField:@"Content-Type"];
   [request setValue:@"application/xml" forHTTPHeaderField:@"Accept"];

   NSData *responseData =
     [NSURLConnection
             sendSynchronousRequest:request returningResponse:nil error:nil];

   NSXMLParser *parser = [[NSXMLParser alloc] initWithData:responseData];

   [parser setDelegate:self];
   [parser parse];
}
```

Make sure to replace host and port in the NSURL string with values that correspond to your actual SAS middle tier installation. Here you first construct a URL that points to the REST endpoint for your newly created Web service and then you construct a string that contains the payload of the message you will send to this endpoint (the \ characters at the end of each line enable you to span the string constant across multiple lines in your source code—these characters will not be present in the message that is sent to the Web service endpoint). You must then convert the string into a data representation that can be used by the `NSMutableURLRequest` class. The `dataUsingEncoding` method specifies that you wish to send the request in UTF-8, which is the format supported by SAS BI Web Services. The `NSMutableURLRequest` instance specifies the URL, HTTP method, HTTP headers, and request body of your Web service message. It is important to set the Content-Type and Accept HTTP headers when interacting with SAS BI Web Service REST endpoints since these headers indicate how to process the incoming message and in which format to return the response message.

Sending the `sendSynchonronousRequest:returningResponse:error` message to `NSURLConnection` actually sends your XML payload data to the REST endpoint using the HTTP parameters you specified and returns a data representation of the result. You specify the reserved keyword `nil` as the `returningResponse:` and `error:` parameters to indicate that you do not care about the response metadata nor about any errors that might occur during the connection process. In actual iOS applications, you would very likely want to track any errors that occur so that you could display a message to the user.

Last, you instantiate a `NSXMLParser` and initialize it with the data returned from the Web service invocation. The only thing left to do is to tell it to parse! However, if you were to run the application now, nothing would happen. That is because you have not added the parser method implementations. Do so now:

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qualifiedName
  attributes:(NSDictionary *)attributeDict
{
  inSumElement = [elementName isEqualToString:@"Sum"];
}
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
{
  if (inSumElement && [elementName isEqualToString:@"Sum"])
   [label setText:sum];
}
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
  if (inSumElement)
  {
  if (!sum)
    sum = [string copy];
  else
    sum = [sum stringByAppendingString:string];
  }
}
```

The `didStartElement` method observes when the Sum element is encountered in the XML document during parsing and sets the `BOOL` variable `inSumElement` to track this condition. Next, the `foundCharacters` message is sent and if the application is currently processing the Sum element, the application records the value element. Since `NSXMLParser` can send the `foundCharacters` method more than one time for a given element's contents, you must set the value of `sum` to the found characters on the initial pass (when `sum` is equal to `nil` and `!sum` evaluates to true), then append the found characters on subsequent passes.

Now, make sure addfloats – iPhone Simulator is selected in the configuration drop-down menu in top left of Xcode then click the **Build and Run** (or **Build and Debug** if you've added any breakpoints) button in the Xcode window to test out your application. If all goes well, a simulated iPhone window will appear showing the result of invoking your stored process to add two numbers using Web services (Figure 6). Sure, you could have written one line of Objective-C to perform the same task, but what's the fun in that? Moreover, more practical applications of these technologies will enable you to do things you otherwise never could do on a mobile device alone!

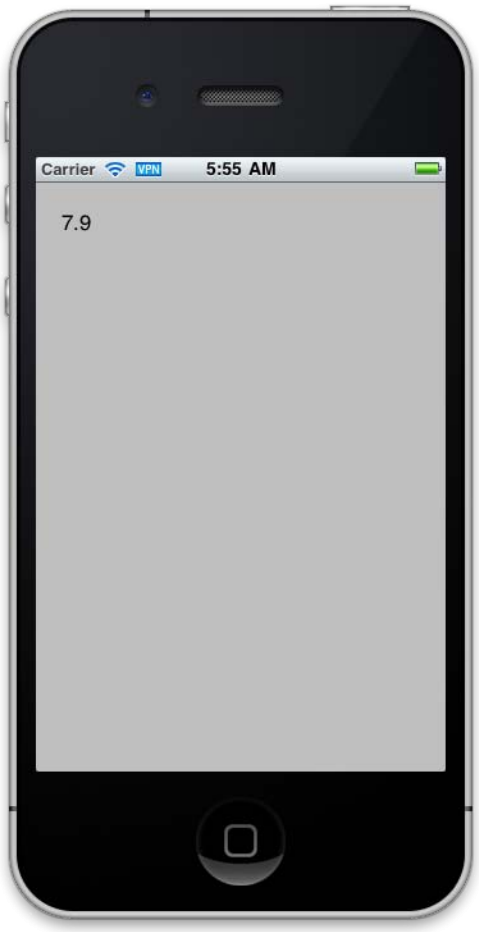


Figure 6: Results of the addfloats Web service in an iPhone application

USING SOAP LIBRARIES TO COMMUNICATE WITH SAS BI WEB SERVICES

While using plain XML with RESTful endpoints is fairly easy for small input and output messages in an iOS application, there are many reasons why you might prefer to use SOAP endpoints instead. Using a SOAP endpoint means that you can take advantage of SOAP tools that generate code for invoking Web services. This generated code will contain methods that you can invoke to set parameter values that will shelter you from having to manually create the XML requests. Further, generated code might support obtaining specific output parameters using simple method invocation. Also, until you upgrade to SAS 9.3, you will need to use SOAP with SAS BI Web Services 9.2 from your iOS applications.

Because the iOS SDK does not currently include any native SOAP libraries or tools, you must use a third-party toolset to interact with SOAP Web services from iOS applications. The SudzC open source project is an online tool available from http://sudzc.com/ that generates SOAP proxy code in Objective-C. You can use SudzC to generate code that you can use to invoke SAS BI Web Services using SOAP.

Navigate to the SudzC Web page in a Web browser and use the tool to upload the WSDL file from the `addfloats` SOAP endpoint after picking a namespace for your generated code (such as SAS). After clicking the **Generate** button, a download will start with a zip file containing your generated Objective-C SOAP proxy code. Unzip this file and open index.html from the Documentation folder. This document contains step-by-step instructions on how to add the generated proxy code to your project and get it compiling. There is also a screencast available on the SudzC homepage that walks you through the process of using the generated proxies. Using the generated proxies, the resulting code needed to invoke the SOAP Web service is reduced to the following:

```
- (void)viewDidLoad {
   [super viewDidLoad];
   SASServices *allServices = [SASServices service];
   SASaddfloats *addfloatsService = [allServices addfloats];
   SASaddfloatsParameters *parameters = [[SASaddfloatsParameters alloc] init];
   parameters.num1 = 3.2;
   parameters.num2 = 4.7;

   [addfloatsService addfloats:self action:@selector(handleAddFloats:)
   parameters:parameters];
}

-(void)handleAddFloats: (id) result
{
   id sum = [[[result valueForKey:@"addfloatsResult"] valueForKey:@"Parameters"]
   valueForKey:@"Sum"];
   [label setText:sum];
}
```

As you can see, the SOAP code using generated proxies is much shorter. However, the overall sizes of the final Xcode project and iOS application are much larger due to the inclusion of the generated code, custom SOAP libraries, and custom XML libraries. Additionally, it is much easier to set the input values and obtain the results using these proxies. You set the input parameters using generated properties on the `SASaddfloatsParameters` class and use Cocoa dictionaries (similar to maps in other programming languages) to retrieve the sum.

ADVANCED INTEGRATION TECHNIQUES

DOWNLOADING AND DISPLAYING BINARY FILES

SAS Stored Processes often produce binary output as either a stream or package, typically using the Output Delivery System (ODS). ODS provides a convenient mechanism for producing assets other than SAS files that can contain images, text, and special formatting. The following SAS code sample represents a stored process that produces a package using ODS output that contains a PROC GHART graph of the SASHELP.CLASS dataset.

```
*ProcessBody;
%STPBEGIN;
  proc gchart data=sashelp.class;
    vbar age / discrete;
  run; quit;

  proc print data=sashelp.class noobs; run; quit;
%STPEND;
```

The `%STPBEGIN` and `%STPEND` macros take care of setting up the ODS destination and ODS options for you. When this stored process is executed, the resulting package is returned via SAS BI Web Services and contains a single PNG image package entry. If this SAS code is deployed with a stored process definition at the location `/Shared Data/classpackage` then the image can be accessed directly by using the URL http://host:port/SASBIWS/rest/storedProcesses/Shared%20Data/classpackage/packages/0 (after changing host and port to match your installation). Observe that `/packages/0` is appended to the path to the stored process resource. The `/packages/` portion of the URL tells SAS BI Web Services that you want to retrieve an item from the package output and the number 0 tells SAS BI Web Service specifically which entry you wish to retrieve starting with an index of 0 for the first item (since not all package entries may be named, you must retrieve entries using an index). You can construct URLs like this in your iOS application to access the contents of packages and display them to the user.

```
NSURL *url = [NSURL URLWithString:
   @"http://host:port/SASBIWS/rest/storedProcesses/Shared%20Data/
   classPackage/packages/0"];

NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
[request setHTTPMethod:@"GET"];
NSData *imageData = [NSURLConnection sendSynchronousRequest:request
   returningResponse:nil error:nil];
imageView.image = [UIImage imageWithData:imageData];
```

This example uses the same `NSURLConnection` technique from earlier to retrieve an `NSData` object that can be used to instantiate a `UIImage`. The image property of a `UIImageView` class instance in a nib is then set to the new `UIImage`. In just five lines of code, you can display binary output from a stored process in an iOS application using SAS BI Web Services. Resizing or scaling the image to fit the iPhone or iPad screen is left as an exercise for the reader. Of course, it's just as easy to view basic binary such as images and PDFs using the built-in iOS Safari Web browser. However, unlike the browser, your custom applications can be written to display any binary format you choose.

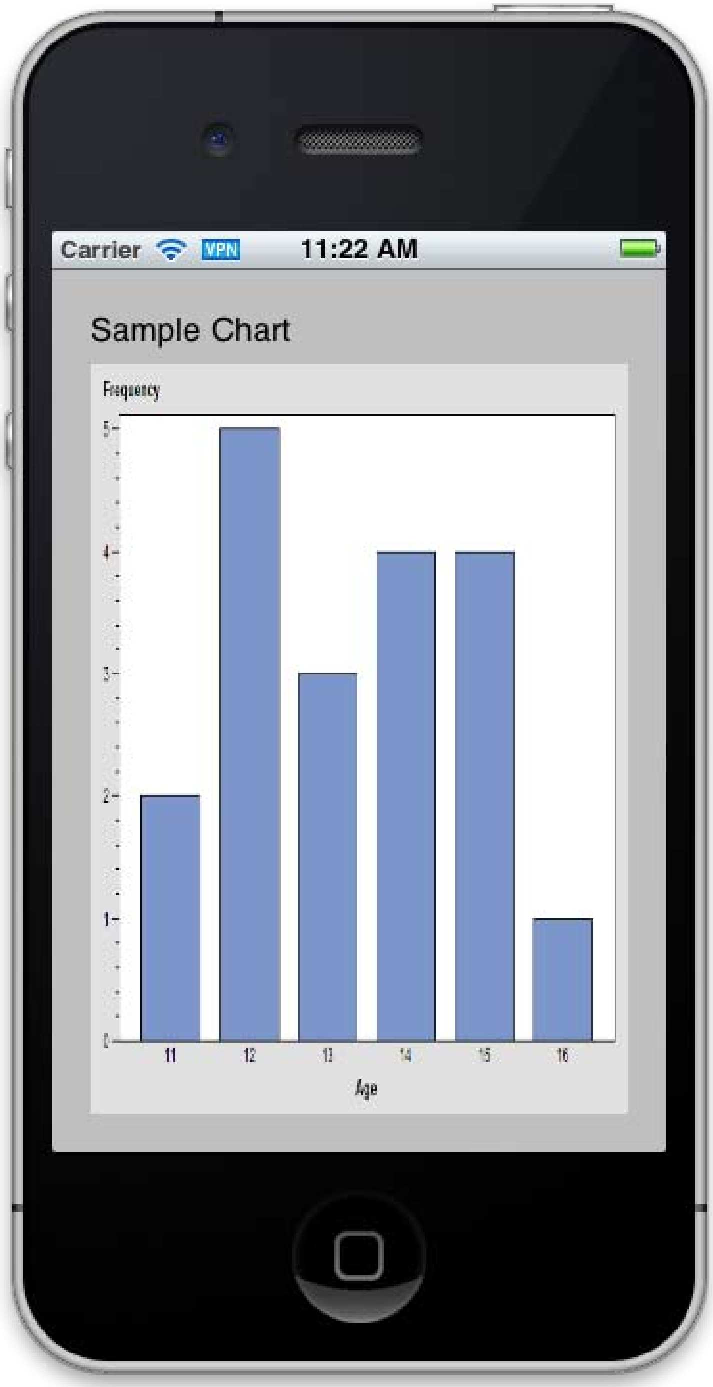


Figure 7: A chart produced using a SAS Stored Process displayed natively on the iPhone

SECURITY

Because some Web services process or output sensitive information, developers often must provide some form of security on the Web service operation. There are three main components of securing Web services: authentication, authorization, and encryption. Authentication refers to how a Web service client asserts his identity, authorization refers to which identities can access specific resources, and encryption refers to making the information in a Web service exchange unreadable by parties other than the intended recipient. Authorization with SAS BI Web Services is typically controlled by setting permissions in metadata on the stored processes you want to protect. Encryption is most easily handled by using SSL over HTTP for all Web service exchanges, though various WS-* standards exist for SOAP that describe more robust and flexible means of achieving encryption.

To authenticate, a client typically supplies a credential that uniquely identifies itself to the service. This credential can come in many forms: a user name and password pair, a Kerberos token, or a client SSL certificate. HTTP supports all of these forms of credentials, although the user name and password pair is most widely used is RESTful Web services because of its simplicity. SAS BI Web Services supports client authentication using a user name and password pair through HTTP Basic Authentication. Basic Authentication allows Web service clients to specify a user name and password in the HTTP headers of an HTTP request. During the authentication exchange, the client provides the Authorization HTTP header with a value that contains a base64-encoded form of the client's user name and password, separated by a colon. When a client's user name is foo and their password is bar, the base64-encoded version of `foo:bar` is `Zm9vOmJhcg==`. Therefore, a simple HTTP GET request from the user foo to access a resource protected by basic authentication might look like:

```
GET /protectedResource HTTP/1.1

Host: somehost.com
Authorization: Basic Zm9vOmJhcg==
```

Because an intercepted base64 string can be easily decoded, basic authentication should always be used in conjunction with encryption: namely, SSL over HTTP.

Both the `NSURLConnection` class for plain XML REST messages and the SudzC generated SOAP proxies support HTTP basic authentication. To supply a basic authentication header in an HTTP request using `NSURLConnection`, refer to the following code:

```
#define M_USER "foo"
#define M_PASS "bar"
#define M_RESOURCE "myhost.com/resource/item"
NSURL *url = [NSURL URLWithString:@"http://" M_USER ":" M_PASS "@" M_RESOURCE];
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:url];
```

Here you supply the user name and password as part of the actual URL and the `NSURLRequest` will take care of supplying those values when the Web service issues a HTTP Unauthorized challenge response (code 401). Other options for providing credentials include using `NSURLCredentialStorage` to provide both basic authentication credentials and more advanced and secure credentials. See the iOS documentation on `NSURLCredentialStorage` for more information.

Specifying a user name and password to the SudzC-generated SOAP proxies is even easier. The proxies provide user name and password properties that can be set to `NSString`s.

```
SASServices *allServices = [SASServices service];
SASaddfloats *addfloatsService = [allServices addfloats];
addfloatsService.username = @"foo";
addfloatsService.password = @"bar";
SASaddfloatsParameters *parameters = [[SASaddfloatsParameters alloc] init];
parameters.num1 = 3.2;
parameters.num2 = 4.7;

[addfloatsService addfloats:self action:@selector(handleAddFloats:)
   parameters:parameters];
```

The SudzC proxies will send basic authentication headers automatically when a Web service issues a 401 Unauthorized response.

However, an out-of-the-box 9.3 SAS BI Web Services configuration does not issue a 401 response when unauthorized access is attempted for either REST or SOAP requests[5]. Instead, it will typically return a 500 Internal Server Error response indicating that it could not complete the request without errors. Therefore, you should preemptively send the HTTP basic authentication headers with your requests to SAS BI Web Service REST endpoints.

```
[request setValue:@"Basic Zm9vOmJhcg==" forHTTPHeaderField:@"Authorization"];
```

Although the SudzC proxies and libraries support HTTP basic authentication, it is rarely used with SOAP. Instead, the WS-Security standard describes how to securely and reliably provide credentials and also secure the SOAP payloads, too, on a variety of platforms.. While difficult, you can add support for WS-Security with heavy modification to the SudzC libraries and proxies.

CONCLUSION

Though this paper details all the ways in which you can communicate with SAS BI Web Services 9.3, it barely scratches the surface with what is possible when writing applications for iOS. Apple ships numerous other UI elements besides the scant few detailed here. When writing your own custom iOS applications you will most certainly not use a simple UIView and UILabel to show the user information. Instead, you will leverage the various UIKit controllers to provide an intuitive navigation hierarchy for the user to make his way through your program. Instead of simply invoking a single Web service each time the application launches, your programs will rely on user interaction to drive background service invocations. And instead of simply displaying the results of Web service invocations, you will use CoreData to store and retrieve objects locally and sync them using SAS Web services running against your company's ETL Stored Processes. Of course, you'll use CoreAnimation to provide spiffy transitions and effects for your users (without overdoing it!).

To accomplish all of this, you'll need to do a lot more reading. Because of the massive popularity for third-party iOS applications, there are literally thousands of resources for beginner, intermediate, and advanced iOS programmers. Blog articles, books, screencasts, tutorials, and mountains of SDK documentation await your perusal. Once you decide who you'll be writing custom iOS applications for in your company and what purpose these applications will serve, you can decide how to continue learning about iOS. Explore open source project repositories such as github (https://github.com/explore), SourceForge (http://sourceforge.net/), and Gitorious (http://gitorious.org/) to find great examples of iOS code to learn from or reuse in your own projects. Use the great community-powered, question-and-answer site StackOverflow (http://stackoverflow.com/) to find answers to your questions about iOS and Web services or to answer others' questions once you're an expert. And because of the similarities between the Cocoa and Cocoa Touch framework collections, once you learn how to write iOS applications you can easily transition to native Mac OS X applications (after all, its market share is climbing, too).

Writing custom iOS applications that integrate with the SAS Platform enables you to reuse portable units of SAS code in the form of stored processes while providing a unique handheld experience for consuming complex analytics. SAS BI Web Services makes this easier than ever in version 9.3. By removing the need to run a Web service generation wizard and providing RESTful access to Web services, you can get started writing applications quicker than ever whether you're using plain XML or SOAP. But when you use plain XML Web service messaging with BI Web Services, you can easily write an iOS client using nothing but the built-in Apple frameworks. BI Web Services continues to support SOAP messaging, both when using the Web service generation wizard and when accessing SOAP endpoints using RESTful URLs. This unique combination of features in SAS BI Web Services 9.3 enables you to decide exactly how to implement your custom enterprise iOS application.

[5] You can configure your application server to use container authentication that typically triggers a 401 Unauthorized response. Refer to the documentation for your application server and the SAS Platform Administration Guide for more information.

REFERENCES

Apple Developer Portal. “iOS Dev Center.” 11 Jan 2011. Available at http://developer.apple.com/devcenter/ios/index.action

Apple Press Release. “Apple’s iOS 4.2 Available Today for iPad, iPhone & iPod touch.” 11 Jan 2011. Available at http://www.apple.com/pr/library/2010/11/22ios.html

ACKNOWLEDGEMENTS

The author would like to thank Tony Dean for reviewing this paper.

RECOMMENDED READING

“Getting Started with iOS,” Apple. Available at http://developer.apple.com/library/ios/#referencelibrary/GettingStarted/GS_iPhoneGeneral/.

CocoaDev Wiki. Available at http://cocoadev.com/.

SAS BI Web Services Developers Guide. Available at http://support.sas.com/documentation/cdl/en/wbsvcdg/61496/HTML/default/wbsvcdgwhatsnew902.htm.

Stanford University. “CS 193P iPhone Application Development.” Available at http://www.stanford.edu/class/cs193p/cgi-bin/drupal/.

Stevenson, Scott. “Learn Objective-C.” Available at http://www.cocoadevcentral.com/d/learn_objectivec/.

Stevenson, Scott. “Cocoa Style for Objective-C: Part I.” Available at http://www.cocoadevcentral.com/articles/000082.php.

Stevenson, Scott. “Cocoa Style for Objective-C: Part II.” Available at http://www.cocoadevcentral.com/articles/000083.php.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Zachary Marshall
SAS Institute Inc.
SAS Campus Dr.
Cary, NC 27513

E-mail: Zachary.Marshall@sas.com