finPOWER Connect 3
Custom Web Services Programming Guide

Version 3.02

25th May 2020

Table of Contents

Disclaimer 4
Version History 5
Introduction 6
Overview 7
Dates 8
  Returning Dates from Custom Web Services 8
  JavaScript 8
  JavaScript Issues 8
    Parsing JSON Objects 8
    Serialising to a JSON Object 8
Creating a Custom Web Service Script 10
  Calling the Custom Web Service 11
  Testing the Custom Web Service 12
  Debugging the Custom Web Service 13
  Accepting Parameters 14
  Controlling the Response 16
    HTML Response 16
    JSON Response 16
    Text Response 16
    XML Response 16
    PDF Response 17
    Docx Response 17
  Accepting Posted Data 18
  Serialising the Response 20
    Nullable Types 21
    Dates 23
    Enums 25
    Using Attributes to Tweak Serialisation 26
    Preventing Properties from Being Serialised 27
    Serialising Collections 29
  Deserialising the Request 31
    Deserialising XML 31
    Deserialising JSON 33
    Testing Posted XML 34
    Enums 35
    Using Attributes to Tweak Deserialisation 36
    Deserialising Collections 37
    Troubleshooting Deserialisation Issues 39
  Parsing Posted XML Request 40
  Parsing Posted JSON Request 42
  Manually Creating JSON Text 43

Documents, Emails and SMS Messages 45
Important 45
Limitations 45
Documents 46
Script-Type Documents 48
Ad-Hoc Documents 50
Word Document stored in an Account Log and returned as PDF 51
Word Document returned as PDF 53
PDF Document from HTML 55
PDF Document from HTML, Base-64 Encoded in Complex Response 56
Appendix A – Guidelines 58
Formatting HTML for Generating a PDF Document 58

Disclaimer

This document contains information that may be subject to change at any stage.

All code examples are provided "as is".

Version History

Date	Version	Name	Changes
14/04/2015	1.00	PH	Created.
13/07/2015	1.01	PH	Added manual JSON creating/ parsing sections.
22/02/2016	3.00	PH	Updated for finPOWER Connect version 3.
29/1/2020	3.01	PH	Reviewed and updated.
25/05/2020	**3.02**	**PH**	**Reviewed and updated for 3.03.02.**

Introduction

This document is intended for software developers who would like to create custom finPOWER Connect Web Services.

For information on programming non-custom Web Services (and for further reference), see the **finPOWER Connect 3 Web Services Connectivity and Programming Guide** document.

For information on installing and configuring the finPOWER Connect Web Services on a Web Server, see the **finPOWER Connect 2 Web Services Installation and Configuration** document.

All code samples are currently limited to Visual Basic (VB.NET).

Overview

Custom Web Services are implemented via finPOWER Connect Scripts.

Although General, Summary Page (version 2) and Web Summary Page type Scripts can all be used as Custom Web Services, this document focuses mainly on **Web Service (Web API)** type Scripts.

NOTE: HTML Widgets allow both User Interface and server-side code to be written and may be a more suitable solution than custom Web Services.

See the **finPOWER Connect 3 HTML Widgets** document for more information.

Custom Web Services can be written where either there is no Web Service currently available to perform a specific task or, a totally custom task is required, e.g., creating a Quote Account using custom information entered on a Web page (as per the CustomLoanQuote1VB.aspx sample detailed in the **Visual Basic Code Examples** section of the **finPOWER Connect Web Services Connectivity and Programming Guide** document).

NOTE: Always check with Intersoft Systems before choosing to write a Custom Web Service if the task you want to perform could be considered standard finPOWER Connect functionality.

Dates

Web Service dates use the ISO 8601 standard regardless of whether the date is passed as part of a request URL or is included in an XML or JSON response.

Web Services return dates in UTC (Coordinated Universal Time) regardless of how they are stored within the finPOWER Connect database or displayed in the finPOWER Connect Windows interface.

WARNING: Any dates provided as parameters to Web Services will be parsed according to the time zone on the Web Server. Therefore, it is advisable to always provide dates in UTC format.

Dates are covered in detail in the **Dates** topic of the Web Services API reference. Much of this is repeated below.

Returning Dates from Custom Web Services

TODO:

JavaScript

In JavaScript, when deserialising a JSON object, dates are treated as strings and are not automatically converted to dates. This is explained fully in this article by Rick Strahl:

http://weblog.west-wind.com/posts/2014/Jan/06/JavaScript-JSON-Date-Parsing-and-real-Dates

JavaScript Issues

Communication with the finPOWER Connect Web Services would typically not come directly from JavaScript. JavaScript may be used as a method of transferring data to and from a Web browser to a Web application that then communicates with the finPOWER Connect Web Services. Either way, this section may still be useful in highlighting issues or potential issues when using JavaScript dates.

JavaScript stores dates in UTC format and formats these according to the host computer's time zone when displaying a value.

When attempting to parse an ISO 8601 date string that is not specified as a UTC date, i.e., it does not end in **Z**, different browsers may give different results, therefore you may not be able to reliably use the `Date.parse` method.

When serialising a date to a JSON string via either the `dateObject.toJSON` or `JSON.stringify` functions, the date will be formatted according to the ISO 8601 specification and will always result in a UTC date format. For example, serialising a date of birth of 3rd November 1968 will result in the following:

```
1968-11-03T00:00:00.000Z
```

Parsing JSON Objects

Dates returned from Web Services are always returned in UTC format, e.g., a date of birth of **1/1/2001** will be returned as **"2001-01-01T00:00:00Z"**.

This means that time zone issues should not occur when displaying the date from Javascript; the date (providing no time portion is included) will always be displayed properly, e.g., **1/1/2001**.

Serialising to a JSON Object

Consider a date of birth of **1/1/2001** entered on an HTML page on a computer in New Zealand.

If you parse this date normally via JavaSript:

```
window.alert(new Date("2001/1/1"));
```

The result is **Mon Jan 01 2001 00:00:00 GMT+1300 (New Zealand Daylight Time)**.

If however, you serialise this date using `JSON.stringify`:

```
window.alert(JSON.stringify(new Date("2001/1/1")));
```

The result is **"2000-12-31T11:00:00.000Z"**.

This is a hugely confusing issue, especially when dealing with specific dates such as a date of birth.

The first date is formatted with the time zone offset (GMT+1300), the second is formatted as straight UTC and no time zone, thereby showing the date before.

Both of these dates refer to the same point in time but, it is not the point in time you want, you want a date that is 13 hours ahead of this!

What you really want is a UTC date that is the current date but contains no time zone information.

One trick is to append the time zone difference (i.e., add 13 hours) to the date before it is serialised, e.g.:

```
// Return a date with a time zone difference appended so the date will always be correct
function GetDateOnly(value) {
  if (value instanceof Date) {
    var sd = value.getDate();
    var sm = value.getMonth() + 1;
    if (sd < 10) sd = "0" + sd;
    if (sm < 10) sm = "0" + sm;
    return new Date(value.getFullYear() + "-" + sm + "-" + sd + "T00:00:00Z")
  }
  else {
    return value;
  }
}
```

Now, if you do:

```
window.alert(GetDateExact(new Date("2001/1/1")));
```

The result is **Mon Jan 01 2001 13:00:00 GMT+1300 (New Zealand Daylight Time)**.

And, if you then serialise this date using `JSON.stringify`:

```
window.alert(JSON.stringify(GetDateExact(new Date("2001/1/1"))));
```

The result is now **"2001-01-01T00:00:00.000Z"**.

Creating a Custom Web Service Script

This section outlines the steps required to create a simple custom Web Service Script.

- Open finPOWER Connect.
- Open the database being used by the Web Services.
- From the **Admin** menu, select **Scripts**.
- Click the **Add** button.
- Give the Script an Code and Description, e.g.:
  - Code: TESTWS
    - **NOTE:** By default, the Script's code is the name of the custom Web Service and should therefore contain only alphanumeric characters (this can be overridden on the Web page of the Scripts form).
  - Description: Custom Web Service Test
- Specify a Script Type of **Web Service (Web API)**.
- On the **Script Code** page, click the **Paste template Script code** button.
- Click the **Save** button to save the Script.

Calling the Custom Web Service

The custom Web Service created above can now be called just like any other Web Service using the special **Custom** controller, e.g.:

```
http://localhost/finPOWERConnectWS2/Api/Custom/TESTWS
```

In this case, a Script named **TESTWS** will be called or, if a Script has a Web Service Name of 'TESTWS' defined on the Web page of the Scripts form in finPOWER Connect, this Script will be called (i.e., the Script's code is used by default but can be overridden by specifying a Web Service Name).

By default, you must have already authenticated and therefore include the Authorization HTTP header as detailed in **The Authentication Process** section of the **finPOWER Connect Web Services Connectivity and Programming Guide document**.

WARNING: The Web page of the Scripts form has a checkbox to 'Allow Anonymous access'.

Use this option with extreme caution since it allows your custom Web Service to be called without having first authenticated.

Testing the Custom Web Service

You can test your custom Web Service from the **Test Web Services** form in finPOWER Connect as follows:

- Connect to the Web Services from the **Connect** page.
- Switch to the **Web Services** page.
- Select the **Custom**, **ExecuteGet** node in the Web Services explorer.
- Enter the Id of your Script, e.g., TESTWS
- Click the **Test** button.
  - You should see a Response of:

```
<string>A simple text response (that will be serialised as XML or JSON), e.g., as per the Ping service.</string>
```

- Switch to the **Connect** page and change the Format to **JSON**.
- Switch to the **Web Services** page and click the **Test** button again.
  - You should now see a Response of:

```
"A simple text response (that will be serialised as XML or JSON), e.g., as per the Ping service."
```

NOTE: Custom Web Services can be called using either the GET or POST HTTP methods via the ExecuteGet and ExecutePost nodes in the Web Services explorer.

GET-based Web Services receive all of their parameters from the URL; POST-based services can receive parameters from the URL and also from the 'Posted' RequestText, e.g., the RequestText might be XML formatted Client information.

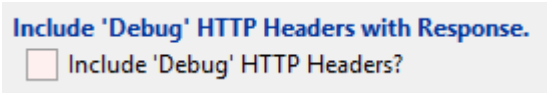
Debugging the Custom Web Service

Generally, when writing Script code within finPOWER Connect, the Script can contain `finBL.DebugPrint` statements which can then be viewed in either the Debug Window or, if running via the 'Test' button on the Scripts form, the Debug page.

When a Custom Web Service Script is running however, it is running on a Web Server and not within the Windows version of finPOWER Connect. Therefore, any `finBL.DebugPrint` statements will be ignored.

However, the Script can be configured to send any debug information back with the Web Service's HTTP Response.

A checkbox on the Web page of the Scripts form allows this feature to be enabled:



When enabled, the Script's HTTP Response will then contain one or more DEBUG headers for each `finBL.DebugPrint` encountered whilst the Script was ruinning, e.g.:

```
HTTP/1.1 400 Bad Request
Pragma: no-cache
WS-User: webadmin
Debug-1: Client failed to load.
Debug-2-0: Failed to load Client 'ABCDEFG'.
Debug-2-1:
Debug-2-2: Record not found.
```

The following example demonstrates how the above debug information was written:

```
Client = finBL.CreateClient()

If Not Client.Load("ABCDEFG") Then
  finBL.DebugPrint("Client failed to load.")
  finBL.DebugPrint(finBL.Error.Message(True))
End If
```

Accepting Parameters

We will now update the custom Web Service to accept parameters.

The Script will be updated to accept a **ClientId** parameter and will return the Client's name together with a list of all their aliases (Akas).

Update your custom Web Service Script code to the following:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean

  Dim Client As finClient
  Dim ClientAka As finClientAka
  Dim ClientDetails As clsClientDetails
  Dim ClientId As String

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Get Parameters
  ClientId = request.Parameters.GetString("ClientId")

  ' Load Client
  Client = finBL.CreateClient()
  If Client.Load(ClientId) Then
    ' Create and update Client Details
    ClientDetails = New clsClientDetails()
    ClientDetails.Name = Client.Name

    ' List AKAs
    For Each ClientAka In Client.Akas
      ClientDetails.Akas.Add(ClientAka.NameFull)
    Next
  Else
    Ok = False
  End If

  ' Response
  If Ok Then
    Return request.CreateResponse(HttpStatusCode.OK, ClientDetails)
  End If

  ' Error
  If Not Ok Then
    Return request.CreateErrorResponse(ErrorStatusCode, "Failed to get Client name details.",
ErrorCode, finBL.Error.Message(True, True))
  End If

End Function

<System.Xml.Serialization.XmlType("ClientDetails")>
Public Class clsClientDetails

  Public Name As String
  Public Akas As New List(Of String)

End Class
```

We will now test the updated service using the Test Web Service form.

- Select the **Custom**, **Execute (GET)** node in the Web Services explorer.
- Enter the Id of your Script, e.g., TESTWS
- In the **Parameters** field, enter `clientId=C10000`.
  - Parameters must be URL encoded and separated by a '&', e.g., if you had two parameters, you would specify them as param1=value1&param2=value2
- Click the **Test** button.

- You should see a Response similar to:

```
<ClientDetails xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Smith, John</Name>
  <Akas>
    <string>Jonny Smith</string>
    <string>Butch</string>
  </Akas>
</ClientDetails>
```

- Switch to the **Connect** page and change the Format to **JSON**.
- Switch to the **Web Services** page and click the **Test** button again.
  - You should now see a Response similar to:

```
{"Name":"Smith, John",
"Akas":["Jonny Smith",
"Butch"]}
```

NOTE: As of version 2.02.01 of finPOWER Connect, Custom Web Service Scripts can define Parameters which are then used by the Test Web Service form when selecting the **Custom**, **Execute (Parameters)** node in the Web Services explorer.

This provides a more user-friendly way of testing Custom Web Services that use Parameters.

In the above example, adding a Text or Range parameter named 'ClientId' to the Script would allow the Custom Web Service to be tested in this way.

Controlling the Response

Note that in the above example, the XML response has a root node of `<ClientDetails>`. This is because we applied an `XmlType` attribute to the `clsClientDetails` class. Without this, the root node will have been named `<clsClientDetails>`.

Also, because we have declared the `Akas` property as a `List(Of String)`, we do not have any control over how this is serialised. Using a custom collection would allow more control.

Our custom Web Service switches automatically between XML and JSON serialisation due to the way we are returning the response as an object, i.e.:

```
Return request.CreateResponse(HttpStatusCode.OK, ClientDetails)
```

Alternatively, to return an error response:

```
Return request.CreateErrorResponse(ErrorStatusCode, "Failed to add Client.", ErrorCode,
finBL.Error.Message(True, True))
```

For most services, this is probably desirable but, using the various 'Create' methods of the request object we can force the response to be a particular format as follows:

HTML Response

`CreateHtmlResponse` allows us to force the response to be HTML, e.g.

```
Return request.CreateHtmlResponse(HttpStatusCode.OK, "<h1>This is HTML</h1>")
```

This simply sets the HTTP Content-Type header to **text/html**.

JSON Response

`CreateJsonResponse` allows us to force the response to be JSON, e.g.

```
Return request.CreateJsonResponse(HttpStatusCode.OK, "{FirstName: "John", LastName: "Smith"}")
```

This simply sets the HTTP Content-Type header to **application/json**.

NOTE: See the Manually Creating JSON Text section for information on using the JsonBuilder business layer object which was added in finPOWER Connect 2.03.00.

Text Response

`CreateTextResponse` allows us to force the response to be plain text, e.g.

```
Return request.CreateTextResponse(HttpStatusCode.OK, Account.AccountId)
```

This simply sets the HTTP Content-Type header to **text/plain**.

NOTE: If a Web Service is designed to return only a very simple value, e.g., the Id of an Account, returning it as plain text means that any calling applications do not need to parse XML or JSON and therefore simplifies the code.

XML Response

`CreateXmlResponse` allows us to force the response to be XML, e.g.

```
Return request.CreateXmlResponse(HttpStatusCode.OK, Branch.ToXmlString())
```

This simply sets the HTTP Content-Type header to **text/xml**.

PDF Response

`CreatePdfResponse` and `CreatePdfResponseFromHtml` allows us to force the response to be PDF document, e.g.

```
Return request.CreatePdfResponseFromHtml(HttpStatusCode.OK, "<h1>My PDF document</h1>")
```

Or:

```
Dim PdfData() As Byte

If finBL.PdfUtilities.CreatePdfByteArrayFromHtml("<h1>My PDF document</h1>", PdfData) Then
  Return request.CreatePdfResponse(HttpStatusCode.Ok, PdfData)
Else
  ' Error
End If
```

This sets the HTTP content to the binary PDF data and sets the Content-Type header to **application/pdf**.

See the PDF Documents section for more information.

Docx Response

`CreateDocxResponse` allows us to force the response to be a Microsoft Word (docx) document, e.g.

```
Dim DocxData() As Byte

' Populate DocxData, e.g., from the embedded file data on a finAccountLog

Return request.CreateDocxResponse(HttpStatusCode.Ok, DocxData)
```

This sets the HTTP content to the binary Docx data and sets the Content-Type header to **application/vnd.openxmlformats-officedocument.wordprocessingml.document**.

NOTE: The content type above may seem long-winded but is the official MIME type specified my Microsoft:

http://blogs.msdn.com/b/vsofficedeveloper/archive/2008/05/08/office-2007-open-xml-mime-types.aspx

Accepting Posted Data

Sometimes, you may wish to simply POST data directly to a custom Web Service, e.g., an XML document created from an external application.

The following Script will accept a POSTed piece of XML in the following format and create a Transaction for an Account. It will not return anything, just an HTTP Status code of 200 if successful:

```
<AccountPayment>
  <AccountId>L10035</AccountId>
  <Amount>35.00</Amount>
  <Reference>REF</Reference>
</AccountPayment>
```

Update your custom Web Service's Script code to the following:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean

  Dim AccountPayment As finAccountPayment
  Dim AccountPaymentDetails As clsAccountPaymentDetails
  Dim Obj As Object

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Parse XML (use business layer helper Function)
  If finBL.Runtime.WebUtilities.DeserialiseXmlStringToObject(request.RequestText,
GetType(clsAccountPaymentDetails), Obj) Then
    AccountPaymentDetails = DirectCast(Obj, clsAccountPaymentDetails)
  Else
    Ok = False
  End If

  ' Add Account Payment
  If Ok Then
    AccountPayment = finBL.CreateAccountPayment()
    With AccountPayment
      ' Load Account
      Ok = .AccountLoad(AccountPaymentDetails.AccountId)

      ' Update
      If Ok Then
        .TransactionTypeId = "PAY"
        .PaymentMethodId = "CASHR"
        .PaymentValue = AccountPaymentDetails.Amount
        .TransactionReference = AccountPaymentDetails.Reference
      End If

      ' Commit
      If Ok Then
        Ok = .ExecuteCommit()
      End If
    End With
  End If

  ' Response
  If Ok Then
    Return request.CreateResponse(HttpStatusCode.OK)
  End If

  ' Error
  If Not Ok Then
    Return request.CreateErrorResponse(ErrorStatusCode, "Failed to make Account payment.",
ErrorCode, finBL.Error.Message(True, True))
  End If

End Function

<System.Xml.Serialization.XmlType("AccountPayment")>
Public Class clsAccountPaymentDetails
```

```
  Public AccountId As String
  Public Amount As Decimal
  Public Reference As String

End Class
```

We will now test the updated service using the Test Web Service form.

- Select the **Custom**, **Execute (POST)** node in the Web Services explorer.
- Enter the Id of your Script, e.g., TESTWS
- In the **Request Text** field, past the XML shown at the start of this section.
- Click the **Test** button.
  - You should receive a Response that contains nothing, just an HTTP Status code of 200.

NOTE: The Deserialising the Request and Parsing Posted XML Request sections cover handling Posted data in more detail.

Serialising the Response

As shown in some of the previous samples in this section, serialisation to either XML or JSON is handled automatically by the Web API and the .NET framework.

This section has examples of automatic serialisation of the Response from Script objects.

WARNING: Never attempt to return built-in finPOWER Connect business layer objects from a Custom Web Service. These objects are not designed for serialisation.

NOTE: Many of the examples in this section remove the following attributes from the response's root XML node for clarity:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

Nullable Types

.NET nullable types are not used within the finPOWER Connect business layer however, they can, and often should, be used when returning the results of a Web Service.

For instance if you attempt to return an object that has a Date property, e.g., `DateOfBirth`, and this does not have a value, it will be returned as `0001-01-01T00:00:00` since the .NET Date type cannot have no value (within .NET, the `Date = Nothing` assignment and comparison is actually assigning and comparing the date with `0001-01-01T00:00:00`).

For example, the following Script:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim ClientDetails As ClientDetails

  ' Create Object
  ClientDetails = New ClientDetails()
  With ClientDetails
    .Name = "Paul"
  End With

  ' Return
  Return request.CreateResponse(HttpStatusCode.OK, ClientDetails)

End Function

Public Class ClientDetails

  Public Name As String
  Public DateOfBirth As Date

End Class
```

Returns:

```
<ClientDetails>
  <Name>Paul</Name>
  <DateOfBirth>0001-01-01T00:00:00</DateOfBirth>
</ClientDetails>
```

If this is modified to use a nullable data type, e.g.:

```
Public Class ClientDetails

  Public Name As String
  Public DateOfBirth As Date?

End Class
```

The response is much more intuitive:

```
<ClientDetails>
  <Name>Paul</Name>
  <DateOfBirth xsi:nil="true" />
</ClientDetails>
```

Nullable types would typically be used for the following:

- Dates
  - Only where the date is optional, e.g., a Date of Birth.
- Numeric types, e.g., Decimal, Double and Integer
  - Only where returning zero does not make sense.
- Boolean values
  - Only where returning `True` or `False` does not make sense.

NOTE: Strings cannot be nullable since the String data type in the .NET framework is an object and only value types can be nullable.

WARNING: Returning nullable types does add an extra level of complexity to the application having to parse the XML and you should consider not including the elements in the response at all as described in the Preventing Properties From Being Serialised section.

Dates

Dates are serialised using the ISO 8601 standard as detailed in the Dates section of the API Reference.

All built-in Web Services return dates containing a time portion in UTC format, regardless of how they are stored within finPOWER Connect.

Unless you have a requirement to explicitly ignore this rule, all dates containing a time portion that are returned from a Custom Web Service, e.g., Log dates, should be returned as UTC dates.

WARNING: .NET Dates have a `Kind` property which may affect how the date is serialised. Therefore, when setting dates on objects that are to be serialised, it is important to bear this in mind, e.g., if `Kind` is `DateTimeKind.Local`, serialising a date containing a time portion may not produce the result you expect, e.g., the date may be converted to a UTC date when this is not expected.

When testing a Custom Web Service, if dates and times are involved, ensure that you test the service on a Web Server running in the time zone that the production server will be using to ensure there are no unforeseen issues.

finPOWER Connect contains business layer functionality to convert a dates to nullable, UTC format dates, e.g.:

```
' Create Object
ClientDetails = New ClientDetails()
With ClientDetails
  .Name = Client.Name
  .DateOfBirth = finBL.Runtime.DateUtilities.CastToNullableUtcDate(Client.DateOfBirth)
  If LastLog IsNot Nothing Then
    .LastLogDate = finBL.ToUniversalTime(LastLog.Date)
  End If
End With
```

Produces the following response:

```
<ClientDetails>
  <Name>Smith, John</Name>
  <DateOfBirth>1978-02-04T00:00:00Z</DateOfBirth>
  <LastLogDate>2014-08-07T03:26:11Z</LastLogDate>
</ClientDetails>
```

Note that both dates are returned as UTC dates even though the Date of Birth does not contain a time portion.

The following helper functions are available in finPOWER Connect business layer:

- `finBL.ToUniversalTime(value)`
  - Converts a date recorded in local time (as Log dates are in finPOWER Connect 2) to UTC time.
  - The time will be adjusted from local time to UTC time.
- `finBL.Runtime.DateUtilities`
  - `CastToNullableUtcDate(value)`
    - Casts a date containing no time portion (E.g., a Date of Birth) to a nullable UTC date.
    - If the Date = Nothing then this will be handled correctly.
    - Any time portion is removed.
  - `CastToNullableUtcDateTime(value)`
    - Casts a date, optionally containing a time portion to a nullable UTC date.
    - If the Date = Nothing then this will be handled correctly.

- Does not 'convert' the date to UTC, i.e., no time adjustment is made so this is not suitable for converting local dates to UTC dates.

Enums

Enums are serialised as String values, e.g., `isefinAccountStatus.ClosedPending` is serialised to 'ClosedPending' using a method such as `finAccount.Status.ToString()`.

finPOWER Connect generally has an Enum value, e.g., `isefinAccountStatus.ClosedPending` and also a display value, e.g., 'Closed (Pending)'. This is not the same as the serialised version of the Enum ('ClosedPending') although often the two are the same.

WARNING: Do not confuse the finPOWER Connect display value returned from methods such as `finBL.Enums.isefinAccountStatus_ToString()` methods with the actual Enum values.

These methods are used for displaying Enum values in a user-readable form, e.g., including spaces or varying the display value based upon the database country.

When returning an object that contains an Enum value, it may be desirable to return both the Enum and the display value. Typically, the property holding the display value is named the same as the Enum value property but with a 'Text' suffix, e.g.:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Account As finAccount
  Dim AccountDetails As AccountDetails

  ' Load (ignore errors for simplicity)
  Account = finBL.CreateAccount()
  Account.Load("L10000")

  ' Create Object
  AccountDetails = New AccountDetails()
  With AccountDetails
    .Name = Account.Name
    .Status = Account.Status
    .StatusText = Account.StatusText
  End With

  ' Return
  Return request.CreateResponse(HttpStatusCode.OK, AccountDetails)

End Function

Public Class AccountDetails

  Public Name As String
  Public Status As isefinAccountStatus
  Public StatusText As String

End Class
```

Produces the following response:

```
<AccountDetails>
  <Name>Smith, John</Name>
  <Status>ClosedPending</Status>
  <StatusText>Closed (Pending)</StatusText>
</AccountDetails>
```

Using Attributes to Tweak Serialisation

You may wish to serialise classes using an XML element named differently from the class name.

This is achieved by applying the `XmlType` or `XmlRoot` attributes to the class, e.g.:

```
<System.Xml.Serialization.XmlType("Transaction")>
Public Class finwsAccountTransaction

  ' Properties
  Public [Date] As Date
  Public ElementId As String
  Public Reference As String
  Public Value As Decimal

End Class

< System.Xml.Serialization.XmlRoot("Transactions")>
Public Class finwsAccountTransactions : Inherits List(Of finwsAccountTransaction)
End Class

< System.Xml.Serialization.XmlType("AvailableCredit")>
Public Class finwsAccountAvailableCreditDetails

  Public AvailableCredit1 As Decimal
  Public AvailableCredit2 As Decimal
  Public AvailableCredit3 As Decimal
  Public CreditLimit1 As Decimal
  Public CreditLimit2 As Decimal
  Public CreditLimit3 As Decimal

End Class
```

NOTE: Both of these attributes work in a similar way in that they determine the name of the XML element containing their properties.

Preventing Properties from Being Serialised

Sometimes, you may wish to define properties on your object but not have them serialised under certain circumstances.

For example, you may have a `ClientDetails` class that contains properties that should only be serialised for 'Individual' type Clients, e.g., `DateOfBirth`.

NOTE: These `ShouldSerialize` methods are applied when serialising as XML or JSON and are built-in to the .NET framework.

The method MUST be named after the property name, e.g., to control whether a `DateOfBirth` property is serialised, you must have a method named `ShouldSerializeDateOfBirth()` that returns a Boolean value.

Also note the U.S. spelling of the word 'serialize'.

The following example shows how to achieve this using special `ShouldSerialize` methods which are used by the .NET serialisation process to decide whether or not to serialise a property:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Client As finClient
  Dim ClientDetails As ClientDetails

  ' Load (ignore errors for simplicity)
  Client = finBL.CreateClient()
  Client.Load("C10000")

  ' Create Object
  ClientDetails = New ClientDetails(Client)
  With ClientDetails
    .Name = Client.Name
    .DateOfBirth = finBL.Runtime.DateUtilities.CastToNullableUtcDate(Client.DateOfBirth)
    .OrganisationNumber = Client.OrganisationNumber
  End With

  ' Return
  Return request.CreateResponse(HttpStatusCode.OK, ClientDetails)

End Function

Public Class ClientDetails

  ' Properties
  Public Name As String
  Public DateOfBirth As Date?
  Public OrganisationNumber As String

  ' Other
  Private mClient As finClient

  Public Sub New(client As finClient)

    mClient = client

  End Sub

  Public Sub New()
  End Sub

  ' Prevent Serialisation for Not Applicable Properties
  Public Function ShouldSerializeDateOfBirth() As Boolean
    Return mClient Is Nothing OrElse mClient.IsIndividual OrElse mClient.IsSoleTrader
  End Function

  Public Function ShouldSerializeOrganisationNumber() As Boolean
    Return mClient Is Nothing OrElse mClient.IsOrganisation
  End Function

End Class
```

Produces the following response for an 'Individual' Client:

```
<ClientDetails>
  <Name>Smith, John</Name>
  <DateOfBirth>1978-02-04T00:00:00Z</DateOfBirth>
</ClientDetails>
```

And the following response for an 'Organisation' Client:

```
<ClientDetails>
  <Name>Company Limited</Name>
  <OrganisationNumber>1234567890</OrganisationNumber>
</ClientDetails>
```

WARNING: Serialisable classes must always have an empty, public constructor which is why the above example has two constructors.

This is also why the test `mClient Is Nothing` is performed since theoretically, a `ClientDetails` object could be created without passing in a `finClient` object.

Serialising Collections

All of the above examples have dealt with serialising a single-level object.

The following example shows how to return a collection object:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Account As finAccount
  Dim AccountTransaction As finAccountTransaction
  Dim AccountDetails As AccountDetails
  Dim Transaction As Transaction

  ' Load (ignore errors for simplicity)
  Account = finBL.CreateAccount()
  Account.Load("L10000")

  ' Create Object
  AccountDetails = New AccountDetails()
  With AccountDetails
    .AccountId = Account.AccountId
    .Name = Account.Name
    .Transactions = New List(Of Transaction)
  End With

  ' Add Transactions
  For Each AccountTransaction In Account.Transactions
    ' Create
    Transaction = New Transaction()

    ' Update
    With AccountTransaction
      Transaction.Date = finBL.Runtime.DateUtilities.CastToUtcDate(.Date)
      Transaction.ElementId = .ElementId
      Transaction.Value = .Value
    End With

    ' Add to Collection
    AccountDetails.Transactions.Add(Transaction)
  Next

  ' Return
  Return request.CreateResponse(HttpStatusCode.OK, AccountDetails)

End Function

Public Class AccountDetails

  Public AccountId As String
  Public Name As String
  Public Transactions As List(Of Transaction)

End Class

Public Class Transaction

  Public [Date] As Date
  Public ElementId As String
  Public Value As Decimal

End Class
```

Produces the following response:

```
<AccountDetails>
  <AccountId>L10000</AccountId>
  <Name>Smith, John</Name>
  <Transactions>
    <Transaction>
      <Date>2014-06-08T00:00:00Z</Date>
      <ElementId>ADV</ElementId>
      <Value>10000</Value>
    </Transaction>
     <Transaction>
      <Date>2014-06-09T00:00:00Z</Date>
      <ElementId>ACCF</ElementId>
      <Value>123</Value>
    </Transaction>
    <Transaction>
```

```
      <Date>2014-07-09T00:00:00Z</Date>
      <ElementId>ACCF</ElementId>
      <Value>2.22</Value>
    </Transaction>
    <Transaction>
      <Date>2014-07-09T00:00:00Z</Date>
      <ElementId>PAY</ElementId>
      <Value>-4.22</Value>
    </Transaction>
  </Transactions>
</AccountDetails>
```

Deserialising the Request

As shown in the Accepting Posted Data section, it is often desirable to deserialise XML or JSON posted to the Custom Web Service into an object that the Script can use.

Deserialising XML

The finPOWER Connect business layer provides functionality to deserialise XML into an object as shown in the following example:

WARNING: Deserialisation is useful for small, flat objects or simple collections. For complex XML it may be more suitable to parse the posted XML directly.

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean

  Dim Client As finClient
  Dim ClientDetails As ClientDetails
  Dim Obj As Object

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Parse XML (use business layer helper Function)
  If finBL.Runtime.WebUtilities.DeserialiseXmlStringToObject(request.RequestText,
GetType(ClientDetails), Obj) Then
    ClientDetails = DirectCast(Obj, ClientDetails)
  Else
    Ok = False
  End If

  ' Add Client
  If Ok Then
    Client = finBL.CreateClient()
    With Client
      ' Update
      .FirstName = ClientDetails.FirstName
      .LastName = ClientDetails.LastName
      .DateOfBirth = ClientDetails.DateOfBirth

      ' Save
      Ok = .Save()
    End With
  End If

  ' Return Response
  If Ok Then
    Return request.CreateResponse(HttpStatusCode.OK, Client.ClientId)
  Else
    Return request.CreateErrorResponse(ErrorStatusCode, "Failed to add Client.", ErrorCode,
finBL.Error.Message(True, True))
  End If

End Function

Public Class ClientDetails

  Public FirstName As String
  Public LastName As String
  Public DateOfBirth As Date

End Class
```

The following XML can be posted to the above Script:

```
<ClientDetails>
  <FirstName>Paul</FirstName>
  <LastName>Jones</LastName>
```

```
  <DateOfBirth>1986-03-17</DateOfBirth>
</ClientDetails>
```

Points of note in the above example are:

- Use of the `DeserialiseXmlStringToObject()` helper method.
  - Internally, this uses the .NET framework's deserialisation functionality.
- `ClientDetails.DateOfBirth` is defined as a date.
  - This means that if the XML omits this tag, the property will remain as the default value for a Date which equals `Nothing`, therefore there is no special handling required.
  - This is a Date with no time portion hence there is no need to convert from a UTC date.
  - Supplying an empty `DateOfBirth` element in the XML, e.g.:

    ```
    <DateOfBirth/>
    ```

    Will cause a deserialisation error. Either omit the element completely or use the following format:

    ```
    <ClientDetails xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <FirstName>Paul</FirstName>
      <LastName>Jones</LastName>
      <DateOfBirth xsi:nil="true" />
    </ClientDetails>
    ```

    And change the `DateOfBirth` property to a nullable type:

    ```
    Public Class ClientDetails
      Public FirstName As String
      Public LastName As String
      Public DateOfBirth As Date?
    End Class
    ```

    And handle accordingly when updating the `finClient` object, e.g.:

    ```
    If ClientDetails.DateOfBirth IsNot Nothing Then
      .DateOfBirth = CDate(ClientDetails.DateOfBirth)
    End If
    ```

Deserialising JSON

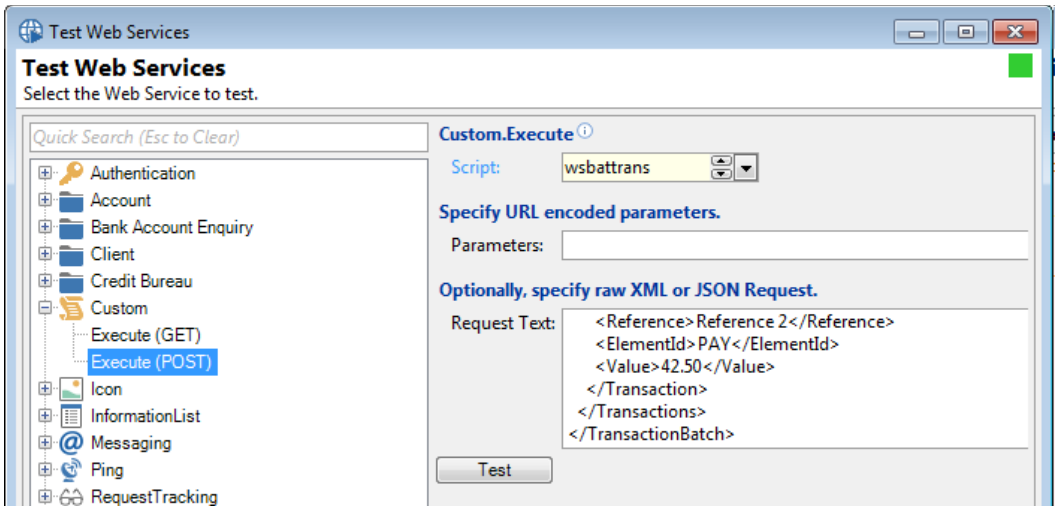
Deserialising JSON text passed to a custom Web Service is done in exactly the same way as XML (as discussed in the previous section). The only difference is that the **`DeserialiseJsonStringToObject`** method is used instead of the **`DeserialiseXmlStringToObject`** method.

WARNING: Deserialisation is useful for small, flat objects or simple collections. For complex XML it may be more suitable to parse the posted JSON directly.

Testing Posted XML

Testing a Custom Web Service that accepts posted XML can be achieved from the Test Web Services form.

Simply paste a sample of the XML into the 'Request Text' field, e.g.:



Enums

Enums are included in XML using their String representation, e.g., `isefinAccountStatus.ClosedPending` would be serialised using a method such as `finAccount.Status.ToString()`. This would produce a value of 'ClosedPending'.

WARNING: Do not confuse this with the finPOWER Connect display value for the Enum returned from methods such as `finBL.Enums.isefinAccountStatus_ToString()` methods with the actual Enum value.

These methods are used for displaying Enum values in a user-readable form, e.g., including spaces or varying the display value based upon the database country.

Deserialisation of Enums is handed internally by the .NET framework however, it is not recommended that you use nullable types for Enums since this complicates the deserialisation process. Therefore, do not include an Enum tag in the XML if it is not required.

Using Attributes to Tweak Deserialisation

By default, XML will be deserialised into properties that match the XML element names.

This can be tweaked using attributes, e.g.:

```
<System.Xml.Serialization.XmlType("TransactionBatch")>
Public Class clsTransactionBatch

  Public BatchId As String
  Public TransactionType As String
  Public Transactions As List(Of clsTransaction)

End Class

<System.Xml.Serialization.XmlType("Transaction")>
Public Class clsTransaction

  Public AccountId As String
  Public [Date] As Date
  Public Reference As String
  Public ElementId As String
  Public Value As Decimal

End Class
```

Deserialising Collections

Collections can also be deserialied as shown in the following example which creates a Batch of Transactions and saves and then commits the Batch:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Batch As finBatch
  Dim BatchTransaction As finBatchTransaction
  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Obj As Object
  Dim Ok As Boolean
  Dim Transaction As clsTransaction
  Dim TransactionBatch As clsTransactionBatch

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Deserialise the XML to an Object
  If finBL.Runtime.WebUtilities.DeserialiseXmlStringToObject(request.RequestText,
GetType(clsTransactionBatch), Obj) Then
    TransactionBatch = DirectCast(Obj, clsTransactionBatch)
  Else
    Ok = False
  End If

  If Ok Then
    ' Initialise
    Batch = finBL.CreateBatch()

    ' Setup Batch
    With Batch
      .BatchId = TransactionBatch.BatchId
      .TransactionTypeId = TransactionBatch.TransactionType
      .SourceSet(isefinTransactionSource.ExternalA)
    End With

    ' Add Transactions
    For Each Transaction In TransactionBatch.Transactions
      ' Create
      BatchTransaction = Batch.Transactions.CreateBatchTransaction()

      ' Update (ignore errors for simplicity of sample)
      With BatchTransaction
        .AccountIdSet(Transaction.AccountId)
        .ElementIdSet(Transaction.ElementId)
        .Date = Transaction.Date
        .Value = Transaction.Value
        .Reference = Transaction.Reference
      End With

      ' Add
      Batch.Transactions.Add(BatchTransaction)
    Next

    ' Save Batch
    Ok = Batch.Save()

    ' Commit Batch
    If Ok Then
      Ok = Batch.ExecuteCommit(False)
    End If
  End If

  ' Return Response
  If Ok Then
    Return request.CreateResponse(HttpStatusCode.OK)
  Else
    Return request.CreateErrorResponse(ErrorStatusCode, String.Format("Failed to execute custom Web
Service Script '{0}'.", ScriptInfo.ScriptId), ErrorCode, finBL.Error.Message(True, True))
  End If

End Function

<System.Xml.Serialization.XmlType("TransactionBatch")>
Public Class clsTransactionBatch
```

```
  Public BatchId As String
  Public TransactionType As String
  Public Transactions As List(Of clsTransaction)

End Class

<System.Xml.Serialization.XmlType("Transaction")>
Public Class clsTransaction

  Public AccountId As String
  Public [Date] As Date
  Public Reference As String
  Public ElementId As String
  Public Value As Decimal

End Class
```

The following XML can be posted to the above Script:

```
<TransactionBatch>
  <BatchId>WSBatch001</BatchId>
  <TransactionType>PAY</TransactionType>
  <Transactions>
    <Transaction>
      <AccountId>L10000</AccountId>
      <Date>2014-08-07</Date>
      <Reference>Reference 1</Reference>
      <ElementId>PAY</ElementId>
      <Value>100.00</Value>
    </Transaction>
    <Transaction>
      <AccountId>L10035</AccountId>
      <Date>2014-08-08</Date>
      <Reference>Reference 2</Reference>
      <ElementId>PAY</ElementId>
      <Value>42.50</Value>
    </Transaction>
  </Transactions>
</TransactionBatch>
```

Points of note in the above example are:

- Use of the `DeserialiseXmlStringToObject()` helper method.
  - Internally, this uses the .NET framework's deserialisation functionality.
  - This deserialises the `<Transactions>` block in to the `Transactions` property which is defined as a `List(Of clsTransaction)`.
- Attributes (E.g., `<System.Xml.Serialization.XmlType("Transaction")>`) are used to map the class names, e.g., `clsTrasaction` to an XML element name, e.g., `Transaction`.

Troubleshooting Deserialisation Issues

Problems can arise when attempting to deserialise XML into an object and these are often hard to track.

The `DeserialiseXmlStringToObject()` method will return `False` if deserialisation failed. This may be due to the following:

- The XML being deserialised was invalid.
- Dates or numbers within the XML are invalid.
  - E.g., Date values are not in the ISO 8601 format.
    - ISO 8601: 2014-08-16
    - Non-standard format: 8/16/2014
- Dates or numbers elements in the XML are empty.
  - Even if the property on the object that the element is being serialised into is a nullable type, including an empty element will still produce an error unless the `xsi:nil="true"` attribute is applied as described in the 'Date of Birth' example at the end of the Deserialising the Request section.

Additionally, if the property name in the object does not exactly match the XML element name (this is case-sensitive), the property will not be populated during the deserialisation process.

Parsing Posted XML Request

Parsing the posted XML request manually may be a better option than deserialising it if:

- The XML is complex, e.g., many levels of nesting.
- You want more control than the deserialisation process can provide, e.g.:
  - The XML has different versions, e.g., a 'version' attribute on the root node affects how the XML should be parsed.
  - The XML contains dates not formatted according to the ISO 8601 standard.
    - This may be the case if the supplied XML is in a legacy format of comes from a source that you do not have control over.

The finPOWER Connect business layer has helper methods to assist in the parsing of XML.

The following example shows how to manually parse the XML used in Deserialising Collections sample:

```
Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Batch As finBatch
  Dim BatchTransaction As finBatchTransaction
  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean
  Dim XmlDocument As XmlDocument
  Dim Node As XmlNode
  Dim Nodes As XmlNodeList

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Parse XML
  With finBL.Runtime.XmlUtilities
    ' Load XML Document
    If .LoadXmlDocumentFromString(request.RequestText, XmlDocument) Then
      ' Initialise
      Batch = finBL.CreateBatch()

      ' Get Root Node
      Node = XmlDocument.SelectSingleNode("TransactionBatch")
      If Node Is Nothing Then
        Ok = False
        finBL.Error.ErrorBegin("TransactionBatch node not found.")
      End If

      ' Setup Batch
      If Ok Then
        Batch.BatchId = .GetSubNodeString(Node, "BatchId")
        Batch.TransactionTypeId = .GetSubNodeString(Node, "TransactionType")
        Batch.SourceSet(isefinTransactionSource.ExternalA)
      End If

      ' Transactions
      If Ok Then
        Nodes = Node.SelectNodes("Transactions/Transaction")

        For Each Node In Nodes
          ' Create
          BatchTransaction = Batch.Transactions.CreateBatchTransaction()

          ' Update (ignore errors for simplicity of sample)
          BatchTransaction.AccountIdSet(.GetSubNodeString(Node, "AccountId"))
          BatchTransaction.ElementIdSet(.GetSubNodeString(Node, "ElementId"))
          BatchTransaction.Date = .GetSubNodeDate(Node, "Date")
          BatchTransaction.Value = .GetSubNodeDecimal(Node, "Value")
          BatchTransaction.Reference = .GetSubNodeString(Node, "Reference")

          ' Add
          Batch.Transactions.Add(BatchTransaction)
        Next
      End If
```

```
      ' Save Batch
      If Ok Then
        Ok = Batch.Save()
      End If

      ' Commit Batch
      If Ok Then
        Ok = Batch.ExecuteCommit(False)
      End If
    Else
      Ok = False
    End If
  End With

  ' Return Response
  If Ok Then
    Return request.CreateResponse(HttpStatusCode.OK)
  Else
    Return request.CreateErrorResponse(ErrorStatusCode, String.Format("Failed to execute custom Web
Service Script '{0}'.", ScriptInfo.ScriptId), ErrorCode, finBL.Error.Message(True, True))
  End If

End Function
```

Points of note in the above example are:

- Use of the `finBL.Runtime.XmlUtilities` helper methods to load and parse XML.

Parsing Posted JSON Request

Parsing the posted JSON request manually may be a better option than deserialising it if:

- The JSON is complex, e.g., many levels of nesting.
- You want more control than the deserialisation process can provide, e.g.:
  - The JSON contains dates not formatted according to the ISO 8601 standard.
    - This may be the case if the supplied JSON is in a legacy format of comes from a source that you do not have control over.

The finPOWER Connect business layer has helper methods to assist in the parsing of JSON.

NOTE: This functionality was added as at version 2.03.00 of finPOWER Connect.

The following example (a 'General' type Script, not a 'Custom Web Service' Script) shows how to manually parse a JSON string:

```
Public Function Main(parameters As ISKeyValueList) As Boolean

  Dim i As Integer
  Dim JsonText As String
  Dim JsonToken As ISJsonToken
  Dim JsonTokenRoot As ISJsonToken
  Dim JsonTokens() As ISJsonToken

  ' Assume Success
  Main = True

  JsonText = "{""AccountId"":""L1000"",""Payments"":[{""Date"":""2015-05-
01T00:00:00"",""Reference"":""AP"",""Value"":120.0},{""Date"":""2015-06-
01T00:00:00"",""Reference"":""DD"",""Value"":125.0}]}"

  If finBL.Runtime.JsonUtilities.LoadJsonTokenFromString(JsonText, JsonTokenRoot) Then
    finBL.DebugPrint(String.Format("AccountId : {0}",
                     JsonTokenRoot.GetPropertyString("AccountId")))

    JsonTokens = JsonTokenRoot.GetPropertyArray("Payments")
    For i = 0 To JsonTokens.Length - 1
      JsonToken = JsonTokens(i)

      finBL.DebugPrint("")
      finBL.DebugPrint(String.Format("Item {0}", i))
      finBL.DebugPrint(String.Format("Date : {0}",
                       finBL.FormatDateLong(JsonToken.GetPropertyDate("Date"))))
      finBL.DebugPrint(String.Format("Reference : {0}", JsonToken.GetPropertyString("Reference")))
      finBL.DebugPrint(String.Format("Value : {0}", JsonToken.GetPropertyDecimal("Value")))
    Next
  Else
    Main = False
  End If

End Function
```

Points of note in the above example are:

- Use of the `finBL.Runtime.JsonUtilities` helper methods to load and parse JSON.

Manually Creating JSON Text

Manually creating JSON may be necessary where a specific JSON format is required, e.g., a format that cannot easily be replicated via automatic serialisation of the Response.

For example, consider the following JSON which would be difficult (or maybe even impossible) to replicate via serialisation:

```
{
  "First_Item": {
    "product name": "Product A",
    "0": {
      "type_of_loan": "Loan Type A",
      "options": [
        {
          "Principal_and_Interest": {
            "account type id": "VL",
            "name": "Variable Loan"
          }
        },
        {
          "Interest_Only": {
            "account type id": "IO",
            "name": "Interest Only"
          }
        }
      ],
      "terms": "1 year,2 years,3 years,4 years,5 years"
    }
  },
  "Second_Item": {
    "product_name": "Product B"
  }
}
```

This can be manually created using the `JsonBuilder` business layer object as per the following example:

```
Dim JsonBuilder As ISJsonBuilder
Dim JsonText As String

JsonBuilder = finBL.Runtime.CreateJsonBuilder(True)
With JsonBuilder
  .ObjectBegin()

  .ObjectBegin("First_Item")
  .WritePropertyString("product_name", "Product A")

  .ObjectBegin("0")
  .WritePropertyString("type_of_loan", "Loan Type A")
  .ArrayBegin("options")

  .ObjectBegin()
  .ObjectBegin("Principal_and_Interest")
  .WritePropertyString("account_type_id", "VL")
  .WritePropertyString("name", "Variable Loan")
  .ObjectEnd("Principal_and_Interest")
  .ObjectEnd()

  .ObjectBegin()
  .ObjectBegin("Interest_Only")
  .WritePropertyString("account_type_id", "IO")
  .WritePropertyString("name", "Interest Only")
  .ObjectEnd("Interest_Only")
  .ObjectEnd()

  .ArrayEnd("options")

  .WritePropertyString("terms", "1 year,2 years,3 years,4 years,5 years")

  .ObjectEnd("0")

  .ObjectEnd("First_Item")


  .ObjectBegin("Second_Item)
  .WritePropertyString("product_name", "Product B")
  .ObjectEnd("Second_Item")
```

```
   .ObjectEnd()
End With

JsonText = JsonBuilder.ToJsonString()
```

Documents, Emails and SMS Messages

This section details creating documents, Emails and SMS messages from Custom Web Services.

This includes both documents defined in the finPOWER Connect Documents library and also ad-hoc documents, Emails and SMS messages generated on-the-fly.

Important

- It is generally advisable to record any document sent in a Log, e.g., an Account Log.
  - The normal finPOWER Connect publishing process requires that, under normal circumstances, documents defined in the Documents library are published from Logs.
  - However, when sending Ad-Hoc documents, it is the responsibility of the Custom Web Service to record details of the document sent.
    - Generally this should be against a Log such as an Account Log.
- Never attempt to contact an external Web Service or server from within a database transaction.
  - Sending of SMS messages via a one of the built-in finPOWER Connect services will enforce this.
  - However, if sending an Email via SMTP (or some other external means), ensure that this does not occur within a database transaction.

Limitations

The following limitations should be kept in mind when attempting to publish documents from Custom Web Services:

- The Document Manager may be unavailable:
  - The Document Manager is used to store documents, e.g., all documents relating to a particular Account.
  - Unless the Web Service hosting the finPOWER Connect Web Services can access the Document Manager, e.g., if has been configured by a network administrator to access the a Windows UNC path at which the physical files exist, Custom Web Services will be unable to read or write files.
    - This does not prevent the document content from being embedded in the Log as either HTML or binary file data (e.g., a PDF document).
      - This will obviously impact on the database size.
- Sending of SMS messages or Emails may require the network administrator to make changes to the firewall to allow these requests to occur from the Web Server.
- finPOWER Connect uses an external component to convert HTML to a PDF document therefore the formatting is outside of the control of Intersoft Systems.

Documents

The finPOWER Connect database allows a library of Documents to be defined under Admin, Documents.

NOTE: The next section, Ad-Hoc Documents, relates to creating documents not using the built-in finPOWER Connect Documents library.

Documents can be of the following File Types:

- Word VBA
- Excel VBA
- Email
- SMS Message
- Script
- Log
- HTML

Creation of a Document (e.g., an Account Document) requires the following to be performed:

- Create a Log
  - E.g., a `finAccountLog` object
- Configure the Log:
  - Set properties, e.g.:
    - The Account to link to
    - Other mandatory information such as a subject
  - Link the log to the required Document
  - Save the Log
- Publish the Log

Of all the steps listed above, the final one, **Publish the Log** is the only part that needs to be handled in a special manor from Custom Web Services.

Up until finPOWER Connect version 2.02.06, all Document Publishing occurred through the finPOWER Connect User Interface. This meant that although you could create Document Logs using the business layer, publishing of these Logs (e.g., sending the Email, running the Script or generating the Word document) relied on finPOWER Connect being run on the desktop.

In version 2.02.06, the concept of 'Unattended Publishing' was introduced (although only for Script-type Documents in that particular release, the other Document types were supported from the version after this). This allows Documents to be published without relying on the finPOWER Connect User Interface, e.g., from within a Custom Web Service Script.

The following types of Document can be published using 'Unattended Publishing' and are therefore relevant to this section:

- Email
- SMS Message
- Script
- HTML

Logs are published individually using the Publish method of the Log object (e.g., `finAccountLog.Publish`). To publish a Log from as custom Web Service, you must specify that an `unattendedPublish` parameter of `True`, e.g.:

```
Dim AccountLog As finAccountLog
Dim Ok As Boolean

AccountLog = finBL.CreateAccountLog()
With AccountLog
  ' Load
  Ok = .Load(3375)

  ' Publish
  If Ok Then Ok = .Publish(isefinLogPublishType.Publish, True)
End With
```

WARING: The following will never be able to be published from Custom Web Services due to their reliance on Microsoft Office:

Word VBA
Excel VBA

Script-Type Documents

This section shows how to use a Script-type Document to create a PDF document.

The sample Custom Web Service Script does the following:

- Creates an Account Document Log based on the Script-type Document 'LDP.PDF'.
- Publishes this Log.
  - This embeds the binary File Data within the Log.
- Streams the Log's embedded file data.

This sample relies on the following:

- The sample Loan Declaration of Purpose PDF Document existing in the database with a code of 'LDP.PDF'.
  - **NOTE:** This can be imported from the **Document_Loan_DeclarationOfPurpose_PDF.xml** file found in the finPOWER Connect **/Template/Script Samples** folder.
- The Document relies on having access to the **Loan_DeclarationOfPurpose.docx** Word document.
  - **NOTE:** The path of this document is defined as a constant at the top of the Document Script (as opposed to the Custom Web Service Script). This file can be found in the finPOWER Connect **/Templates/Script Samples** folder.

```
Option Explicit On
Option Strict On

' ###################################################################
' Create a Loan Declaration of Purpose PDF from LDP.PDF Document
'
' Version: 1.00 (13/04/2015)
'
' Usage: Custom Web Service
' ###################################################################

' Constants
Const DocumentId As String = "LDP.PDF"

Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim AccountId As String
  Dim AccountLog As finAccountLog
  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Get Parameters
  AccountId = request.Parameters.GetString("AccountId")

  ' Validate
  If Len(AccountId) = 0 Then
    Ok = False
    ErrorCode = "AccountId.Missing"
    finBL.Error.ErrorBegin("Account Id not specified.")
  End If

  ' Create Account Document Log
  If Ok Then
    AccountLog = finBL.CreateAccountLog()
    With AccountLog
      .AccountId = AccountId
      .Subject = "Loan Declaration of Purpose"
      .DocumentId = DocumentId

      ' Save
```

```
      If Ok Then Ok = .Save()

      ' Publish (creates Embedded Document in Log)
      If Ok Then Ok = .Publish(isefinLogPublishType.Publish, True)

      ' Refresh Log (so Embedded Document is available)
      If Ok Then Ok = .Refresh()
    End With
  End If

  ' Return Response
  If Ok Then
    Return request.CreatePdfResponse(HttpStatusCode.OK, AccountLog.EmbeddedFileData)
  Else
    Return request.CreateErrorResponse(ErrorStatusCode,
                                       String.Format("Failed to execute custom Web Service Script
'{0}'.", ScriptInfo.ScriptId),
                                       ErrorCode,
                                       finBL.Error.Message(True))
  End If

End Function
```

The following points should be noted from the above example:

- The sample 'LDP.PDF' Document creates and embeds a PDF file in the Account Log.
  - If the Document instead embedded a Word Document (docx) in the Log, the above example would need to be changed to either:
    - Use the `request.CreateDocxResponse` method to return the binary file data.
    - Use the `ISWordDocument` object to load the Account Log's embedded file data and get it as an array of PDF bytes.
- The Account Log is published using the `Publish` method.
  - A `True` parameter is passed in so that the publish can be performed from Web Services (by default this is handled by the finPOWER Connect User Interface which is not available from Custom Web Services).
  - The Account Log must be refreshed after the `Publish` method has been called otherwise the embedded file data will not be available.

NOTE: Although the sample Document specifies the Word document path as a constant, you might wish to modify this code to pick it up from a fixed location if it is running under Web Services. This would mean modifying the 'Get Constants' section of the **LDP.PDF** Document e.g.:

```
If finBL.IsRunningFromWeb Then
  TemplateFilename = "c:\webserver\Loan_DeclarationOfPurpose.docx"
End If
```

Ad-Hoc Documents

This section shows examples of creating Ad-Hoc documents, i.e., document such as PDFs, Word, Emails and SMS messages that do not relate to the finPOWER Connect Documents library.

finPOWER Connect contains the following functionality that is useful for creating both Word and PDF documents:

- The `ISWordDocument` object that can be used to create a Word document (without Microsoft Word) and save as either Docx or PDF format.
- The ability to convert HTML into a PDF document.

Word Document stored in an Account Log and returned as PDF

This section shows how to create a Word document (docx) using the `ISWordDocument` business layer object.

The sample Custom Web Service Script does the following:

- Creates a Word document using the `ISWordDocument` functionality.
- Creates an Account Log and attaches this document to the Account Log.
  - **NOTE:** At this point, it is possible to attach the document as a PDF file. However, storing it within the Log as a Word document has some advantages, e.g., the ability to easily print the Document from the Account Log form.
- Streams the generated Word Document as a PDF file.

This sample relies on the following:

- The Web Service having access to the **Loan_DeclarationOfPurpose.docx** Word document.
  - **NOTE:** The path of this document is defined as a constant at the top of the Script. This file can be found in the finPOWER Connect **/Templates/Script Samples** folder.

```
Option Explicit On
Option Strict On

' ###################################################################
' Create a Loan Declaration of Purpose PDF with Account Log
'
' Version: 1.00 (13/04/2015)
'
' Usage: Custom Web Service
' ###################################################################

' Constants
Const WordTemplateFileName As String = "c:\test\Loan DeclarationOfPurpose.docx"

Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Account As finAccount
  Dim AccountId As String
  Dim AccountLog As finAccountLog
  Dim Bookmark As ISWordDocumentBookmark
  Dim Bookmarks As ISWordDocumentBookmarks
  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim FileDataDocX As Byte()
  Dim FileDataPdf As Byte()
  Dim Ok As Boolean
  Dim WordDocument As ISWordDocument

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Get Parameters
  AccountId = request.Parameters.GetString("AccountId")

  ' Validate
  If Len(AccountId) = 0 Then
    Ok = False
    ErrorCode = "AccountId.Missing"
    finBL.Error.ErrorBegin("Account Id not specified.")
  End If

  ' Load Account
  If Ok Then
    Account = finBL.CreateAccount()
    Ok = Account.Load(AccountId)
  End If

  ' Create Word Document
  If Ok Then
```

```
    WordDocument = finBL.CreateWordDocument()
    With WordDocument
      ' Load Word Document
      Ok = .Open(WordTemplateFileName)

      ' Replace Bookmarks
      If Ok Then
        ' Get Bookmarks
        Bookmarks = WordDocument.GetBookmarks()

        ' Update (just a small subset for example)
        For Each Bookmark In Bookmarks
          Select Case UCase(Bookmark.Table)
            Case "ACCOUNT"
              ' Account Table
              Select Case UCase(Bookmark.TableField)
                Case "ACCOUNTID"
                  Bookmark.ContentNew = Account.AccountId
                Case "NAME"
                  Bookmark.ContentNew = Account.Name
              End Select
          End Select
        Next

        ' Update Bookmarks
        WordDocument.UpdateBookmarks(Bookmarks)
      End If

      ' Get binary File Data (as Docx for Log and PDF to send back from this Web Service)
      If Ok Then Ok = .SaveDocXToByteArray(FileDataDocX)
        If Ok Then Ok = .SavePdfToByteArray(FileDataPdf)
    End With
  End If

  ' Create Account Log
  If Ok Then
    AccountLog = finBL.CreateAccountLog()
    With AccountLog
      .AccountPk = Account.Pk
      .Subject = "Loan Declaration of Purpose"

      ' Embed Word Document
      Ok = .EmbeddedFileSetFromByteArray(isefinLogEmbeddedFileType.Docx, FileDataDocx,
                          finBL.Runtime.FileUtilities.GetFileName(WordTemplateFileName))

      ' Save
      If Ok Then Ok = .Save()
    End With
  End If

  ' Return Response
  If Ok Then
    ' Even though a Word File has been produced, we can get it as PDF to stream back
    Return request.CreatePdfResponse(HttpStatusCode.OK, FileDataPdf)
  Else
    Return request.CreateErrorResponse(ErrorStatusCode,
                                       String.Format("Failed to execute custom Web Service Script
'{0}'.", ScriptInfo.ScriptId),
                                       ErrorCode,
                                       finBL.Error.Message(True))
  End If

End Function
```

NOTE: The bookmark replacement code in the above example is taken from the sample Document, **Document_Loan_DeclarationOfPurpose_PDF.xml**, found in the finPOWER Connect **/Templates/Script Samples** folder.

Word Document returned as PDF

This example is the same as the previous one but bypasses creating an Account Log and simply streams the PDF file back to the caller.

```
Option Explicit On
Option Strict On

' ###############################################################
' Create a Loan Declaration of Purpose PDF
'
' Version: 1.00 (13/04/2015)
'
' Usage: Custom Web Service
' ###############################################################

' Constants
Const WordTemplateFileName As String = "c:\test\Loan_DeclarationOfPurpose.docx"

Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim Account As finAccount
  Dim AccountId As String
  Dim Bookmark As ISWordDocumentBookmark
  Dim Bookmarks As ISWordDocumentBookmarks
  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim FileDataPdf As Byte()
  Dim Ok As Boolean
  Dim WordDocument As ISWordDocument

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Get Parameters
  AccountId = request.Parameters.GetString("AccountId")

  ' Validate
  If Len(AccountId) = 0 Then
    Ok = False
    ErrorCode = "AccountId.Missing"
    finBL.Error.ErrorBegin("Account Id not specified.")
  End If

  ' Load Account
  If Ok Then
    Account = finBL.CreateAccount()
    Ok = Account.Load(AccountId)
  End If

  ' Create Word Document
  If Ok Then
    WordDocument = finBL.CreateWordDocument()
    With WordDocument
      ' Load Word Document
      Ok = .Open(WordTemplateFileName)

      ' Replace Bookmarks
      If Ok Then
        ' Get Bookmarks
        Bookmarks = WordDocument.GetBookmarks()

        ' Update (just a small subset for example)
        For Each Bookmark In Bookmarks
          Select Case UCase(Bookmark.Table)
            Case "ACCOUNT"
              ' Account Table
              Select Case UCase(Bookmark.TableField)
                Case "ACCOUNTID"
                  Bookmark.ContentNew = Account.AccountId
                Case "NAME"
                  Bookmark.ContentNew = Account.Name
              End Select
          End Select
        Next
```

```
        ' Update Bookmarks
        WordDocument.UpdateBookmarks(Bookmarks)
      End If

      ' Get binary File Data (as PDF to send back from this Web Service)
      If Ok Then Ok = .SavePdfToByteArray(FileDataPdf)
    End With
  End If

  ' Return Response
  If Ok Then
    ' Even though a Word File has been produced, we can get it as PDF to stream back
    Return request.CreatePdfResponse(HttpStatusCode.OK, FileDataPdf)
  Else
    Return request.CreateErrorResponse(ErrorStatusCode,
                                       String.Format("Failed to execute custom Web Service Script
'{0}'.", ScriptInfo.ScriptId),
                                       ErrorCode,
                                       finBL.Error.Message(True))
  End If

End Function
```

PDF Document from HTML

Custom Web Services can return a Response representing a PDF document using either the **request.CreatePdfResponseFromHtml** or **request.CreatePdfResponse** methods.

The following example returns a simple PDF document:

```
Option Explicit On
Option Strict On

Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Return Response
  If Ok Then
    Return request.CreatePdfResponseFromHtml(HttpStatusCode.OK, "<h1>My PDF document</h1>.")
  Else
    Return request.CreateErrorResponse(ErrorStatusCode, String.Format("Failed to execute custom Web
Service Script '{0}'.", ScriptInfo.ScriptId), ErrorCode, finBL.Error.Message())
  End If

End Function
```

This returns the PDF file content as binary data. The following is a sample response with the majority of the HTTP body stripped out since it is meaningless:

```
HTTP/1.1 200 OK
Content-Length: 2401
Content-Type: application/pdf
Date: Wed, 04 Jun 2014 00:29:48 GMT
Expires: -1
Server: Microsoft-IIS/8.0
X-AspNet-Version: 4.0.30319
X-Powered-By: ASP.NET

%PDF-1.5
%����
%Created by EVO PDF Tools v3.5
```

Note that the Content Type header specifies **application/pdf**.

A consumer of this Web Service could then retrieve the response data as an array of bytes and either write it to a file or stream it to the user's Web browser.

PDF Document from HTML, Base-64 Encoded in Complex Response

In the previous section, a custom Web Service was used to return a PDF document. The response from the custom Web Service was a complete PDF document and contained no other information.

In certain situations, it may be desirable to return the PDF document as part of a more complex response, e.g., an XML response containing a `<PdfDocument>` node.

The following example returns a response containing Base 64 encoded PDF data. This can then be decoded by the Web Service consumer and written to a PDF file:

```
Option Explicit On
Option Strict On

Public Function Main(request As finwsHttpRequest) As finwsHttpResponse

  Dim ErrorCode As String
  Dim ErrorStatusCode As HttpStatusCode
  Dim Ok As Boolean
  Dim PdfDetails As PdfDetails
  Dim PdfDocumentBase64 As String

  ' Assume Success
  Ok = True

  ' Initialise
  ErrorStatusCode = HttpStatusCode.BadRequest

  ' Create PDF Document from HTML
  Ok = finBL.PdfUtilities.CreatePdfBase64StringFromHtml("<h1>My PDF Document</h1>",
PdfDocumentBase64)

  ' Create PDF Details
  If Ok Then
    PdfDetails = New PdfDetails()
    With PdfDetails
      .Title = "PDF Title"
      .PdfDocument = PdfDocumentBase64
    End With
  End If

  ' Return Response
  If Ok Then
    ' Return PDF Details
    Return request.CreateResponse(HttpStatusCode.OK, PdfDetails)
  Else
    ' Error
    Return request.CreateErrorResponse(ErrorStatusCode, String.Format("Failed to execute custom Web
Service Script '{0}'.", ScriptInfo.ScriptId), ErrorCode, finBL.Error.Message())
  End If

End Function


Public Class PdfDetails

  Public Title As String
  Public PdfDocument As String

End Class
```

Since this Script returns an object (PdfDetails), it will automatically be serialised as either XML or JSON as per the following (stripped down) responses:

```
<PdfDetails>
<Title>PDF Title</Title>
<PdfDocument>JVBERi0xLjUNCiWxsrO0DQolQ3JlYXRlZCBieSBFVk8gUERGIFRvb2xzIHYzLjUNCjEgMCBvYmoNCjw8DQovUGF
nZXMgMiAwIFINCi9QYWdlTGF5b3V0IC9PbmVDb2x1bW4NCi9QYWdlTW9kZSAvVXNlTm9uZQ0KL1ZpZXdlclByZWZlcmVuY2VzIDM
gMCBSDQovVHlwZSAvQ2F0YWxvZw0KPj4NCg0KZW5kb2JqDQo2IDAgb2JqDQo8PA0KL0ZpbHRlciAvRmxhdGVEZWNvZGUNCi9MZW5
ndGggMTMzDQo+Pg0Kc3RyZWFt</PdfDocument>
</PdfDetails>
```

```
{"Title":"PDF Title",
"PdfDocument":"JVBERi0xLjUNCiWxsrO0DQolQ3JlYXRlZCBieSBFVk8gUERGIFRvb2xzIHYzLjUNCjEgMCBvYmoNCjw8DQovU
GFnZXMgMiAwIFINCi9QYWdlTGF5b3V0IC9PbmVDb2x1bW4NCi9QYWdlTW9kZSAvVXNlTm9uZQ0KL1ZpZXdlclByZWZlcmVuY2VzI
```

DMgMCBSDQovVHlwZSAvQ2F0YWxvZw0KPj4NCg0KZW5kb2JqDQo2IDAgb2JqDQo8PA0KL0ZpbHRlciAvRmxhdGVEZWNvZGUNCi9MZW5ndGggMTMyDQo+Pg0Kc3RyZWFt"}

Appendix A – Guidelines

Formatting HTML for Generating a PDF Document

finPOWER Connect contains functionality to convert HTML into a PDF via the **finBL.PdfUtilities** class.

When creating PDF documents from HTML, note the following:

- The PDF file is generated on the Web Server on which the Web Services are running and can therefore only access any fonts available on this Server.
  - You may wish to use safe fonts such as:
    - Arial
    - Times New Roman
    - Verdana
- Any images you wish to include in the PDF, e.g., company logos should be referenced via a URL that can be resolved by the Web Server.
- You do not have the same amount of control over the document that is produced as you would using a Word processing package such as Microsoft Word.
- By default, the HTML is formatted as if displayed in a browser window that is 1024 pixels wide.
  - This can be changed via the ViewportWidth meta tag detailed below.

The following meta tags can be included at the top of the HTML to tweak that PDF file that is produced:

```
<meta name='pdf-PageSize' content='A4'/>
<meta name='pdf-PageOrientation' content='Portrait'/>
<meta name='pdf-LeftMargin' content='1.5cm'/>
<meta name='pdf-RightMargin' content='1cm'/>
<meta name='pdf-TopMargin' content='1.5cm'/>
<meta name='pdf-BottomMargin' content='2.54cm'/>
<meta name='pdf-ViewportWidth' content='1200'/>
```

- PageSize
  - One of the following: **A4**, **A5**, **Legal**, **Letter**
- PageOrientation
  - Either **Portrait** or **Landscape**
- LeftMargin, RightMargin, TopMargin, BottomMargin
  - Can be specified in the following units: **cm**, **inches** (if omitted, points are assumed)
- ViewportWidth
  - This defaults to **1024** but specifying a different value will vary how the PDF is generated, e.g., how much information fits on a line.
    - By changing this value, the HTML text may scale to a different size so for consistency, it may be best to not set the ViewportWidth but to change the font size in the HTML using CSS.