Tui Objective-C API Specification

# Executive Summary

This document describes the classes implemented for the general-purpose library developed by the Tui Innovation Lab with the objective of being useful for most of Tui brands and back-ends in their mobile developments. This document addresses the native iOS version of the library, although it is already available in browser Javascript and Titanium Javascript. No native Android version has been developed so far.

# Introduction

Tui Travel AG is the biggest travel corporation in the world. It has more than 300 brands, each of them operating their own business, and many of them running their own back-end, managing their own formats, standards, etc. Many of these brands are entering the mobile arena now. They are building their first apps and releasing them to the market. Isolating the UI/UX layer of the app from the business logic/transactional part is always an interesting architectural feature to have. And this was the main idea behind the design and development of the client-side library described in this document: to build an abstraction layer that decouples the UI/UX side of the app from the transactions with the back-end. The library wants to be useful for every brand of Tui, regardless of their back-end. In order to achieve this, it focuses on the common features that typically all apps will have to deal with:

* Connection to servers through either GET or POST calls.
* Translation between JSON, XML and native objects. Setting a standard within the group about how to “translate” JSON to XML and vice-versa. Being able to produce a native object from any of the two formats.

# The JSON to XML Standard

Both JSON and XML define rules for encoding documents in a way that are both human-readable and machine-readable. Both aim to be simple, general and usable over the Internet. XML dates back from the 90’s while JSON was created in 2001, but it is widely used only since 2005-2006. Most of the old back-ends use and maintain XML feeds, but the majority of the new ones use JSON. Because it’s simpler, because it’s lighter and because it’s native to Javascript the most widely used language in the web. Tui brands use XML massively in their backends, and therefore it is important to have a way to “translate” XML into JSON and vice versa. We don’t need to strictly follow the XML standard. It’s enough (and more practical) to have something that is fit-for-purpose.

JSON is a quite simple standard. It defines collections of key-value pairs (also called *dictionaries*) where the key is always a string, and the value could be a string, another dictionary or an array (of strings and/or dictionaries). XML, on the other hand, defines a quite complicated range of elements, nodes, attributes, namespaces. But in the end, everything can be reduced to nested dictionaries and arrays, and that is precisely what we are aiming to do with the definition of our translation standard. Here are the basic rules of our translation standard:

* **JSON keys translate to XML elements** except if:
  + The **key starts with the char “@”**. This key represents an attribute, and the value associated to this key has to be a string and represents the value of the attribute.
  + **The key is “#value”**. The value of this key has to be a string and represents the value of the parent node.
  + **The key is “#list”**. The value of this key has to be an array of key-value pairs, and represents a list of XML elements.
* JSON values **that are strings** translate to XML text/attribute values.
* JSON values **that are dictionaries** translate to nested documents within the XML.
* JSON values **that are arrays** translate to lists of elements in the XML document.
* **Namespaces are not considered in JSON**. If an XML document uses namespaces, they will be translated to plain keys in JSON.

The best way to illustrate this standard is through examples.

**Example 1: basic features of the standard**

The following XML is used as a query in the HotelBeds back-end to request a list of hotels in a certain location:

<HotelListRQ echoToken=“DummyEchoToken”

xmlns=“http://www.hotelbeds.com/schemas/2005/06/messages”

xmlns:xsi=“http://www.w3.org/2001/XMLSchema-instance” xsi:schemaLocation=“http://www.hotelbeds.com/schemas/2005/06/messages HotelListRQ.xsd”>

<Language>ENG</Language>

<Credentials>

<User>TEST</User>

<Password>TEST</Password>

</Credentials>

<Destination code=“PMI” type=“SIMPLE”/>

</HotelListRQ>

The translation of this XML to a JSON object following the standard described above would be:

{“HotelListRQ”: {

“@echoToken”: “DummyEchoToken”,

“@xmlns”: “http://www.hotelbeds.com/schemas/2005/06/messages”,

“@xmlns:xsi”: “http://www.w3.org/2001/XMLSchema-instance”,

“@xsi:schemaLocation”: “http://www.hotelbeds.com/schemas/2005/06/messages HotelListRQ.xsd”,

“Language”: “ENG”,

“Credentials”: {

“User”: “TEST”,

“Password”: “TEST”

},

“Destination”: {

“@code”: “PMI”,

“@type”: “SIMPLE”

}

}}

This example illustrates the basic features of the standard: keys and values, attributes (the keys starting by “@”) and nested dictionaries. Note how namespaces (xmlns:xsi) are translated to JSON as plain strings (keys).

**Example 2: the “#value” keyword**

Let’s take the same XML document from Example 1 and add a value to the “Destination” tag, that already has two attributes in it. The XML document would look like this:

<HotelListRQ echoToken=“DummyEchoToken”

xmlns=“http://www.hotelbeds.com/schemas/2005/06/messages”

xmlns:xsi=“http://www.w3.org/2001/XMLSchema-instance” xsi:schemaLocation=“http://www.hotelbeds.com/schemas/2005/06/messages HotelListRQ.xsd”>

<Language>ENG</Language>

<Credentials>

<User>TEST</User>

<Password>TEST</Password>

</Credentials>

<Destination code=“PMI” type=“SIMPLE”>**Palma**</Destination>

</HotelListRQ>

In this case, the equivalent JSON would look like this:

{“HotelListRQ”: {

“@echoToken”: “DummyEchoToken”,

“@xmlns”: “http://www.hotelbeds.com/schemas/2005/06/messages”,

“@xmlns:xsi”: “http://www.w3.org/2001/XMLSchema-instance”,

“@xsi:schemaLocation”: “http://www.hotelbeds.com/schemas/2005/06/messages HotelListRQ.xsd”,

“Language”: “ENG”,

“Credentials”: {

“User”: “TEST”,

“Password”: “TEST”

},

“Destination”: {

“@code”: “PMI”,

“@type”: “SIMPLE”

**“#value”: “Palma”**

}

}}

Note that the “#value” keyword is only needed when the tag has already some elements in it. If “Destination” hadn’t had attributes, and just the value “Palma”, like this:

<Destination code=“PMI” type=“SIMPLE”>Palma</Destination>

the translation to JSON would have been just a key with a string value:

“Destination”:”Palma”

**Example 3: lists with an element tag**

Consider now the following XML Document defining a certain geographic zone:

<Region code=“054”>

<Name>Mallorca</Name>

<ZoneList>

<zone code=“01”>Palma</zone>

<zone code=“02”>Alcudia</zone>

<zone code=“03”>Soller</zone>

</ZoneList>

</Region>

The equivalent JSON object would be:

{“Region”: {

“@code”: “054”,

“Name”: “Mallorca”,

“ZoneList”: [{“zone”: {“@code”: “01”, “#value”: “Palma”}},

{“zone”: {“@code”: “01”, “#value”: “Alcudia”}},

{“zone”: {“@code”: “01”, “#value”: “Soller”}}]

}}

The list in the XML is translated as an array of dictionaries in the JSON, and the tag enclosing the list in the XML is now a key in the JSON.

**Example 4: lists within the body of the document**

The following XML document is a shortened version of the reply served by the HotelBeds system when queried about the classification of tickets in the area of Palma.

<TicketClassificationListRS

xsi:schemaLocation=“http://www.hotelbeds.com/TicketCListRS.xsd”

totalItems=“9”

echoToken=“DummyEchoToken”>

<AuditData>

<ProcessTime>4</ProcessTime>

<ServerName>FORM</ServerName>

<ServerId>FO</ServerId>

</AuditData>

<Classification code=“CULTU”>Culture</Classification>

<Classification code=“FD”>Full Day</Classification>

<Classification code=“FOOD”>Food</Classification>

</TicketClassificationListRS>

The equivalent JSON object would be:

{TicketClassificationListRS: {

“@xsi:schemaLocation”: “http://www.hotelbeds.com/TicketCListRS.xsd”,

“@totalItems”: “9”,

“@echoToken”: “DummyEchoToken”,

“AuditData”: {

“ProcessTime”: “4”,

“ServerName”: “FORM”,

“ServerId”: “FO”

}

“#list”: [{“Classification”: {“@code”:”CULTU”,”#value”: “Culture”}},

{“Classification”: {“@code”:”FD”,”#value”: “Full Day”}},

{“Classification”: {“@code”:”FOOD”,”#value”: “Food”}}]

}}

Now the list is not enclosed by a dedicated tag, but in the same level as other tags and attributes. The list in translated as an array in the JSON preceded by the keyword “#list”.

# Description Maps

When a server returns an XML feed as a response for a query, it usually spits out a lot of data whereas apps normally require a smaller set of data. Description maps are a way to specify which fields are we interested in, and where can we locate these fields within the XML. Description Maps are used to parse XMLs, look for certain field and information, and return a JSON array (called “*output JSON*”) with the required information. Description Maps are themselves JSON arrays and they comply with the following rules:

* Each of the elements of the array is **a dictionary with one key and one value** (there is just one special exception that is explained below).
* The **key** can be:
  + A **“regular” string**, with no dots in it. This indicates the name of a field in the *output JSON*. The value of this key is always another string.
  + A **“dot-separated” string**. The value of this key is always a JSON array that represents a list of items in the XML document. The elements in this JSON array will indicate which fields of the list are interesting and where to find them taking the root of the list as “current document”. The “dot-separated” string is the path of the list of items in the XML document. The key that will identify the list in the output JSON is the last of the “dot-separated” strings adding “List” to the end of it.
* The **value** can be:
  + A **“dot-separated” string** indicating the path of the desired value in the XML document. If any of the strings between the dots starts with “@”, it indicates that it’s an attribute in the XML document. If string is empty, it means that the desired value is in the root of the current document. If the string is identical to the key, then the whole key-value pair can be replaced by just one string (this is the one special exception mentioned at the beginning of these bullets).
  + A **JSON array**, indicating that there is a list of elements in the XML that we want to map to the *output JSON*. This array is just a Description Map of the contents of the list that takes the root of the list as the current document and complies with all the above rules.

This whole explanation about Description Maps might look a little bit cumbersome, but the tool is really easy to use, as shown in the following examples:

**Example 2: TicketClassification**

The following XML is the response returned by the HotelBeds backend when queried about the ticket classification list.

<TicketClassificationListRS xsi:schemaLocation="http://www.hotelbeds.com/schemas/2005/06/messages TicketClassificationListRS.xsd"

totalItems="9"

echoToken="DummyEchoToken">

<AuditData>

<ProcessTime>4</ProcessTime>

<Timestamp>2013-05-15 13:21:03.741</Timestamp>

<RequestHost>10.162.29.83</RequestHost>

<ServerName>FORM</ServerName>

<ServerId>FO</ServerId>

<SchemaRelease>2005/06</SchemaRelease>

<HydraCoreRelease>2.0.201304221213</HydraCoreRelease>

<HydraEnumerationsRelease>1.0.21213</HydraEnumerationsRelease>

<MerlinRelease>N/A</MerlinRelease>

</AuditData>

<Classification code="CULTU">Culture Museums</Classification>

<Classification code="FD">Full Day</Classification>

<Classification code="FOOD">Food Nightlife</Classification>

<Classification code="HD">In the morning</Classification>

<Classification code="MD">Multi Day Services</Classification>

<Classification code="OUTAC">Outdoor Adventure</Classification>

<Classification code="PARTE">Theme Aquatic Parks</Classification>

<Classification code="SHOW">Shows and Events</Classification>

<Classification code="SIGHT">Sightseeing Tours</Classification>

</TicketClassificationListRS>

Imagine that we are only interested in the following fields:

* Total items returned (attribute *totalItems*).
* Code for each of the classification items (attribute *code*).
* Text value for each of the classification items.

The Description Map that will retrieve these values from the XML would look like:

[{"TotalItems":"@totalItems"},

{"Classification":[{"Code":"@code"},

{"Name":""}]

}

]

The output JSON that we will receive if we processed the XML document with the Description Map would look like:

[{“TotalItems”:”9”,

“ClassificationList”:[{“Code”:”CULTU”, “Name”:”Culture Museums”},

{“Code”:”FD”, “Name”:”Full Day”},

{“Code”:”FOOD”, “Name”:”Food Nightlife”},

{“Code”:”HD”, “Name”:”In the morning”},

{“Code”:”MD”, “Name”:”Multi Day Services”},

{“Code”:”OUTAC”, “Name”:”Outdoor Adventure”},

{“Code”:”PARTE”, “Name”:”Theme Aquatic Parks”},

{“Code”:”SHOW”, “Name”:”Shows and Events”},

{“Code”:”SIGHT”, “Name”:”Sightseeing Tours”}

]

}]

The output JSON is an array with a single element, and this single element contains the information specified by the Description Map.

Note that the list names in the *output JSON* are formed by getting the last of the “dot-separated” strings in the Description Map, and adding “List” to the end of it (as in “ClassificationList”).

**Example 2: TicketAvailRS**

The following XML is heavily inspired (added some complexity to it) in the response provided by HotelBeds when you send a TicketAvailRQ query.

<TicketAvailRS xsi:schemaLocation="http://www.hotelbeds.com/schemas/2005/06/messages TicketAvailRS.xsd" totalItems="2" echoToken="DummyEchoToken">

<AuditData>

<ProcessTime>647</ProcessTime>

<Timestamp>2013-05-13 10:49:38.031</Timestamp>

<RequestHost>10.162.29.83</RequestHost>

<ServerName>FORM</ServerName>

<ServerId>FO</ServerId>

<SchemaRelease>2005/06</SchemaRelease>

<HydraCoreRelease>2.0.201304221213</HydraCoreRelease>

<HydraEnumerationsRelease>1.0.201213</HydraEnumerationsRelease>

<MerlinRelease>N/A</MerlinRelease>

</AuditData>

<PaginationData currentPage="1" totalPages="2"/>

<ServiceTicket xsi:type="ServiceTicket" availToken="9ey6mENxtyujqkVKnqvpMA==">

<DateFrom date="DateFrom1"/>

<DateTo date="DateTo1"/>

<Currency code="EUR1">Euro1</Currency>

<TicketInfo xsi-type="ProductTicket">

<Code>000200515</Code>

<Name>Ticket1</Name>

<Descriptions>

<Description type="generalDescription" languageCode="ENG">Description 11</Description>

<Description type="generalDescription" languageCode="SPA">Description 12</Description>

</Descriptions>

<Images>

<Image>

<Type>S</Type>

<Order>0</Order>

<VisualizationOrder>0</VisualizationOrder>

<Url>Image11</Url>

</Image>

<Image>

<Type>S</Type>

<Order>0</Order>

<VisualizationOrder>0</VisualizationOrder>

<Url>Image12</Url>

</Image>

<Image>

<Type>S</Type>

<Order>0</Order>

<VisualizationOrder>0</VisualizationOrder>

<Url>Image13</Url>

</Image>

</Images>

</TicketInfo>

</ServiceTicket>

<ServiceTicket xsi-type="ServiceTicket" availToken="9ey6mENxtyujqkVKnqvpMA==">

<DateFrom date="DateFrom2"/>

<DateTo date="DateTo2"/>

<Currency code="EUR2">Euro2</Currency>

<TicketInfo xsi-type="ProductTicket">

<Code>000200515</Code>

<Name>Ticket2</Name>

<Descriptions>

<Description type="generalDescription" languageCode="ENG">Description 21</Description>

<Description type="generalDescription" languageCode="SPA">Description 22</Description>

</Descriptions>

<Images>

<Image>

<Type>S</Type>

<Order>0</Order>

<VisualizationOrder>0</VisualizationOrder>

<Url>Image21</Url>

</Image>

<Image>

<Type>S</Type>

<Order>0</Order>

<VisualizationOrder>0</VisualizationOrder>

<Url>Image22</Url>

</Image>

<Image>

<Type>S</Type>

<Order>0</Order>

<VisualizationOrder>0</VisualizationOrder>

<Url>Image23</Url>

</Image>

</Images>

</TicketInfo>

</ServiceTicket>

</TicketAvailRS>

This document delivers some audit data and headers about the data returned, and then a list of “*ServiceTickets*” with the relevant info about each ticket. Imagine that the client is only interested in the following:

* Total items returned (attribute *totalItems*).
* Within the list of *ServiceTickets*, we are interested in:
  + *DateFrom*
  + *DateTo*
  + *Currency*
  + *Name*
  + Within the list of images of each ticket (*Images*) we are interested in:
    - *Type*
    - *Url*

The Description Map for these requirements will look like:

[{'TotalItems':'@totalItems'},

{'ServiceTicket':[{'DateFrom':'DateFrom.@date'},

{'DateTo':'DateTo.@date'},

{'Currency':’Currency’},

{'Name': 'TicketInfo.Name'},

{'TicketInfo.Images.Image':

[{'Type': 'Type'},

{'Url': 'Url'}]}

]}

];

The element {'CurrencyCode': 'Currency.@code'} could be replaced by the string ‘CurrencyCode’ and it will work as well. The native object (*output JSON*) that we would receive after processing the XML document with the description map would be (we are using JSON here to describe the object, but the object would be native to the platform used):

[{‘TotalItems’: ‘2’,

‘ServiceTicketList’:[

{‘DateFrom’: ‘DateFrom1’,

‘DateTo’: ‘DateTo1’,

‘Currency’: ‘Euro1’,

‘Name’:’Ticket1’,

‘ImageList’:[

{‘Type’: ‘S’, ‘Url’: ‘Image11’},

{‘Type’: ‘S’, ‘Url’: ‘Image12’},

{‘Type’: ‘S’, ‘Url’: ‘Image13’}

]

},

{‘DateFrom’: ‘DateFrom2’,

‘DateTo’: ‘DateTo2’,

‘Currency’: ‘Euro2’,

‘Name’:’Ticket2’,

‘ImageList’:[

{‘Type’: ‘S’, ‘Url’: ‘Image21’},

{‘Type’: ‘S’, ‘Url’: ‘Image22’},

{‘Type’: ‘S’, ‘Url’: ‘Image23’}

]

}

]

}]

Again, note that the list names in the *output JSON* are formed by getting the last of the “dot-separated” strings in the Description Map, and adding “List” to the end of it (as in “ImageList”).

# Base API Classes

The API defines a set of classes in order to deal with basic stuff such as HTTP calls or XML parsing.

## TuiHttpRequest Class

Basic class for making **synchronous** HTTP calls to servers (either GET or POST) and return the result of the call. It will throw exceptions if something fails during the call.

### Class Methods

**synchronousGetRequestWithURL:**

Sends a synchronous GET request to a server and returns the response from the server. Throws an exception in case something fails.

+ (NSData \*)synchronousGetRequestWithURL:(NSString \*)url

**Parameters**

*url*

The url to call.

**Return Value**

A buffer containing the raw response of the server.

**Exceptions**

*TuiUnauthorizedException*

Thrown when the server returns a 401 HTTP status code.

*TuiInvalidUrlException*

Thrown when the server returns an HTTP status code, different from 200 and 401.

*TuiOfflineException*

Thrown when the device can’t reach the server.

*TuiUrlConnectionException*

Thrown when there is a connection error, other than the device being offline.

**synchronousPOSTRequestWithURL:**

Sends a synchronous POST request to a server and returns the response from the server. Throws an exception in case something fails.

+ (NSData \*)synchronousPostRequestWithURL:(NSString \*)url

**Parameters**

*url*

The url to call.

**Return Value**

A buffer containing the raw response of the server.

**Exceptions**

*TuiUnauthorizedException*

Thrown when the server returns a 401 HTTP status code.

*TuiInvalidUrlException*

Thrown when the server returns an HTTP status code, different from 200 and 401.

*TuiOfflineException*

Thrown when the device can’t reach the server.

*TuiUrlConnectionException*

Thrown when there is a connection error, other than the device being offline.

## TuiXmlReader Class

A singleton class that reads an XML document and a Description Map, and produces an array with the desired keys and values.

### Class Methods

**sharedInstance**

Returns a unique instance of the TuiXmlReader.

+ (TuiXmlReader \*)sharedInstance

**Return Value**

A singleton instance of the TuiXmlReader.

### Instance Methods

**readObjectsFromXmlString:lookingFor:usingDescriptionMap:**

Reads an XML document looking for a specific key within the document and then returns an array containig the key-value pairs specified in the given Description Map.

- (NSArray \*)readObjectsFromXmlString:(NSString \*)xmlString

lookingFor:(NSString \*)key

usingDescriptionMap:(NSArray \*)descriptionMap

**Parameters**

*xmlString*

The raw XML document, in UTF-8 format.

*key*

A key to look for in the XML document. This key will define the “root” tag for the XML parser. It’s normally used to retrieve a list of elements from the XML document, but it could be used for any purpose. If empty, the “root” will be set to the opening tag of the document.

*descriptionMap*

The Description Map specifying what to look for within the XML document.

**Return Value**

An Array containing the data parsed from the XML as specified by the Description Map.

**Exceptions**

*TuiInvalidDescriptionMapException*

Thrown when the provided Description Map does not comply with the defined standard.

## TuiObjectReader Class

A singleton class that builds a native object of a certain class from a given dictionary.

### Class Methods

**sharedInstance**

Returns a unique instance of the TuiObjectReader.

+ (TuiObjectReader \*)sharedInstance

**Return Value**

A singleton instance of the TuiObjectReader.

### Instance Methods

**readObjectOfClass:usingDictionary:**

Reads a native object of a certain class using the key-value pairs provided in a dictionary.

- (NSObject \*)readObjectOfClass:(Class)type

usingDictionary:(NSDictionary \*)JSONObject;

**Parameters**

*type*

The class of the object to be returned.

*jsonObject*

The dictionary providing the key-value pairs to be used when building the object.

**Return Value**

A native object of the class with the keys and values specified by the dictionary provided.