Developing the Mobile Service Solution

The mobile service solution manages the synchronization of the offline data.

In This Chapter

- Requirements
- Constructing an Offline Data Package for the Mobile Client
- User Specific Data
- Providing/Retrieving Entity(=table) Row Data
- Syncing Back the Changes from the Mobile Client
    - Transaction Based Syncing
    - User Properties

Requirements

1. Needs a form with name 'offline_data'
2. The form offline_data needs a method ws_read(version,name)
3. If the data shown in mobile app is user specific the form offline_data needs a method ws_authenticate(useruid,password)
4. The mobile_service and rest_ws plugin to be installed

By default Servoy will load the first 200 pks when a form is used. This is not needed in the service solution and can be prevented by calling databaseManager.setCreateEmptyFormFoundsets() in the solution onopen global method:

```
function onSolutionOpen() {
        // prevent prefilling of form foundsets with default query
        databaseManager.setCreateEmptyFormFoundsets()
}
```

Constructing an Offline Data Package for the Mobile Client

The ws_read(version,name) on the offline_data form has to return an OfflineDataDescription (=JSON) object filled with foundset data the developer wants the mobile client to retrieve.
An OfflineDataDescription instance is created with:

```
var retval = plugins.mobileservice.createOfflineDataDescription('data_')
```

Note: the argument is optional and is used to call rest endpoints for row data at forms starting with this prefix

The next step is determining what related data should be traversed and sent to the client. By default (since Servoy 7.2) any relations that exist in a "Mobile Shared Module" attached to the "Mobile Service Solution" will be included in the traversal.

Alternatively, if you provide a list of relation names to include, this will **override** the inclusion of the relations contained in the "Mobile Shared Module", and only include those relations that you specify.

```
var traverse = [
        'accountmanager_to_companies',
        'companies_to_contacts'

]
```

Lastly we have instruct the OfflineDataDescription to collect the data, starting with root foundset (containing records) and return.

```
retval.addFoundSet(fs_contact, traverse)
return retval
```

Basically addFoundSet in the service solution exposes an unrelated foundset to the mobile client, which can be used in an unrelated way like in a (first) form or databaseManager.getFoundset(...)
For each record in the provided (unrelated) foundset the specified relations are traversed and all data taken.

User Specific Data

In order to provide a mobile client with user specific data the ws_authenticate(useruid,password) method should be added:

```
function ws_authenticate(useruid, password) {
        //TODO find user and check password against pwhash column
        if (password === 'demo') { //static demo check
                return {
                  username: useruid
                }
        }
        return false;
}
```

In the ws_read method we can utilize the authenticate username variable via

```
var authenticate_info = questionParams.ws_authenticate[0]
globals.username = authenticate_info.username
```

Here the authenticate username is put into a global variable which in turn can be used like:

```
//prepare personal data
var fs_contact = globals.contact_data$username //global foundset to account manager contact record
```

Full ws_read method for personalized data.

```
function ws_read(version,name) {
        var questionParams = arguments[arguments.length-1]

        //create return value
        var retval = plugins.mobileservice.createOfflineDataDescription('data_')

        //setting the key for user_select relation
        var authenticate_info = questionParams.ws_authenticate[0]
        globals.username = authenticate_info.username

        //prepare personal data
        var fs_contact = globals.contact_data$username //global foundset to account manager contact record

        var traverse = [
                'accountmanager_to_companies',
                'companies_to_contacts'
        ]

        retval.addFoundSet(fs_contact, traverse)
        return retval
}
```

Providing/Retrieving Entity(=table) Row Data

Row/record data is retrieved in separate calls for each entity, for example for 'orders' row data results in a call to 'orders' form is made on ws_read method.
Note: If a prefix is provided in the offlinedata the call will end up at prefix+entityname, example for prefix 'data_' the call happens on form 'data_orders'

ws_read is with a list of pks it wants as row data for, example code:

```
function ws_read(version, method) {
	var questionParams = arguments[arguments.length-1]

	if (method == 'list') {
		/** @type {String} */
		var ids = questionParams.ids[0]
		if (ids != null && ids != '') {
			var idsa = ids.split(',', -1)
			if (idsa.length > 0) {
				var json = plugins.mobileservice.getRowDescriptions(foundset.getDataSource(),
idsa)
				return json
			}
		}
	}

	throw 404;
}
```

TIP: since ws_read for entities is likely the same, it might be beneficial to create a base form containing this logic and extend from this form

Syncing Back the Changes from the Mobile Client

ws_update is called for changes made by mobile client, example code:

```
function ws_update(data, version, pk) {
	if (foundset.find()) {
		foundset.contact_id = pk
		var count = foundset.search()
		if (count > 0) {
			var rec = foundset.getRecord(1)
			databaseManager.copyMatchingFields(data,rec,true)
			databaseManager.saveData(rec)
		}
	}
}
```

ws_create is called for new records on the mobile client, example code:

```
function ws_create(data, version, pk) {
	var rec = foundset.getRecord(foundset.newRecord())

	databaseManager.copyMatchingFields(data,rec,true)
	databaseManager.saveData(rec)
}
```

Note: the retrieved PK (and derived FK's) is always UUID's if the underlaying datamodel is not UUID based, keep and apply a mapping!

ws_delete is called for deleted record in the mobile client, example code:

```
function ws_delete(version, pk) {
   if (foundset.find()) {
     var table = databaseManager.getTable(controller.getDataSource())
     var pkname = table.getRowIdentifierColumnNames()[0]
     foundset[pkname] = pk
     var count = foundset.search()
     if (count > 0) {
       var rec = foundset.getRecord(1)
       foundset.deleteRecord(rec)
     }
   }
}
```

Transaction Based Syncing

By default the mobile client will do a rest call per changed row, these are separated calls so every call will be on a fresh new client on the serverside.

Because of this a transaction can't be used for all the changes of 1 mobile client and then fail a sync when something goes wrong. If the sync should be in 1 call, so that a transaction can be used over all the calls to the above ws_update/ws_create/ws_delete methods, a ws_update method must be created on the offline_data form.

This ws_update method then gets a full data package of all the changes that a mobile client has, this will then be dispatched over all the methods that are described above.

An example of a ws_update method on the offline_data form:

```
function ws_update(data, version, authenticateResult) {
	try {
		databaseManager.startTransaction()
		plugins.mobileservice.performSync(data, version, authenticateResult)
		databaseManager.commitTransaction()
	} catch (e) {
		databaseManager.rollbackTransaction()
		// log the error and return false to that the mobile client will know the sync did fail.
		application.output(e, LOGGINGLEVEL.ERROR)
		return false
	}
}
```

This code above starts a transaction then calls the performSync method of the mobile service plugin. This plugin will dispatch all the changes to the various ws_update/create/delete methods of the entity forms. If something goes wrong then an exception will be thrown and the transaction will rollback.

User Properties

If using get/setUserProperty() in the Service Solution, the properties are stored in the Mobile Client that connected to the Service Solution.
As such, a user property can be used to store for example an authentication token to be used between multiple calls to the Service Solution from the Mobile Client.

When performing custom Ajax calls to the Service Solution from the Mobile Client, for example using jQuery.ajax, extra steps need to be taken to make sure the user properties are available in the Service Solution and properly persisted. The user properties are to be send with the Ajax request as a special Request Header, identified by the name 'servoy.userproperties'. The current values for the user properties of the Service Solution are stored in the Mobile Client in the localstorage, from which they can be retrieved using the code sessionStorage.getItem('servoy.userproperties'). In the Response of the custom Ajax call, the possible altered value can be stored back into the localstorage using this code: sessionStorage.setItem('servoy.userproperties', value)

Below an example of how to make a custom Ajax request using JQuery.ajax

```
var url = ....
$.ajax({
   url: url,
   beforeSend: function (request) {
        ...
        request.setRequestHeader("servoy.userproperties:", sessionStorage.getItem("servoy.userproperties"));
        ...
   }
}).done(function(data, textStatus, jqXHR) {
   sessionStorage.setItem("servoy.userproperties", jqXHR.getResponseHeader("servoy.userproperties")
});
```