one-X™ R5.2 Web Application SDK

Dashboard Application

16-603493
Release 5.2
November 2009
Issue 1

Notice

While reasonable efforts were made to ensure that the information in this document was complete and accurate at the time of printing, Avaya Inc. can assume no liability for any errors. Changes and corrections to the information in this document might be incorporated in future releases.

Documentation disclaimer

Avaya Inc. is not responsible for any modifications, additions, or deletions to the original published version of this documentation unless such modifications, additions, or deletions were performed by Avaya. Customer and/or End User agree to indemnify and hold harmless Avaya, Avaya's agents, servants and employees against all claims, lawsuits, demands and judgments arising out of, or in connection with, subsequent modifications, additions or deletions to this documentation to the extent made by the Customer or End User.

Link disclaimer

Avaya Inc. is not responsible for the contents or reliability of any linked Web sites referenced elsewhere within this documentation, and Avaya does not necessarily endorse the products, services, or information described or offered within them. We cannot guarantee that these links will work all the time and we have no control over the availability of the linked pages.

Warranty

Avaya Inc. provides a limited warranty on this product. Refer to your sales agreement to establish the terms of the limited warranty. In addition, Avaya’s standard warranty language, as well as information regarding support for this product, while under warranty, is available through the Avaya Support Web site:

http://www.avaya.com/support

License

USE OR INSTALLATION OF THE PRODUCT INDICATES THE END USER'S ACCEPTANCE OF THE TERMS SET FORTH HEREIN AND THE GENERAL LICENSE TERMS AVAILABLE ON THE AVAYA WEB SITE http://support.avaya.com/LicenseInfo/ ("GENERAL LICENSE TERMS"). IF YOU DO NOT WISH TO BE BOUND BY THESE TERMS, YOU MUST RETURN THE PRODUCT(S) TO THE POINT OF PURCHASE WITHIN TEN (10) DAYS OF DELIVERY FOR A REFUND OR CREDIT. Avaya grants End User a license within the scope of the license types described below. The applicable number of licenses and units of capacity for which the license is granted will be one (1), unless a different number of licenses or units of capacity is specified in the Documentation or other materials available to End User. "Designated Processor" means a single stand-alone computing device. "Server" means a Designated Processor that hosts a software application to be accessed by multiple users. "Software" means the computer programs in object code, originally licensed by Avaya and ultimately utilized by End User, whether as stand-alone Products or pre-installed on Hardware. "Hardware" means the standard hardware Products, originally sold by Avaya and ultimately utilized by End User.

License type(s)

Designated System(s) License (DS). End User may install and use each copy of the Software on only one Designated Processor, unless a different number of Designated Processors is indicated in the Documentation or other materials available to End User. Avaya may require the Designated Processor(s) to be identified by type, serial number, feature key, location or other specific designation, or to be provided by End User to Avaya through electronic means established by Avaya specifically for this purpose.

Concurrent User License (CU). End User may install and use the Software on multiple Designated Processors or one or more Servers, so long as only the licensed number of Units are accessing and using the Software at any given time. A “Unit” means the unit on which Avaya, at its sole discretion, bases the pricing of its licenses and can be, without limitation, an agent, port or user, an e-mail or voice mail account in the name of a person or corporate function (e.g., webmaster or helpdesk), or a directory entry in the administrative database utilized by the Product that permits one user to interface with the Software. Units may be linked to a specific, identified Server.

Database License (DL). Customer may install and use each copy of the Software on one Server or on multiple Servers provided that each of the Servers on which the Software is installed communicate with no more than a single instance of the same database.

CPU License (CP). End User may install and use each copy of the Software on a number of Servers up to the number indicated by Avaya provided that the performance capacity of the Server(s) does not exceed the performance capacity specified for the Software. End User may not re-install or operate the Software on Server(s) with a larger performance capacity without Avaya's prior consent and payment of an upgrade fee.

Copyright

Except where expressly stated otherwise, the Product is protected by copyright and other laws respecting proprietary rights. Unauthorized reproduction, transfer, and or use can be a criminal, as well as a civil, offense under the applicable law.

Third-party components

Certain software programs or portions thereof included in the Product may contain software distributed under third party agreements ("Third Party Components"), which may contain terms that expand or limit rights to use certain portions of the Product ("Third Party Terms"). Information identifying Third Party Components and the Third Party Terms that apply to them is available on the Avaya Support Web site:

http://support.avaya.com/ThirdPartyLicense/

Trademarks

Avaya, the Avaya logo, and MultiVantage are either registered trademarks or trademarks of Avaya Inc. in the United States of America and/or other jurisdictions.
Microsoft is a registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners.

Downloading documents

For the most current versions of documentation, see the Avaya Support Web site:

http://www.avaya.com/support

Avaya support

Avaya provides a telephone number for you to use to report problems or to ask questions about your product. The support telephone number is 1-800-242-2121 in the United States. For additional support telephone numbers, see the Avaya Support Web site:

http://www.avaya.com/support

Content:

1 Introduction

The **one-X™ Web Applications SDK** is a package of software development tools created by AVAYA. It is intended for use by AVAYA customers and partners to efficiently develop applications that consume unified communication services hosted by Avaya platforms. This SDK specifically targets development of event-driven web applications which use web browsers as their user interface.

The SDK is based on **one-X™ Portal Server** (1XS) **Release 5.2** as the application runtime framework. 1XS is a set of applications hosted on an IBM WebSphere JEE Application Server. The applications represent interfaces for communication with various AVAYA server products (Communication Manager, Application Enablement Services, Meeting Exchange, Modular Messaging, Intelligent Presence Server) which serve to control the services they offer from a central point of access.

1.1 Purpose

The SDK comprises

- Reusable software libraries and documentation
- Demonstration applications

The **Dashboard** is a set of applications which intend to present most of the features and possibilities of 1XS-based client applications in a comprehensible way. It demonstrates the usage of the 1XS API and may serve as starting point for the design of derived software products.

This document targets:

- Basic structures of 1XS-based client applications.
- Reusable software libraries.
- The **Dashboard** application(s).

For a presentation of test environment and tools (especially the Eclipse IDE delivered with the SDK), please refer to the document "**1X-WebApp-SDK-Workstation**".

1.2 Scope

Required **one-X™ Portal Server Version** is: **5.2**

The current version of the "Dashboard" application does not support the 'UserAssistant' resource!

1.3 Terminology

Term	Meaning
1XP	AVAYA one-X Portal (**same as 1XS**)
1XS	AVAYA one-X Portal Server
ACL	Access Control List (Presence)
ACP	"Avaya Communication Portal", a legacy name for 1XP
ADS	Microsoft Active Directory Service
AES	AVAYA Application Enablement Services
AJAX	Asynchronous JavaScript and XML
A/S	Application Server (JEE)
API	Application Programming Interface
CM	Avaya Communication Manager
DHTML	Dynamic HTML (Hypertext Markup Language)
DTMF	Dual Tone Multifrequency
EAR	Enterprise Archive
EC500	Extension to Cellular (Avaya technology)
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IPS	AVAYA Intelligent Presence Service
JAR	Java Archive
JEE	Java Enterprise Edition
JNDI	Java Naming and Directory Interface
JSDOC	JavaScript Doc (JavaScript documentation derived from source code)
JSP	Java Server Pages
MIME	Multipurpose Internet Mail Extensions
MM	AVAYA Modular Messaging
MX	AVAYA Meeting Exchange
SDK	Software Development Kit
UI	User Interface
URL	Uniform Resource Locator
TLS	Transport Layer Security
VOIP	Voice over IP
WAF	1X Web Application Framework (a set of Avaya libraries)
WAR	Web Archive
WAS	(IBM) WebSphere Application Server
XML	Extensible Markup Language

2 1XS Client Application Architecture

2.1 Client Application Basics

Figure 1 below provides an overview about the structure of 1XS-based web applications.

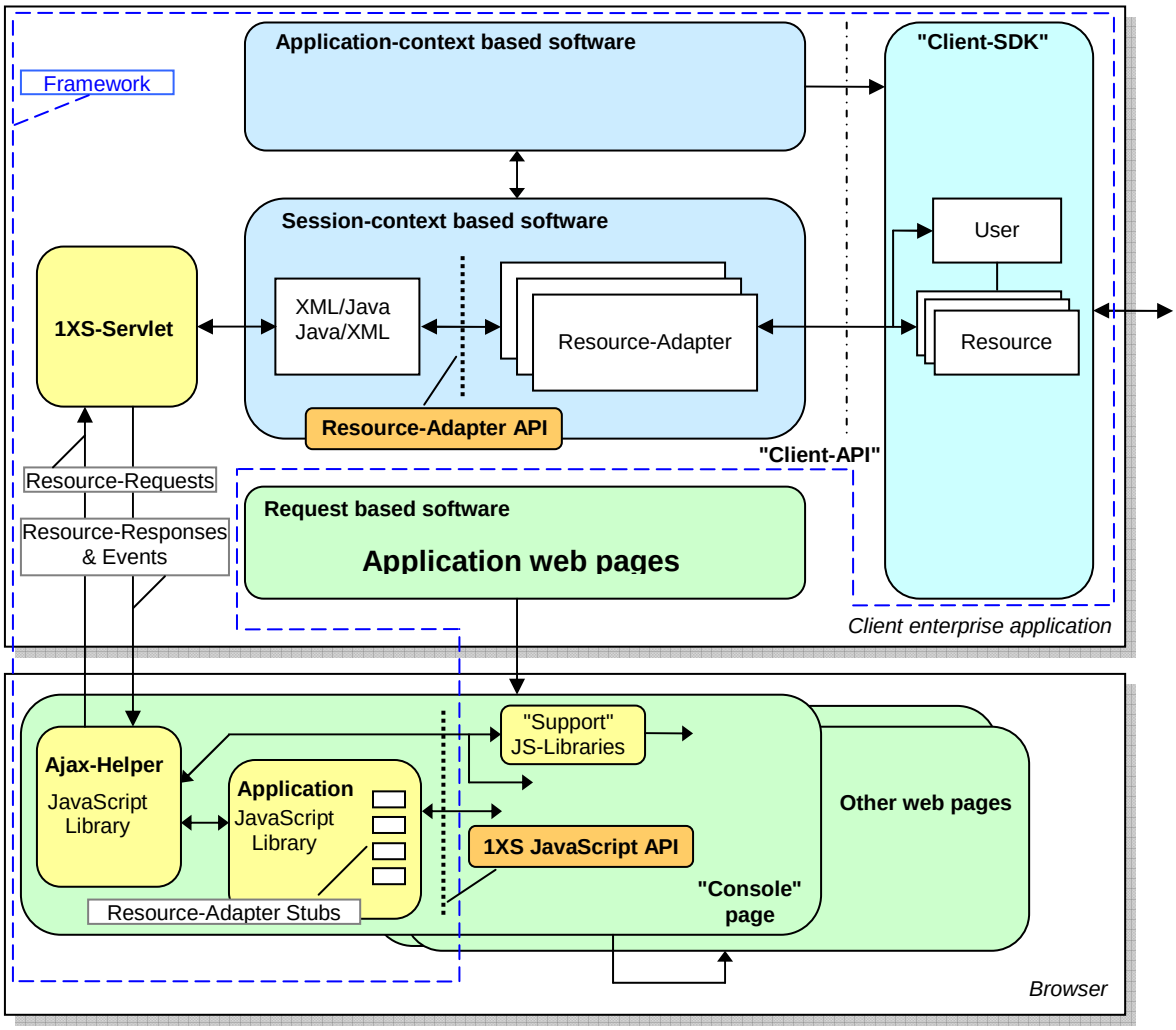

FIGURE 1: Architecture of 1XS based Web Applications

A 1XS Client is a JEE enterprise application to be installed on an IBM WebSphere application server. Basically, 1XS Web Clients consist of a reusable Web Application Framework (**WAF**) and a set of web pages which serve as user interface displayed by a web browser.

Client framework components include:

- **The Client SDK** (**Client API**):
  Library components which interface to Avaya background servers (e.g. AES/CM) via other enterprise applications installed on the application server (e.g. a CM Adapter) or to a database (IBM DB2 in the case of 1XS).
  In 1XS terminology, each instance of an interface component represents a **Resource**. A "Resource" is the "placeholder" for a user-associated resource in one of the Avaya communication server products (e.g. a specific phone defined in CM, a voicemail box in MM, or a conference bridge in MX).

- **Framework components running in application context**:
  Application context software components handle dynamic data valid throughout the lifecycle of the client enterprise application, especially for processing of authentication operations and management of user sessions during their lifetime.
- **Framework components running in session context:**
  Session context software handles instances of data created for each logged-in user, valid until user logout or browser session expiration.
  During a client session, Client-API resource interface objects - called **Resource Adapters** - are created. The client browser communicates with resources via these adapters.
  A Resource Adapter may receive requests from a client browser, forward the requests to one of the Client API resources, receive responses from the Client API resources and transfer response data back to the client browser.
  Additionally, it catches resource events for transfer to the client browser.
  Resource Adapters are data serialization/de-serialization objects, they translate the world of Java method calls (Client API) into/from XML-coded messages.
- **The 1XS-Servlet**:
  A servlet responsible for XML data communication with client browsers.
  Clients use an **AJAX** poll technology (on top of HTTP) to retrieve XML messages from the server.
- **Client web-pages:**
  Web pages displayed inside of the client browser communicate with Resource Adapters via the AJAX-XML path using a JavaScript library called **Ajax-Helper**.
  Due to the nature of AJAX-XML communication, the web pages are loaded once only, and all dynamic page content is rendered using JavaScript (DHTML).
- **Reusable JavaScript Application and Support Libraries:**
  Contain assistant functionality for communication with the server.
- **Client web pages:**
  Communicate with resources using the **1XS JavaScript API** which represents a translation of the server-side **Resource-Adapter API** (Java methods) into JavaScript methods via the AJAX system.
- **Resource Adapter Stub Objects:**
  Provide methods for calling the remote Resource Adapter API methods.

Another client framework component (presented in the next figure) is the **1XS-File-Transfer-Servlet**. This servlet manages transfer (up-/download) of arbitrary file data between a user's browser and a intermediate storage on the server.
The intermediate storage is a folder in the server's file system, it is handled user-session based.

A client application's web page may request a resource to put a file in the intermediate storage, afterwards the data can be downloaded to a web page window, to a plugin used in a web page window or for storage in the user machine's file system.

For upload, a web page may post form data into a file of the intermediate storage, afterwards a resource can be requested to fetch the file content from there.

The 1XS-File-Transfer-Servlet is used for up-/download of text-, audio- or image- parts of messages (AVAYA Modular Messaging).

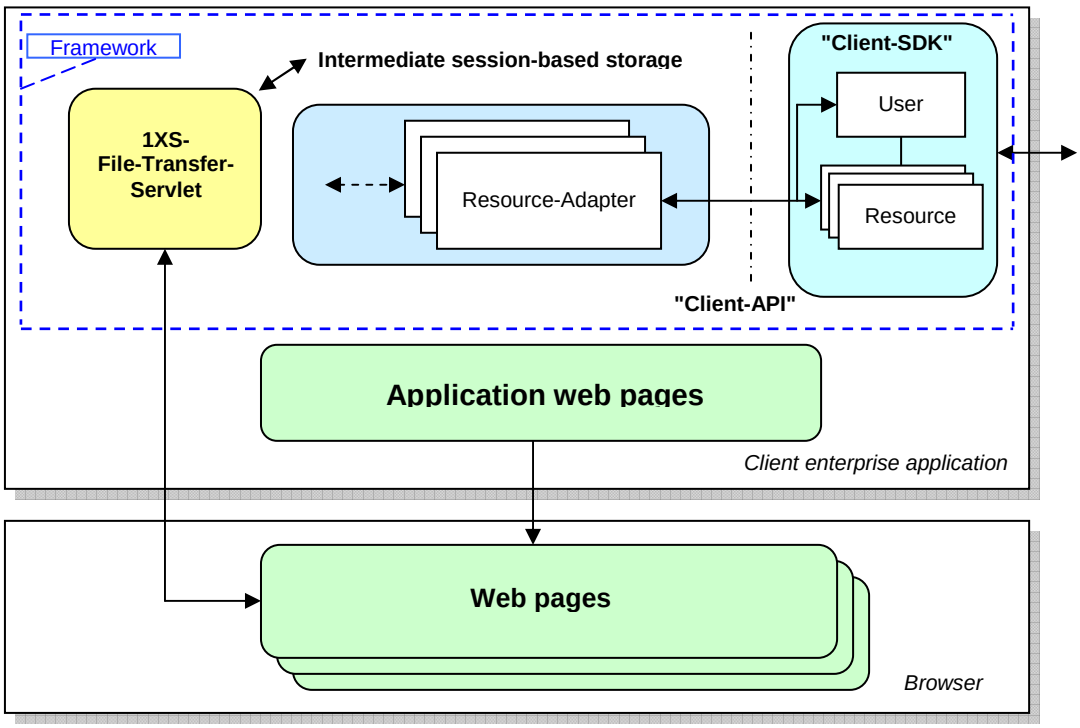

FIGURE 2: File Transfer in 1XS based Web Applications

2.1.1 Requests, Responses and Events

The AJAX system transports 3 types of communication items:

- A **Resource Request** is a method call in the Resource Adapter API. It contains a defined set of parameters and is initiated via the 1XS JavaScript API.
- Think of a **Resource Response** as of the return value of a Request method call.
  For many (but not all) of the server methods, the return value plays an inferior role and data are delivered in the form of additional event messages.
  The Response shows if a Request is understood and can be handled well by the server. In case of errors it may contain exception data.
- Most of the information received by a client web page is transported in the form of **Resource Events**. Event messages are fired upon resource state changes and client requests.

2.1.2 Event-Listeners and Response-Handlers

In order to handle messages sent by 1XS, client web pages have to provide handler and listener methods with well known names.

A handler is a JavaScript object, which can be either a page's window or any other programmer-defined object that implements the known handler/listener method.

For reception of events from resources of interest, web pages have to register a listener object in the AJAX communication layer (AJAX-Helper) provided by basic framework libraries.
Each time that events arrive, they will be propagated by calling the handler method **handleAjaxEvent** included in each registered listener object.

Handler objects for Resource Responses are set as parameter values of Resource-Request 1XS-JavaScript API method calls. The AJAX communication layer calls the handler's method **handleAjaxResourceResponse** upon reception of the response. Usually the page's window object serves as the handler (the window object implements a function "handleAjaxResourceResponse"); thus, the handler object is "**this**".

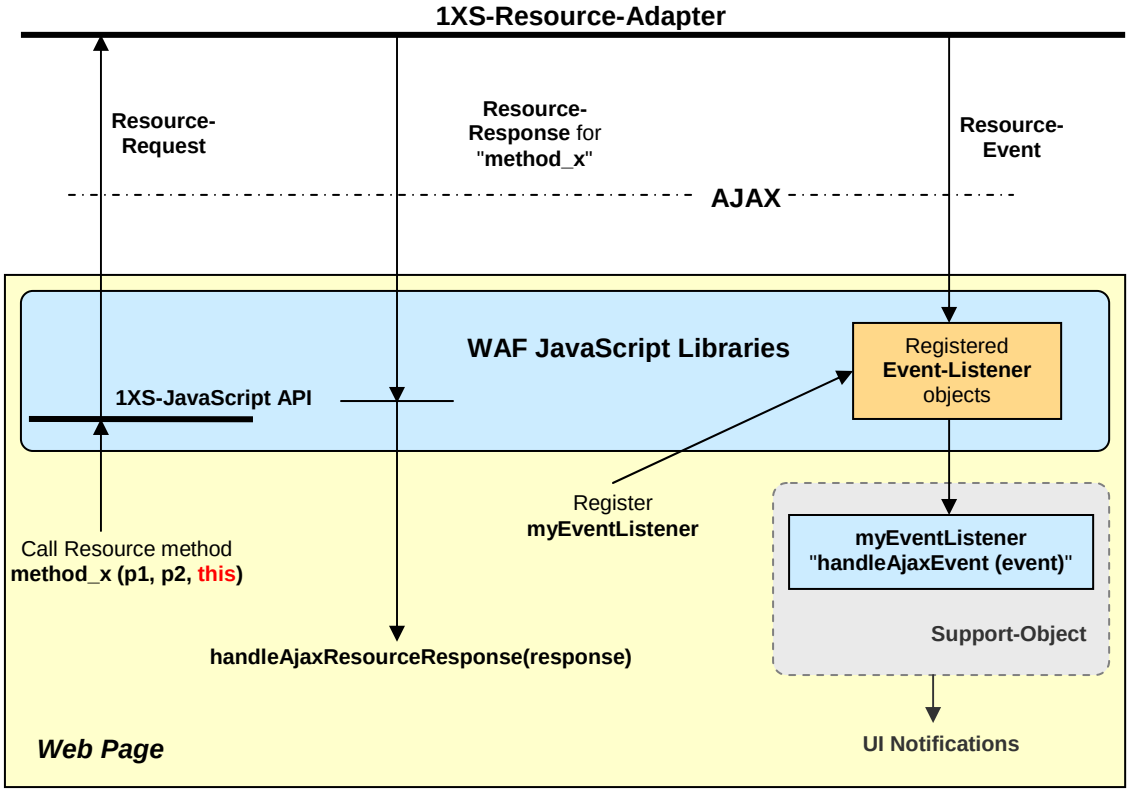

FIGURE 3: Resource Requests, Responses and Events

The typical approach to handling events is to extract payload data from received messages and store this data for later use by decision logic and by UI updates.

Reusable convenience libraries, the **Support** libraries, can optionally perform this task.
In case of support library usage, events are received by a support object which, upon data changes, calls a "UI-Notification" handler for the web page.

An important fact concerning the transport sequence of **Resource Responses** and **Resource-Events** must be noted:

Important:
If a series of Resource Events is fired because of a Resource Request (e.g., contact items after a contact search request), there is no guarantee that the Resource Response arrives after the last event.

2.1.3 Resources and Capabilities

1XS currently provides the following Resources.

Resource	Functions
User	Basic functionality, Presence ACL handling
Extension	Control of the user's phone in an AVAYA CM infrastructure
ContactLog	Logging of contact events (currently: call logging)
ContactList	Management of contact lists (Enterprise/Personal contacts, Favorites)
UserAssistant	Storage/Retrieval of modes (e.g. Telephony settings)
ConferenceBridge	Control of a conference bridge on MX (AVAYA Meeting Exchange)
VoiceMessaging	Control of a Voice-Mailbox on MM (AVAYA Modular Messaging)
Presence	Presence Handling (Publication, Subscriptions)

Dependent on configuration and status of systems, resources may or may not be available.

If a user is logged in to 1XS (which means his account exists in ADS and he is provisioned in 1XS), basic functionality of the User resource is always available (otherwise he could not have logged in). The same is valid for ContactLog, ContactList and UserAssistant resources, because they access the 1XS database and/or ADS only.

The other resources maintain connections to background services (e.g. AES/CM) outside of 1XS, which have to be individually configured and which have to be available.

In order to control resources via 1XS, a user has to login (establish a session) and then open a control communications path to the resource. In other words, instances of control objects have to be created on the server, or briefly stated: "**Resources have to be created**".

As mentioned above, resources may be unavailable for creation (e.g. if there is no conference bridge configured for the user or if the connection to MX is down, the resource isn't available).

Resources available for creation are denoted as **Capabilities**.

A 1XS client application usually will determine capabilities for a logged-in user and then create **Resources** for each of the **Capabilities** of interest.

The **User** resource does not have to be explicitly created. It plays a special role and is implicitly created upon successful user login.

2.1.4 Enterprise Applications and Web Applications

1XS client applications are architected as JEE Web Applications. Web applications usually will be packaged in WAR archive files (*.war).
For deployment on a JEE A/S like IBM WebSphere, web applications are part of an Enterprise Application. Content of enterprise applications will be packaged in an EAR archive file (*.ear), which serves as an envelope for web applications.

Thus, a 1XS client application is a JEE Enterprise Application (delivered as EAR file) which contains a Web Application.

2.2 1X Web Application Framework

1XS client applications, like the SDK's "Dashboard" application, are built on reusable libraries provided by AVAYA and bundled as the Web Application Framework (**WAF**).
You will find WAF files and documentation in the folder *Software-Libraries\1X_WebApp_Framework* of the SDK distribution.

The WAF consists of server runtime libraries delivered as JAR archive files (*.jar) as well as JavaScript libraries to be used in web pages.

2.2.1 Server Modules

Essential WAF server libraries are stored in *Software-Libraries\1X_WebApp_Framework\jars* of the SDK. Any 1XS client application needs to include them all.

The Web Application framework package does not contain additional libraries (e.g., 3rd-party libraries) referenced by the essential libraries "acp-xxx.jar".
A straightforward way to reference the essential libraries in your application is to re-use or copy the libraries (and library inclusion code) from the Dashboard application. The libraries are part of the Dashboard EAR file; you can find them all in *IDE\1X_SDK_Dashboard\EarContent*.

2.2.2 Client JavaScript Libraries

The WAF provides basic JavaScript libraries for the client side of an application. They are located in *Software-Libraries\1X_WebApp_Framework/js* of the SDK distribution.

The libraries contain:

- AJAX communication handling
- Basic functionality, e.g. start of the AJAX system and determination of capabilities
- Resource Adapter "Stub" classes
- Utility functions

The SDK provides different packaging of the libraries:

- A normal version: structured code, comments and documentation included
- A minified version: Comments, documentation and white space stripped off
- A G-Zipped version

2.2.2.1 API Documentation

Reusable libraries of the SDK contain JSDoc (JavaScript Doc) comments, from which JSDoc HTML documents can be generated.

The JSDoc of WAF are found in *Software-Libraries\1X_WebApp_Framework\doc*.

It is recommended to use the combined documentation of WAF and the Dashboard's Support libraries in *Software-Libraries\WAF&SupportLib_JSDOC* of the SDK.

2.2.2.2 Ajax-Helper

The AJAX-Helper library (*avaya_ajax_helper.js*) handles the AJAX communication with the server.

The library contains class definitions and some global constants.

In order to communicate with the server, each application has to maintain a single object of the class **AvayaAjaxHelper**.

There is no need to study the details of this library. Most of its functionality is hidden by the Application libraries (see following Chapter).
Nevertheless, in certain situations, an application has to handle objects or constants defined in the Ajax-Helper library directly.
Therefore you should be familiar with the following definitions:

Class **AvayaAjaxResourceResponse**

Upon call of a resource method, a response is returned. The AJAX-Helper object forwards the response as an instance of AvayaAjaxResourceResponse to a handler provided in the method call.

This response object contains

- Type and ID for identification of the resource
- The operation (the name of the called method) done in the resource API
- The return value of the method call
- A status, which is "*success*" in case of no error
- An **AvayaAjaxError** object, which is populated if the method call was not successful

Class **AvayaAjaxEvent**

Events fired by a resource are forwarded to registered event handlers as instances of AvayaAjaxEvent.

An event object contains

- Type and ID for identification of the resource
- A name for identification of the event type
- The event data, an array of key/value-pairs which represent the XML content of the event message
- A "Batch-ID".
  The AJAX system receives messages by polling. More than one message waiting in the server (response or event) may be transferred in a poll operation. All messages of a poll contain the same "Batch-ID" (a running number) for identification.
  This enables clients to reduce UI updates by doing necessary updates only upon changes of the Batch-ID (NOTE: The Dashboard application does not use this mechanism).
- A "Handback" object.
  Upon registration for event reception, an additional handback- object may be set. This object is returned to the listener as attribute of the event object (NOTE: the Dashboard application does not use this mechanism).

Resources may become unavailable during runtime of a user's session. This is especially valid for those resources provided by AVAYA server applications outside of 1XS.
Therefore 1XS client applications working with resources should track their state.

The state of a resource can be one of "*not initialized*", "*available*", "*unavailable*" or "*impaired*".

Appropriate JavaScript constant definitions are

- **AAH_RESOURCE_STATE_XXX** in avaya_ajax_helper.js
- **AcpResource.STATE_XXX** in acp_client.js
  Applications should prefer to use these !

2.2.2.3 Application Libraries

Besides Ajax-Helper, client functionality provided by WAF is contained in 3 libraries:

- **Client library** (*acp_client.js*)
  Comprises classes for the central, unique "Client" object (**AcpClient**) of each application and the aforementioned resource "Stubs".
- **Browser-Client library** (*acp_browserclient.js*)
  Contains a "Browser-Client" class definition (**AcpBrowserClient**) for functionality targeted at browser-based clients.
- **Application Library** (*acp_application.js*)
  This library is not a part of WAF (WAF provides a sample only). Each application has to establish its own application library and define an "Application" class (**AcpApplication**) in it. This class contains specific handling of AJAX communication errors and changes of connection state. Other centralized functionality of an application may be added here.

AcpApplication is a subclass of **AcpBrowserClient** which is a subclass of **AcpClient**.

An instance of a subclass of AcpClient (usually AcpApplication) plays the role of a unique, central object of an application.

2.2.2.3.1 Client Library

acp_client.js contains the class **AcpClient**, a base class which incorporates most of the basic central functionality an application is built upon. That is:

- An instance of **AvayaAjaxHelper** for communication with the server and methods for initialization, start, and stop of the communication.
- An instance of the class **AcpUser**, which represents the User-resource "Stub". This resource does not have to be created, it is automatically available after successful user login.
- Abstract handlers for AJAX communication state and errors.
  AcpClient registers itself in its **AvayaAjaxHelper** attribute to track the AJAX communication status. The handlers should be redefined by a subclass.
- Methods for access of some basic application parameters like the context path (common part of the web application's URL), server-name and -port, version, etc.
- Storage for request parameters received upon application start (as "Query" string).
- Storage for strings to be used for localization purposes in software running on the browser (JavaScript).
- A facility which is able to notify registered "listener" objects (usually windows) upon creation or removal of resources.
  The listener objects may register with *addResourceCreatedListener()* resp. *addResourceRemovedListener()*, they have to implement methods *handleResourcesCreated()* resp. *handleResourceRemoved()*.
  Consider the following use case: A window of an application is able to create/remove resources during runtime, while other application windows want to get information if a creation/removal happens; then they should register as listeners.
- Methods for start/stop of a "**Transaction**".
  A transaction is a means to temporarily increase the AJAX polling speed.
  Think of a series of resource commands which have to be emitted. The next request shall be sent upon reception of the response to the previous command. Normal AJAX polling speed would make this a lengthy operation. The "Transaction" mechanism may be used to achieve more immediate action.

- Methods to enable/disable the "Debug" window feature of the AJAX-Helper and to write to this window.
  The Ajax-Helper object writes raw XML communication to the debug window.

Furthermore, the client library contains "**Resource Stub**" classes which are the counterparts of the remote Resource-Adapter API on the client browser. A "Stub" class exists for each resource.
All these classes inherit from **AcpResource**.

Resource	JavaScript Resource "Stub" class
User	**AcpUser**
Extension	**AcpExtension**
ContactLog	**AcpContactLogger**
ContactList	**AcpContactList**
UserAssistant	**AcpUserAssistant**
ConferenceBridge	**AcpConferenceBridgeControl**
VoiceMessaging	**AcpMailbox**
Presence	**AcpPresence**

Classes and methods which carry "Bean" in their names (e.g. "AcpBean" or "<AcpClient-obj>.getBeanById()") should be ignored.
These "Beans" are AVAYA-private server objects used by the 1X Portal Client, an Avaya product which is supported by the WAF. They are out of the scope of the SDK.

Class **AcpUser** plays a special role. There is no need to create an instance of it, because after login and start of the AJAX server communication, a unique "user" object is automatically made available as an attribute of the unique "client" object. You can get a reference to it by executing *<AcpClient-object>.getUser()*.
AcpUser is not simply a "Stub" which provides methods to be called on the server-side User resource. It is also convenience class for functionality in the logged-in user's domain. **AcpUser** offers:

- "Stub" methods for retrieval/adjustment of user and other resource settings. "**Resource Settings**" are parameters of a resource. They are preset by system configuration and may be modified by using the "1XP Admin Client". Some of the settings may be updated by a client application.
- "Stub" methods for retrieval/adjustment of "**Application Settings**". Application Settings are arbitrary application-specific parameters, e.g. the size and position of a browser window at startup. Each application may define application settings according to its needs.
- "Stub" methods for handling of presence ACL lists (Access Control Lists).
- Methods for creation of instances of "Stub" objects (AcpXXX) from a known resource type or ID (*getResourceByType()*, *getResourceById()*).
- Methods for retrieval of "Capabilities" and created "Resources" from the AJAX-Helper. Returned Capabilities and Resources are wrapped in **AcpCapability** resp. **AcpResource** objects.
- A method *createResources()* for resource creation from known "Capabilities"
- Various utility methods which serve as quick checks for a configuration, e.g. *hasTelephony()* checks whether the telephony resource is available for the logged-in user.

Note:
The way to call a method on a remote resource adapter is:

- Get a resource "stub" object via the user object (e.g. by using a resource type constant).
- Call the method and provide a resource-response handler as call parameter.
- Handle the response and any events fired by the resource.

Resource type name constants are defined as attributes of class **AcpResource**:

Resource	Resource-Type
User	**AcpResource.TYPE_USER**
Extension	**AcpResource.TYPE_TELEPHONY**
ContactLog	**AcpResource.TYPE_CONTACT_LOG**
ContactList	**AcpResource.TYPE_CONTACT**
UserAssistant	**AcpResource.TYPE_USER_ASSISTANT**
ConferenceBridge	**AcpResource.TYPE_CONFERENCE_BRIDGE_CONTROL**
VoiceMessaging	**AcpResource.TYPE_VOICE_MESSAGING**
Presence	**AcpResource.TYPE_PRESENCE**

Name definitions of resource settings resp. "Properties" of interest (a property is a key-name/value pair) can be found in AcpResource as constants **AcpResource.PROPERTY_XXX** or (more specifically) in the "Stub" classes as constants **<AcpXXX-class>.PROPERTY_XXX**.

Because of the inheritance from **AcpResource**, the "Stub" classes have some basic characteristics beyond provisioning of methods to be called on the server:

- They offer methods for registration/deregistration of event-listener objects in the AJAX transport layer (*addListener()*, *removeListener()*).
- You may obtain the resource's unique ID, its type and its "Properties" from them. Properties may change during runtime (e.g. caused by modifications via 1XP Admin Client), clients are able to automatically track changes via event listeners..

At the top of the client library file (*acp_client.js*) you find possible Resource Event names as globally defined constants **ACP_EVENT_XXX**. The names allow for discrimination of the events sent by 1XS.

2.2.2.3.2 Browser Client Library

acp_browserclient.js contains the class **AcpBrowserClient**, a subclass of **AcpClient**.

AcpBrowserClient adds some functionality which is especially useful for browser-based 1XS clients.

2.2.2.3.3 Application Library

The Application Library *acp_application.js* is not part of the framework, it has to be provided by each 1XS client application.

This library at least has to contain the class definition **AcpApplication** which is a subclass of **AcpBrowserClient**.

An instance of AcpApplication plays the role of a unique, central object of an application.

AcpApplication should implement the application-specific handlers *handleAjaxError()* and *handleAjaxConnectionStateUpdate()* for AJAX communication errors and state changes.

Note:
JavaScript does not support class inheritance in the classical way such as programming languages like Java. For method overwriting, it is insufficient to redefine a method in a subclass (as one would in Java). You have to replace methods programmatically. A comprehensive description of class inheritance for JavaScript is beyond the scope of this document. The interested reader is referred to the many references on the JavaScript language available in print or on the web.

AcpClient includes methods *setErrorHandler()*, *setConnectionStateHandler()* for replacement of the handlers.
In JavaScript code, the method replacement mechanism may appear to be awkward to an experienced Java programmer. For example:

```
var myAcpApp = new AcpApplication();
myAcpApp.setErrorHandler(myAcpApp);
myAcpApp.setConnectionStateHandler(myAcpApp);
```

2.2.2.4 Runtime Constant Definitions and the Utility Library

WAF and other parts of a 1XS client application make use of some quasi-constant JavaScript variables which can be determined at runtime only:

- Constant definitions derived from server-side static attributes of Java classes
  (e.g. a product version code)
- Values derived from a browser's HTTP request
  (e.g. name/IP address of the server or the client workstation which sends the request)

Runtime constants are defined in *acp_runtime_constants.js.jsp*. This file contains Java code and, as a JSP file, it is compiled into a Servlet by the WebSphere A/S. The servlet delivers JavaScript code determined during runtime.

As part of WAF, the library *acp_utils.js* contains basic utility functions. Some of them are accessed by the Ajax-Helper and Application libraries and therefore are essential.

The Dashboard application does not make use of this library. It comes with its own utility library which meets requirements of WAF libraries.

2.3 Reusable Dashboard Libraries

Some reusable libraries emerged from development of the Dashboard application.

These "**Support**" libraries are part of the Dashboard and can be found in folder *IDE\1X_SDK_Dashboard_war\WebContent\js\reusable* of the SDK distribution.

Although part of the Dashboard, the libraries are independent from other Dashboard code.

2.3.1 Support JavaScript Libraries

The "Support" libraries are built on top of WAF and should be interpreted as an additional event-handling layer which provides

- Event-to-object transformation (e.g. translation of a received "communication" event into a "Call" object for handover to the telephony handling part of an application)
- Caching of event data (e.g. store a "Call" object in a list and update it upon "communication-change" events)
- Basic plausibility checks
- Calling of notification handlers if data changes occur

An application which uses the support libraries does not have to establish event listeners which receive and decode AJAX event messages. Instead, it uses support objects, waits for their notifications and deals with more abstract objects like a "Phone-Call", a "Conference on a Bridge", a "Contact", etc.

For detailed API information refer to the combined documentation of WAF and Support libraries in *Software-Libraries\WAF&SupportLib_JSDOC* of the SDK.

Currently existing support libraries are

Resource	Support Library	Support Class
Extension	*telephony_support.js*	**TelephonySupport**
ContactLog	*contactLog_support.js*	**ContactLogSupport**
ContactList	*contactList_support.js*	**ContactListSupport**
ConferenceBridge	*bridgeconf_support.js*	**ConferenceBridgeSupport**
VoiceMessaging	*messaging_support.js*	**MessagingSupport**
Presence, User (Presence handling)	*presence_support.js*	**PresenceSupport**

Support libraries all have a similar structure. They contain some class definitions, the most important of them is the "xxxSupport" class.
In order to use a support library, an instance of the support class has to be created and initialized by calling the *initialize()* method.
Typical initialization parameters required (differs from class to class) by a support object is:

- A reference to the unique "user" object, if for some reason the current user has to be identified.
- A reference to a resource stub. The support object has to register as event-listener in the AJAX transport layer via this object.
- A reference to a UI-Notification handler. Usually this is a window object which implements the notification handler method appropriate for the support object.
- Selected options: Some of the support objects are able to load basic data from the server upon initialization.

Support objects maintain internal lists for data received via events from 1XS.

Support Object	Functions
TelephonySupport	Maintains a list (**PhoneCallList**) of **PhoneCall** objects which represent the user's current telephony calls. Several connections (**PhoneConnection** objects) may belong to a call (two (2) for a simple call, more for a conference). Events from 1XS add, modify and remove the call objects.
ContactLogSupport	Maintains a list (**ContactLogList**) of **ContactLog** objects. Events from 1XS add, modify and remove the log items.
ContactListSupport	Maintains a list (**ContactList**) of **Contact** objects. Each contact contains several contact addresses (**ContactAddress**). Events from 1XS add, modify and remove contacts.
ConferenceBridgeSupport	Maintains a **BridgeConference** object which represents a conference on a bridge and which contains a list of **ConferenceParticipant** objects, the current participants of the conference. Events from 1XS start or end the conference and add, modify or remove participants.
MessagingSupport	Maintains a list (**MessageList**) of **Message** objects. Each message contains several message parts (**MessagePart**). Various handler classes (e.g. **GetMessagePartHandler**) ease access to message parts.

PresenceSupport	Maintains two (2) lists: a list of the logged-in user's subscriptions (**SubscriptionList**) for reception of other user's presence information and an authorization list (**AccessControlList**) which controls if others receive the logged-in user's presence info. The subscription list contains subscriptions as **PresenceSubscription** objects. The subscriptions are updated upon events from 1XS. In addition to basic presence info of a monitored user, a subscription may contain several items of information concerning the user's devices (**DevicePresence**). The access control list contains authorization items in form of **AclEntry** objects which are added, updated or removed upon 1XS events.

If data in the lists of a support object change, the object will notify other parts of the application (typically software parts which display information in a UI) by calling a UI-Notification handler method. These methods have predefined names *uiHandle<XXX>Notification()*, e.g. *uiHandleTelephonyNotification()*.

A notification handler method call contains a single parameter, the notification object. Each support class sends its own type of notification objects, the classes are named **<XXX>UINotification**, e.g. **TelephonyUINotification**.

A notification contains

- A type code (the meaning of the notification)
  Possible types are defined as constant attributes **<notification-class>.TYPE_XXX** of the notification class.
- Error information (in case of errors).
- The object (from the support object's internal lists) which was added, removed or has changed.

Support objects extract data from 1XS events. Events are messages with XML content, pieces of information in them are transported in form of key-name/value pairs.

Some of the events contain an "action"-- or "operation"-- field for closer classification.
Certain field value codes have to be known for decision making (e.g. status values).

Key names for extraction, operation codes and certain values used by the support objects are defined as constant attributes of the support class, e.g. **<support-class>.KEY_XXX** or **<support-class>.OPERATION_XXX**.

Many event message delivered by 1XS now contain "address counts" which represent the number of available adresses (e.g. phone call numbers, Email addresses, voicemail addresses) per type.
The support libraries currently extract address counts from events for contact-list and messaging events only.

2.3.2 Utility Library

The Dashboard's library of utility functions *utils.js* can be found in folder *IDE\1X_SDK_Dashboard_war\WebContent\js\reusable* of the SDK distribution.

It is a collection of general-purpose functions.
Each function name ends in "_Util", except some functions extracted from the previously mentioned WAF library *acp_utils.js* (which isn't used by the Dashboard, therefore functions referenced by WAF are cloned).

2.4 Application Installation, Initialization, and Start Considerations

The previous description of basic concepts of 1XS-based applications and of reusable JavaScript libraries and their essential class definitions serve as starting point for an introductory overview on structure and runtime aspects of such an application.

2.4.1 Provisions for Installation and Operation

As mentioned before, a 1XS client application is an Enterprise Application for the IBM WebSphere A/S. The application is packaged as an **EAR** archive file (*.ear).

The enterprise application is a container for a Web Application which is packaged as a **WAR** archive file (*.war).

Both types of archives collect information on how they have to be installed and operated by the Application Server in the form of **Deployment Descriptors**.
Deployment descriptors are files with XML content. They have well known names and are stored at well known positions.

Many parts of the descriptor content are essential and mandatory for 1XS client applications. Therefore it is recommended to use the Dashboard's descriptors as a template for new applications.

Deployment descriptors are discussed in more details in Chapter 2.5.

2.4.2 Common Web Application Framework Initialization

1XS client applications are built on top of the WAF framework.

Upon start of an application by the WebSphere A/S, application context specific parts of WAF have to be initialized (e.g. objects for user session management).

The client's web application (WAR) uses the **Spring Framework** (www.springsource.org) to perform the initializations.
The Spring Framework library *spring.jar* is one of the additional 3rd party archives in the Dashboard EAR project (*IDE\1X_SDK_Dashboard\EarContent*).

Spring initialization tasks are defined in *applicationContext.xml*, a file which is located in the Dashboard's WAR project folder *IDE\1X_SDK_Dashboard_war\WebContent\WEB-INF*.

Initialization tasks are equal for any 1XS client web application. Therefore the file has to be used for any application in unmodified form.

2.4.3 The Login Process

1XS client applications are session-based. Therefore Cookies have to be enabled in the web browser. WebSphere automatically sets the Cookie **JSESSIONID** upon first request to the server.

Login to an 1XS client application is a 2-step process.

1. Step:

A user requests a known start page of the application (optionally with some start parameters added to the URL). The start page URL is protected by a security constraint, therefore the user is redirected to a login page provided by the application.
Security constraints and the login page are declared in a deployment descriptor.

The user now has to login to WebSphere by calling the URL */j_security_check* with parameters **j_username** (user login name) and **j_password** (user password).
WebSphere checks the user's authorization via the directory service ADS.

If the authentication was successful, WebSphere redirects to the start page again and adds those parameters found in the initial start page request.

2. Step:

A specific WAF framework class **AcpPostLoginFilter** has to be set as filter for requests to */j_security_check* (in a deployment descriptor). Upon successful user authentication, the filter object checks if the user is a valid user, provisioned on 1XS. If not, login is denied.

Upon successful login a second Cookie **LtpaToken** is set in the user's browser.
Both cookies have to be passed to the server upon all requests in order to stay authenticated.
This will automatically be done by the AJAX transport layer.

Usually access to the login page is configured as encrypted (TLS, set in a deployment descriptor).
In the described login sequence, WebSphere falls back to "not encrypted" upon redirection to the start page.

2.4.4 Server Communication Start

After login, the main page of an application has to be loaded. This can be the start page itself or a page loaded by the start page.
The main page is loaded once, all subsequent communication is done via the AJAX layer, and page content updates have to be done by JavaScript (DHTML).

An applications main page has to create a single instance of the class **AcpApplication**. The application object contains an **AvayaAjaxHelper** object as AJAX communication handler.

The next steps an application has to perform are:

- Call the application objects *start()* method for initialization of the AJAX system.
  If the communication is started, the callback method *handleAcpClientStarted()* of the main page is triggered.
- The unique "User" object ("Stub") is available now and can be accessed via *<application-object>.getUser()*.
  Available "**Capabilities**" can be read from the user object.
- The application may create "**Resources**" at its needs from available capabilities by calling *<application-object>.getUser().createResources()*.
  To track the resource creation, the main page has to register in the application object as "resources-created" listener via *<application-object>.addResourceCreatedListener()*.
- Once selected resources are created, the main pages' handler *handleResourcesCreated()* is called.
- Resources are available now and ready to use.
  Calling of resources methods is done via a resource "Stub" object, which can be obtained from the user object via
  <application-object>.getUser().getResourceByType(<resource type code>).
- As recommended option: The application creates instances of "Support" objects which receive resource events and translate them into easy-to-handle objects.

2.4.5 Logout

A user is logged out from WebSphere by request of the URL */ibm_security_logout*. He will be redirected to a page provided in parameter *logoutExitPage* of the request.

2.4.6 Start Performance

Due to their AJAX-based communication scheme (i.e., web pages will not be reloaded), 1XS client applications generally have high performance.

The relatively large amount of JavaScript code loaded at application start reduces startup-time performance.
Another performance issue concerns the number of JavaScript files. Browsers usually load them sequentially.

Startup performance may be increased by

- Making JavaScript code as compact as possible (no comments, no white space):
  - Use "minified" WAF libraries
  - Compress your own code (free compressor tools are available, e.g. the YUI Compressor (http://developer.yahoo.com/yui/compressor/)
- Combining JavaScript code in a small number of files

As a training tool, the Dashboard application does not make use of these approaches.
Its target is to provide well structured, understandable, and debuggable code.

2.5 Application Deployment Descriptors

Deployment Descriptors define how an application has to be handled by the Application Server. They affect installation, initialization and runtime features.

Because of the framework (WAF) usage, many aspects of the deployment descriptors are mandatory for an 1XS based client application. This chapter addresses the details.

It is recommended to use the Dashboard's descriptor files as template for new applications.

2.5.1 Enterprise Application Descriptor

The deployment descriptor of the Enterprise Application (EAR) resides in subfolder *\META-INF* of the application. It consists of the standard JEE descriptor file and some IBM WebSphere-specific extensions.
The descriptor provides information about contained Web Applications, security environment and application start.

2.5.1.1 Standard Descriptor

The file *application.xml* represents the JEE standard deployment descriptor.

```
<application ….. id="Application_1XSDashboard" …..>
  <module id="WebModule_1XSDashboard">
    <web>
      <web-uri>1X_SDK_Dashboard_war.war</web-uri>
      <context-root>/dashboard</context-root>
    </web>
  </module>
  <security-role id="SecurityRole_1XSDashboard">
    <role-name>1XP_User_Role</role-name>
  </security-role>
</application>
```

It defines:

- An arbitrary (but unique) ID of the application
- A contained Web Application module, its ID, WAR archive file and **root context** (this is the root part of the URL of the application's web pages)
- An arbitrary (but unique) security role ID definition and the role's name

2.5.1.2 WebSphere Extensions

WebSphere specific extensions of the deployment descriptor are

- *ibm-application-bnd.xmi*
- *deployment.xml* in subfolder *\ibmconfig\cells\defaultCell\applications\defaultApp\deployments\defaultApp*

You will find definitions like "**xmi:id = xxx**" inside of these files. These id-values are formal definitions. You should adjust them to arbitrary values which have to be unique in the scope of the enterprise application.

ibm-application-bnd.xmi:

```
<com.ibm.ejs.models.base.bindings.applicationbnd:ApplicationBinding …>
  <authorizationTable …>
    <authorizations …>
      <role href="META-INF/application.xml#SecurityRole_1XSDashboard"/>
      <groups … name="onexuser"/>
    </authorizations>
  </authorizationTable>
  <application href="META-INF/application.xml#Application_1XSDashboard"/>
</com.ibm.ejs.models.base.bindings.applicationbnd:ApplicationBinding>
```

It defines:

- A security group name for the security role defined in the standard descriptor.
  This group name has to match the user group name defined upon installation of the 1XS system. It is the group of 1XS users in ADS.

deployment.xml:

```
<appdeployment:Deployment …>
  <deployedObject … startingWeight="20" warClassLoaderPolicy="SINGLE">
    <modules … uri="1X_SDK_Dashboard_war.war"/>
    <classloader … mode="PARENT_FIRST"/>
  </deployedObject>
</appdeployment:Deployment>
```

It defines:

- The position in the sequence of started applications upon start of the WebSphere A/S.
  The startingWeight value should not be reduced because essential 1XS applications have to be started before 1XS client applications.

2.5.1.3 Dependencies

There are some references among deployment descriptor files which have to match.

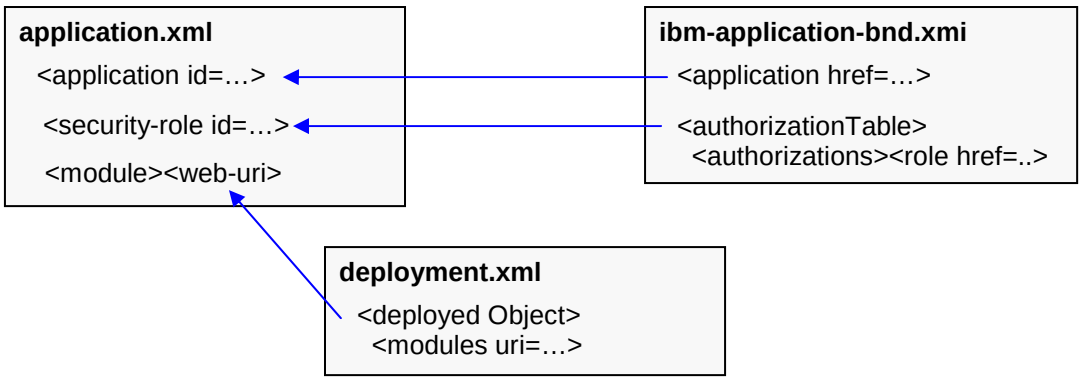

FIGURE 4: Enterprise Application Deployment Descriptor Dependencies

2.5.2 Web Application Descriptor

The deployment descriptor of the Web Application (WAR) resides in subfolder */WEB-INF* of the web application. It consists of the standard JEE descriptor file and some IBM WebSphere specific extensions.
The descriptor contains various information like login configurations, Servlet definitions and their mappings to URLs, security settings for different parts of the application (different URL patterns), etc.

2.5.2.1 Standard Descriptor

The file *web.xml* represents the JEE standard deployment descriptor.
Attributes and their meaning are listed below (column **M** specifies whether attributes are mandatory and have to be set to predefined values).

Item	Functions	M
context-param	Reference to the **Spring Framework** configuration file *applicationContext.xml*	y
filter **filter-mapping**	**PostLoginFilter** for 1XS user check during login and mapping of this filter to Websphere's login address */j_security_check*	y
listener	Listener objects for the WAF framework: Spring context loading and HTTP session listening	y
servlet	Servlet object definitions. Mandatory servlets (parameters included) are: **1XS Servlet** which is the AJAX communication counterpart on the server. **1XS File Transfer Servlet** for down-/upload of messaging audio or attachment files.	(y)
servlet-mapping	Mapping of the servlets to URLs. Mappings of mandatory servlets are mandatory too.	(y)
welcome-file-list	Automatically selected default web page file(s) for browser requests which specifiy a folder but not a file.	
error-page	URLs of error-pages (including optional parameters). The server redirects to these pages upon specified error types.	
security-constraint	Security constraints define whether access to specific areas of an application (URL patterns) require login and if requests to web pages have to be encrypted. 1XS requires login, therefore the **auth-constraint** of all URL areas (defined by **web-resource-collection** items) of the application should be set to the Security-Role name defined in the EAR deployment descriptor. It is recommended to handle the login page differently: Requests should be encrypted by setting a special **user-data-constraint**, the **transport-guarantee**, to "CONFIDENTIAL".	y
login-config	A login page **form-login-page** for FORM-based login has to be specified. WebSphere redirects to the login page upon attempts to access protected application areas without being logged in. Furthermore, a login error page **form-error-page** has to be configured.	y
security-role	A reference to the **role-name** defined in the EAR deployment descriptor.	y
resource-ref	Definition of resources provided by the WebSphere A/S and used by 1XS: **ClientWorkManager** and **ClientAPIWorkManager**	y

2.5.2.2 WebSphere Extensions

WebSphere specific extensions of the deployment descriptor are

- *ibm-web-bnd.xmi*
- *ibm-web-ext.xmi*

You will find definitions like "**xmi:id = xxx**" inside of these files. These id-values are formal definitions. You should adjust them to arbitrary values which have to be unique in the scope of the enterprise application.

ibm-web-bnd.xmi:

```
<webappbnd:WebAppBinding …>
  <webapp href="WEB-INF/web.xml#WebApp_1XSDashboard"/>
  <resRefBindings jndiName="wm/ClientAPIWorkManager">
    <bindingResourceRef href="WEB-INF/web.xml#ResourceRef_0001"/>
  </resRefBindings>
  <resRefBindings jndiName="wm/ClientWorkManager">
    <bindingResourceRef href="WEB-INF/web.xml#ResourceRef_0002"/>
  </resRefBindings>
</webappbnd:WebAppBinding>
```

It defines:

- Bindings of resources referenced in *web.xml* to WebSphere resources via JNDI names.

ibm-web-ext.xmi:

```
<webappext:WebAppExtension … reloadInterval="3" reloadingEnabled="true"
   defaultErrorPage="/errorNot1XSUser.jsp" additionalClassPath=""
   fileServingEnabled="true" directoryBrowsingEnabled="false"
   serveServletsByClassnameEnabled="false">
  <webApp href="WEB-INF/web.xml#WebApp_1XSDashboard"/>
  <jspAttributes name="reloadEnabled" value="true"/>
  <jspAttributes name="reloadInterval" value="10"/>
</webappext:WebAppExtension>
```

It contains:

- Configuration parameters for the WebSphere JSP engine
  (see WebSphere documentation or search for " ibm-web-ext.xmi" in
  http://publib.boulder.ibm.com/infocenter/wsdoc400/v6r0/index.jsp)

2.5.2.3 Dependencies

There are some references among deployment descriptor files which have to match.

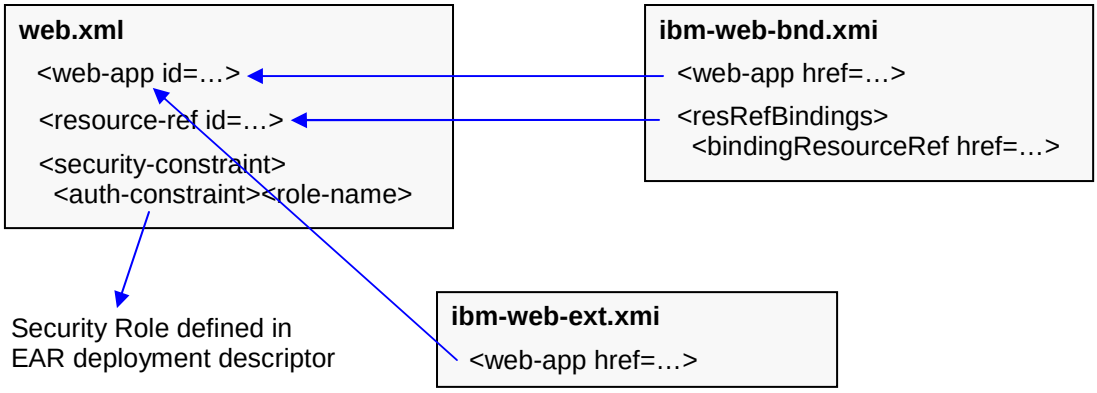

FIGURE 5: Web Application Deployment Descriptor Dependencies

2.5.3 Web Application Manifest

The Manifest file *MANIFEST.MF* in folder *\META-INF* of the Web Application declares which of the library JAR modules of the EAR application have to be added to the Java Classpath of the web application at runtime.

Note: Manifest files require a mandatory Blank character before line breaks.

3 Dashboard Application Basics

The previous chapters established the background for a description of the **Dashboard** application. They presented 1XS client application concepts, reusable libraries (WAF, Support libs) and general aspects of installation and initialization.

The Dashboard is a tool for demonstration of 1XS API features.
It enables developers to explore API handling in detail and reuse any part of the software code according to their needs. It also may serve as basis for derived applications.
The Dashboard's code is freely available as part of the SDK. It places emphasis on

- being understandable (it is extensively documented)
- being simply structured

This chapter will present the layout of application files and show where to search for code references (in most of the JavaScript files).
Furthermore, it introduces to the client/server message tracking facility.

From the server perspective, the Dashboard is a single application running on a WebSphere A/S. Actually, from a browser client perspective, the Dashboard is a set of independently startable applications, or "facets" of an application.
Besides a **Main Dashboard**, some more simple **Samples** exist. These simple applications are a good starting point to investigate how 1XS client applications work.

It is highly recommended to study the demo applications inside of the Dashboard before developing your own applications which use the 1XS JavaScript client libraries..

As mentioned before, the Dashboard Enterprise Application (EAR) is a container for a Web Application (WAR) and some reusable JAR libraries.
Dashboard web pages and logic are part of the web application.
Therefore, in the following, the focus is on the web application.

The web page technology leveraged in the Dashboard is "**Java Server Pages**" (JSP). JSP pages are automatically compiled into servlets (executable Java software) by the WebSphere A/S. The servlets deliver HTML upon request by client browsers.

3.1 Web Application Folder Structure

Folder *WebContent* of the application represents the "**context-root**" (recall the definition in the EAR application's deployment descriptor). A URL, which consists of the server's address/port and the common URL part ("context-root"), points to this folder.

The root folder contains a login page and some error pages.

Subfolders of the root folder are:

console	Console page, associated dialogs and scripting content for the Main Dashboard application
js	General scripting content for the Main Dashboard application. Reusable or optional script files inside of subfolders: **3rdparty** Some 3-rd party libraries, solely for the message-tracking facility **reusable** Reusable Dashboard libraries ("Support"-libs, Utility-lib) **waf** Reusable libraries of the "Web Application Framework" (WAF)
pages	Subpages of the Main Dashboard application's console page: **bridgeconf** Page/scripts for MX bridge conference handling **contactlist** Page/scripts for contact-list handling **contactlog** Page/scripts for contact-log (call log) handling **presence** Page/scripts for presence handling

	telephony	Page/scripts for telephony call handling
	messaging	Page/scripts for messaging
res	Stylesheets and images	
samples	Independent simple demonstration applications:	
	demo1	A single page demo application
	demo2	A single page demo application
	demo3	A single page demo application
	template	A dual page (console + subpage) demo application
tools	Additional tools (AJAX-Tester)	

3.2 Where to find References in JavaScript Code

Web page (JSP) and JavaScript code are almost completely separated in the Dashboard. Main scripts of a page are provided in a single file named similarly to the JSP-page file (e.g., scripts for a page *login.jsp* are contained in *login.js*).

All pages initialize their scripts in an identical way: they call a function *body_OnLoad()* upon page load.

During exploration of the Dashboard's code by means of an IDE (the **Eclipse** package delivered with the SDK), a typical situation will come up:
A JavaScript file references content of another JavaScript file and you want to find the referenced definition.
The IDE can help in a limited way only, and workspace-wide searches are annoying.

The Dashboard's code is designed to be "exploration friendly"; it does not distribute collaborating code among many different files.

Here is a list of files you should scan in order to find things:

- The main script file for the application area of interest, e.g. *telephony.js* for the page *telephony.jsp*.
- The most referenced file is ***\js\waf\acp_client.js*** (basic functionality of the unique "Application" object, resource-adapter "Stub" class definitions and various constants as global definitions or class attributes).
- Sometimes, but rarely, referenced are *\js\waf\avaya_ajax_helper.js* (resource-response and event classes, some global communication status constants) and *\js\waf\acp_browserclient.js* (a small number of additional web client support methods for the "Application" object).
- Highly referenced are the "Support" library files ***\js\reusable\<xxx>_support.js***.
  For each area of the application there is only 1 file of interest. E.g., if you explore scripts for the telephony page, there are code references to *telephony_support.js* only.
- ***\js\reusable\utils.js*** contains functions which are often used. Most of them can be identified by the trailing "_Util" in the function names.
- A small amount of constants are defined in *\js\waf\acp_runtime_constants.js* (some "quasi" constants determined at runtime, rarely used) and *\js\dashboard.js* (console sub-page identifiers used to switch page visibility and definitions for a popup-dialog usable from everywhere).

Certainly there are some more references among files, comments in code provide additional pointers.

3.3 Start of Application Facets

The start of a user session (login process) and of the AJAX communication with the server was already discussed in Chapters 2.4.3 and 2.4.4.

The login sequence offers the ability to start different application facets merely by using a different URL in the initial browser request.

Necessary preparations in the web application's deployment descriptor include protection of all application URL areas and provision of a login page.

Each time a protected URL is requested by a user which is not yet logged in, the user is redirected to the login page. After successful login, he is again redirected to the initially requested page.

The Dashboard utilizes this mechanism to offer some simple demo applications (let's call them "applications" since they are different from the user's viewpoint) besides the Main Dashboard functionality.

All the application facets use the same login process (via page *login.jsp*), but they have to handle the AJAX server-communication start in their own way; thus, users cannot login to a partial application of the Dashboard via a direct request of the login page.

3.4 Message Debug Window ("DebugPlus")

An important Dashboard feature is the extended client/server message monitoring in decoded form.

As differentiation from the "debug" facility of the AJAX-Helper object (which displays all the raw XML communication in a separate browser window), this feature is called "DebugPlus".

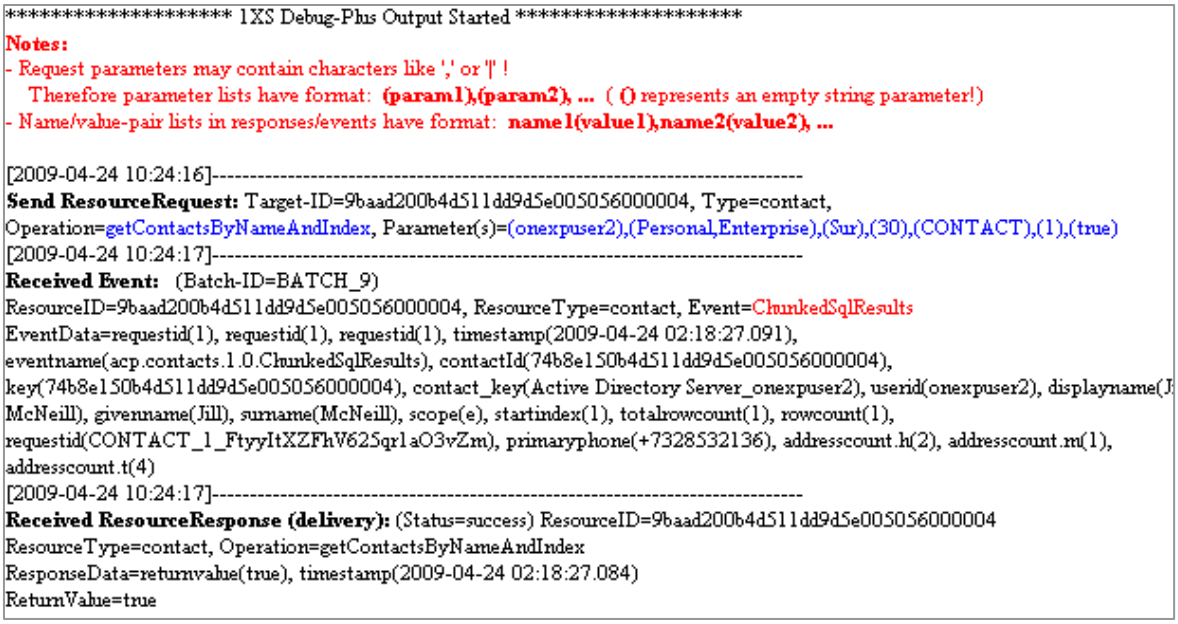

FIGURE 6: "DebugPlus" Output

"DebugPlus" establishes "hooks" for AJAX-Helper methods during runtime (this does not affect the AJAX-Helper source code). The hooks cause a DebugPlus function call each time before a method of interest in the AJAX-Helper object is called. Called functions decode available message information and add it to the content of an open DebugPlus window.

The "DebugPlus" implementation consists if 3 JavaScript libraries:

- *\js\3rdparty\jquery.min.js*
- *\js\3rdparty\aop.min.js*
- *\js\debugplus.js*

Note:

The Dashboard uses the 3'rd party libraries solely for "DebugPlus" implementation.
Applications which do not comprise this feature do not need the libraries.
aop.min.js ("Aspect Oriented Programming") is a plugin for the "jQuery" library.
For details refer to http://jquery.com and http://code.google.com/p/jquery-aop.

To enable "DebugPlus" for an application, the following code has to be added in the script of the application's main page:

```
debugPlusObjRef = new DebugPlus(myAcpApp.getAjaxHelper());
debugPlusObjRef.establishAjaxHelperHooks();
debugPlusObjRef.enableDebugPlus(true);
```

("myAcpApp" is the unique "Application" object).

If "DebugPlus" is enabled, the monitor window is reopened upon every new message.
Thus, close the window every time you want to start with an empty new one.

In some messages you may find single parameters or attributes displayed in the form

(key-1|val-1,key-2|val-2,....key-x|Hello%20to%20all,...key-n,val-n) .

This is caused by the fact that a transported parameter is not a simple value but a list of key/value pairs or an object (which contains some attributes with values assigned to them).

The convention is to separate key/value pairs in the list by a comma and to separate keys from their values by the pipe character. Any arbitrary value text has to be "escaped" for transport to avoid misinterpretations of special characters like "**,**" and "|". The message receiver has to revoke the escaping.

3.5 Simple Demonstration or Template Applications

The Dashboard contains the following Demonstration resp. Template "applications":

- **Demo1:**
  Usage of the ContactList resource. Upon login, the Demo reads contacts with a leading "**o**" in their 1XS username or real name and displays some contact data in a window.
- **Demo2:**
  Usage of ContactList and Extension resources. Upon login, the Demo reads contacts with a leading "**o**" in their 1XS or real name and displays them in a selection box.
  The user may select a contact and place a call by operation of a button "hook-off".
- **Demo3:**
  Usage of the Extension resource. The demo represents a very basic phone application.
  It presents a call number input field, button "hook-off/on" and a status signal.
  The user may place calls to arbitrary extensions or receive calls.
- **Console/Sub-Page Template:**
  This is a template application with a similar structure as the Main Dashboard application:
  A basic console page (which hosts the "Application" object and handles the AJAX communication start) contains as sub-page an intrinsic frame.
  Functionality is equal to "Demo1": Contacts with a leading "**o**" in their names are read and displayed.
  But, there is a fundamental difference. This template does not use a "Support" object for event

processing. Instead, it registers an own AJAX event handler to extract contact data from events (event attribute key name definitions are taken from the support library).

All demo applications implicitly start "DebugPlus" monitoring.

URLs for start of the applications are:

Demo1	*http://<1xs-ip>:9080/dashboard/samples/demo1/demo1.jsp*
Demo2	*http://<1xs-ip>:9080/dashboard/samples/demo2/demo2.jsp*
Demo3	*http://<1xs-ip>:9080/dashboard/samples/demo3/demo3.jsp*
Console/Subpage Template	*http://<1xs-ip>:9080/dashboard/samples/template/console_template.jsp*

(<1xs-ip> = IP-address of the 1X server)

Code of the different samples is quite similar, therefore is doesn't help to describe them all.
The next chapter will go into **Demo3** in more details.

3.5.1 The "Demo3" Application

Demo3 consists of a single web page *demo3.jsp* and scripts in *demo3.js*.
The JSP file includes all needed script libraries. It starts scripts by calling *body_OnLoad()* (see *demo3.js*) when the page is loaded into the browser.

The page layout is:

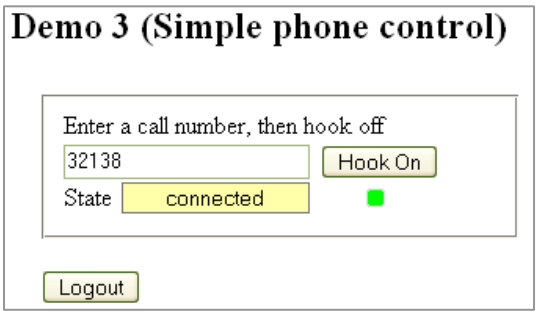

FIGURE 7: Simple Phone Demo Application

The script creates an instance of the class **AcpApplication** (called *myAcpApp*), enables "DebugPlus" message monitoring, starts the AJAX communication and creates an Extension resource.
These activities are already explained in Chapter 2.4.4.

Furthermore 2 objects are created:

- A resource "Stub" (*myExtensionResourceStub*) which allows calling API methods of the remote extension resource-adapter on the server.
  The stub is obtained by
  myExtensionResourceStub =
  myAcpApp.getUser().getResourceByType(AcpResource.TYPE_TELEPHONY)
- A "Support" object (*telephonySupport*) for processing of events.
  The support object is obtained by
  telephonySupport = new TelephonySupport(myAcpApp.getUser(), this)
  The parameter "this" indicates that the page's window is the handler object for notifications fired by the support object. The page's script code has to implement a method *uiHandleTelephonyNotification(uiNotification)*.

The support object has to be initialized by
telephonySupport.initialize(myExtensionResourceStub).

The page registers its window object as an AJAX event handler for resource state or configuration changes. Thus, it is able to take actions if the extension resource becomes unavailable during the logged-in user's session.

Initialization of the application is completed by these steps. All subsequent operations are event driven, they depend on user actions via the UI or notifications sent by the support object.

Events handled by the support object are

- ***CommunicationStarted***: A new call starts
- ***CommunicationEnded***: A call ends
- ***CommunicationParticipantEnter***: A participant enters a call
- ***CommunicationParticipantExit***: A participant exits the call
- ***CommunicationParticipantAttributesChanged***: The state of a participant changes

Event data is translated into objects stored in the support object. A call is represented by a **PhoneCall** object, call participants by a list of **PhoneConnection** objects stored in the call object.

The page window's notification handler method (see above) is called by the support object upon changes. The different notification types reflect the events sent by the server (**TelephonySupport.TYPE_CALL_ADDED**, **...TYPE_PARTICIPANT_ENTER**, ...).
All notifications deliver the current **PhoneCall** object.

The page maintains a global variable *currentCall* as reference to the call (PhoneCall) which is currently processed by the application. The variable is undefined if no call exists.

Unlike the controlled phone (which may support several "call appearances" and therefore can have more than one call at a time), the application does not handle two calls simultaneously.
Once a call is established, "call added" notifications are checked for new calls (calls with different ID) and appearing additional calls are rejected (request *ignoreCall*).

Outgoing call:

- The user enters a call number and clicks the button "Hook-Off".
  A request to establish the call is emitted via the resource stub by
  myExtensionResourceStub.makeCall(<call-nr>, this).
  Parameter "this" means that the page's window is the handler for the resource response, the script has to implement the method *handleAjaxResourceResponse(response)*.
  Note that there is not more to do with responses than to check them for a possible error.
- The server will send an event "CommunicationStarted" and a "call added" notification appears which delivers a new call object (**PhoneCall**).
  The call initially contains a single participant connection (**PhoneConnection**) which represents the caller (the user resp. his extension).
  Each participant connection has its individual "connection-id".
- The call object is modified by subsequent events: A participant connection is added (the called extension) and later changes its state (the called participant answers).
  Besides the state of participant connections (PhoneConnection objects), the call object provides an overall call state ("call-state" and "call-sub-state").
  State definitions are available as constant class attributes of the **TelephonySupport** class.
  The call is answered if state/sub-state are "active"/"connected".
- The server will send an event "CommunicationEnded" if the called party hangs up or if (user operates the "Hook-On" button) the application emits a request to end the call by
  myExtensionResourceStub.hangupConnection(<own connection-id>, this).

Incoming call:

- The server sends an event "CommunicationStarted" and a "call added" notification appears which delivers a new call object (**PhoneCall**).
  The call initially contains two participant connections (**PhoneConnection**) which represent the caller and the called party (the user resp. his extension).
- The call is signaled in the UI (it is "outgoing" and its sub-state is "alerting"), the user can accept the call by operation of the "Hook-Off" button which emits a request by
  myExtensionResourceStub.answerConnection(<own connection-id>, this).
- Subsequent events change the call state to "connected".
  Further functionality is equal to outgoing calls

Note:

This demo application does not adjust the mode settings (Telecommuter, Mobility) of the extension resource after resource creation. Thus, modes already set before (e.g. by another client) are still in effect.

If, e.g., call forwarding was set for the user's main phone, application behavior may be unexpected.

4 Main Dashboard Application

The subject of this chapter is the "Main Dashboard", the facet of the Dashboard application which demonstrates most of the 1XS API abilities in a single, comprehensive UI.

You should have read all the previous chapters as preparation.

Resources supported by the application are:

- User
- Extension
- ContactLog
- ContactList
- ConferenceBridge
- Presence
- Messaging
- UserAssistant **Not supported by the current version !**

4.1 Architecture

The Dashboard utilizes a console/sub-page architecture. The application's UI consists of a basic console page which contains several sub-pages in "intrinsic" frames.
Each of the sub-pages represent a "view" on one of the available resources.
Console and sub-pages are arranged as tabs, only one of the page's UIs is visible at a time. The user may switch among different resource "views" by means of tab selectors.

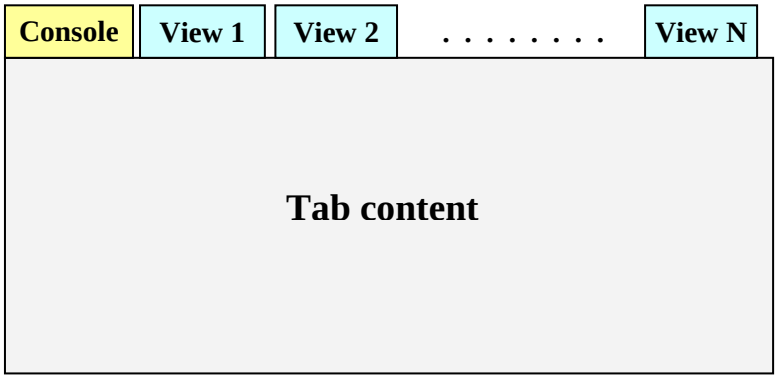

FIGURE 8: Dashboard Console Page Architecture

In order to decrease complexity and increase understandability, each resource is presented separately.
A compact application (like 1XP Client) evidently has many correlations among handling of different resources.
This applies to the Dashboard in moderate form only.
Although there are correlations between resource views (e.g. a call can be made from a contact list item), the basic idea is: Let the user handle dependencies manually.

There is one exception to the general design rules: Messaging cannot be implemented without close correlation to the ContactList resource. Therefore the messaging resource view has access to both resources.

All resource "views" are structured in similar way:

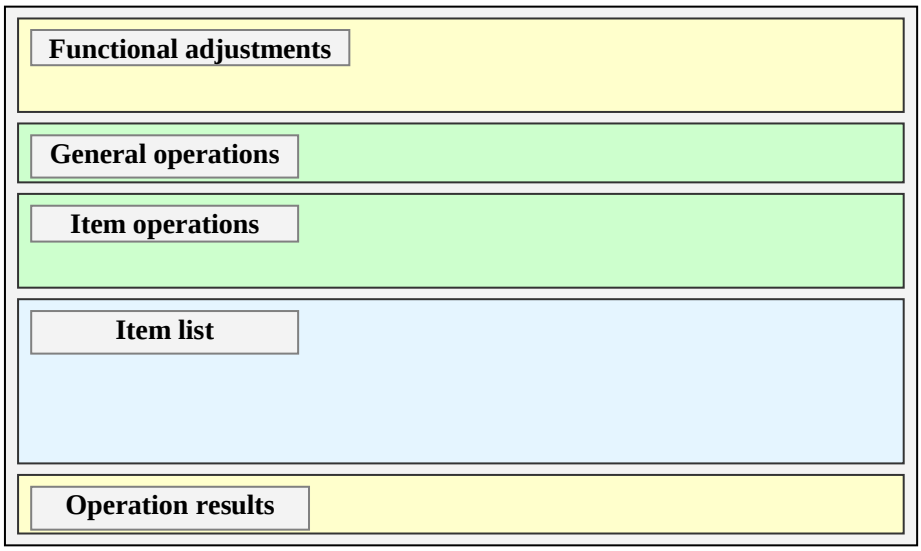

FIGURE 9: Dashboard Sub-Page Sections

They consist of several sections:

- Functional adjustments/configuration of the resource (e.g. switch call transfer mode for a telephony resource)
- Operations to be applied to the resource (e.g. retrieve a list of contact items)
- List of items handled by the resource (e.g. phone calls, contact, or contact-log items).
  The list usually will be updated by event messages received from the resource.
- Operations to be applied to one of the list items (e.g. answer an incoming call, delete a contact).
  The available operations may depend on item state.
- Results of an operation triggered by the user.

Not all of the sections have to be available for each resource view.

On most Dashboard pages, data lists are displayed in tabular form. Table content is updated from received 1XS events and therefore has to be rendered dynamically (JavaScript).
All these tables are implemented by use of the library *\js\dyntable.js*.

4.2 The Start Process

You are already familiar with the process of user login and application start from Chapters 2.4.3, 2.4.4 and 3.3.

The "Main" Dashboard utilizes an additional step in the start process sequence.

A user enters the application by request to a console starter page */console/startConsole.jsp*, is redirected to the login page */login.jsp* (if not authenticated) and, after successful login, is redirected to the console starter page again (all parameter values attached to the URL of the initial request are retained).

The starter page opens the console page */console/console.jsp* automatically and can be closed at the end of the sequence.
As a consequence: Popup blockers must be disabled for the browser.

The use of a popup window offers the possibility to configure the console window's size and decoration. The intent is to present a plain window without menu, tool, and status bars.

Any page navigation or refreshment should be inhibited, as it would break the AJAX communication (which would have to be started again).

As explained before, parameter values (query string) contained in the initial request to the starter page, appear again upon redirection to this page after authentication. The starter page adds the complete query string to the console page load request. Thus, finally the console page receives optional parameters added by users upon start of the client application session.

The Dashboard inhibits starting the console page by direct request because of the missing ability to condition the console window.
The console starter page adds an attribute "*good-start*" to the JSP session object. The console checks this attribute and enforces logout if it is missing (redirection to */consoleStartError.jsp*).

Likewise, starting the Dashboard by direct request to the login page */login.jsp* cannot work without sacrificing the ability to add parameters upon start.
Therefore, if you try to do it anyway, the Dashboard will give you a warning message.

Because of basic differences in session handling by browsers, it may be impossible to start two instances of the Dashboard for different users on the same workstation (using the same browser).
If connected to 1XS, a new instance of Internet Explorer provides a separate session. With Firefox, the new browser instance inherits the existing session. Thus, Firefox does not allow login as two different users.

4.2.1 Launch Options

A user should login to the "Main" Dashboard via the URL:

http://<1xs-ip>:9080/dashboard/startconsole<parameters>

where:

- <1xs-ip> = IP address of the 1X server
- <parameters> = ?<parameter-1>&<parameter-2>& ....<parameter-n>
  Optional parameters, each of them as a pair <name>=<value>

As mentioned before, */dashboard* is the application's "context-root" defined in the deployment descriptor of the Dashboard Enterprise Application (*application.xml*).
The URL fragment */startconsole* represents a servlet defined in the Dashboard Web Application's deployment descriptor *web.xml*. It is equivalent to the console starter page */console/startConsole.jsp* (note that each JSP file is automatically compiled into a servlet by the WebSphere A/S).

The following optional start parameters are available:

Parameter	Value / Meaning
debugplus	**true** Enables a monitoring window for output of decoded 1XS message communication
debug	**true** Enables a monitoring window for output of 1XS message communication in raw XML form
ajax-tester	**console** Enables the AJAX-Tester window for working in parallel to the Dashboard's console **standalone** Enables the AJAX-Tester window for working in standalone mode **true** Same as "console"
keys-context	**true** Revokes standard disablement of certain key functions (e.g. F5 = reload) and of the context menu supported by the browser
presence-disable	**true** Disables support of the presence resource (frequently fired event messages may be disturbing for some investigations)

Setting a parameter's value to "false" is equivalent to omitting the parameter.
Launch parameters may be freely mixed in a single request.

For more information about the AJAX-Tester tool, refer to Chapter 4.5.

4.3 Console Page

The following image shows the Dashboard's "General" window, which actually is the console page window.
By clicking any one of the tab selectors at the window's top, the associated console sub-page (the resource "view") becomes visible.

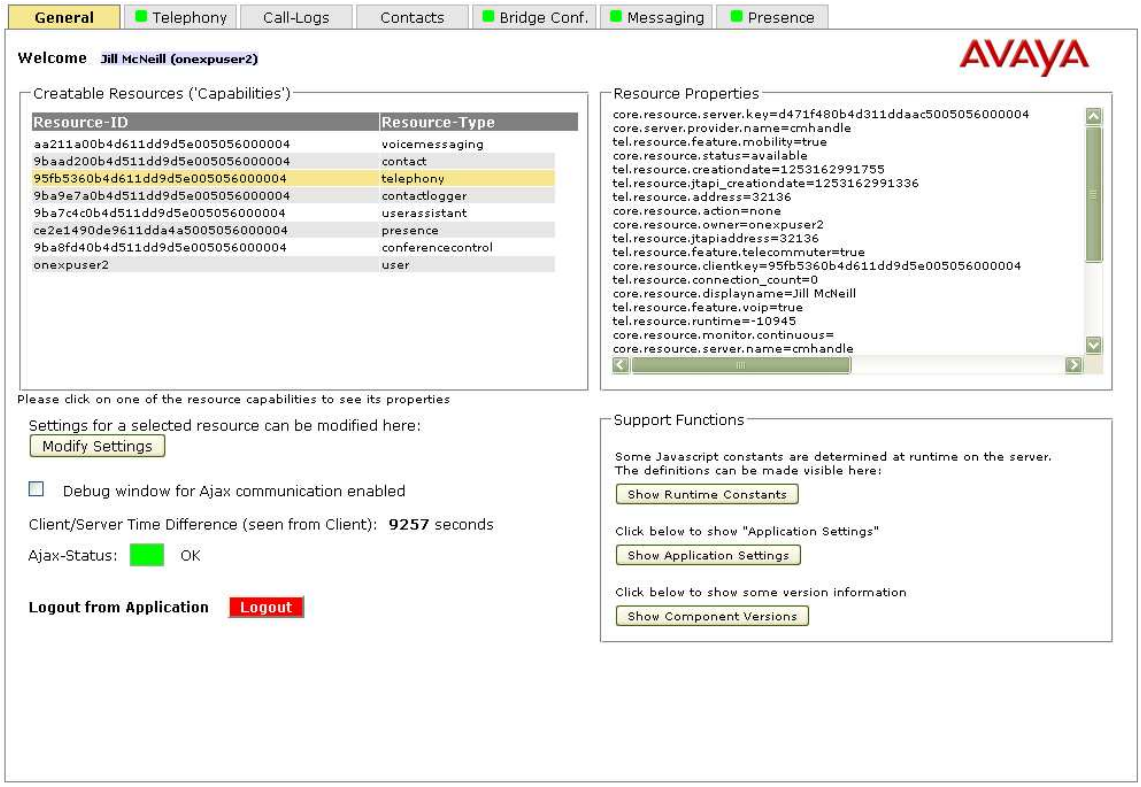

FIGURE 10: Dashboard Console Page

Functionality provided on the console page's "General" tab includes:

- A list of "Capabilities" available for the logged-in user
- Display of property attributes of a capability selected in the list
- Modification dialogs for selected properties of some of the resources (via button "Modify Settings")
- Additional general-purpose functionality

Main script file of the console page is *\console\console.js*.
The console page (or more precisely, its window object) holds an instance of class **AcpApplication** (see *\js\acp_application.js*) as a global object named *myAcpApp*.

This is a central, unique object for the whole application, it handles AJAX communication and creation of resources via its internal instance of class **AvayaAjaxHelper** (see *\js\waf\avaya_ajax_helper.js*).

The console page handles AJAX communication start in the usual way (see Chapter 2.4.4):

- It starts the AJAX layer by *myAcpApp.start(this)*.
- Upon call of the console's start handler function *handleAcpClientStarted()*, it retrieves available "Capabilities" from the "User" object (the implicitly available User resource "Stub" object) via *myAcpApp.getUser().getCapabilities()* and then creates "Resources".

The console provides a simple popup box feature. It may be used for error, warning, and information notifications and also for dialogs of type "yes-no-cancel". This popup box is implemented as an additional layer above the console window. It is modal, i.e., as long as a popup box is visible, the underlying console cannot be accessed by e.g. mouse clicks.

The popup box feature may be used by any part of the software, including sub-pages.

The console page and all its sub-pages include the script file *\js\disableContextAndKeys.js* and initialize it via the global function call *disableContextMenuAndBadBrowserKeys()*.
This script tries to disable the browser's context menu and unwanted handling of keys by the browser (e.g. F5 = reload, F11 = full screen). Its effectiveness depends on the browser type (e.g. disabling of the context menu has to be explicitly permitted on Firefox).

Context menu and key deactivation may be switched off by means of a Dashboard start parameter.

Dashboard pages are not completely independent, there is some communication between them.
The console page, because of its prominent position, offers support for accessing other pages.

- A sub-page may get a reference to another sub-page's window object (in order to call script functions there) by calling the console window's global function *getConsoleTabWindow(<tab-name>)*. Tab name constants are defined in *\js\dashboard.js*.
- A sub-page may set one of the other resource view tabs as the currently visible tab by calling the console window's global function *setConsoleTab(<tab-name>)*.

4.3.1 Resource Handling

The console page creates all available resources except for the Extension resource.

The **Extension** resource is created/removed manually by the user via the "Telephony" sub-page. This process is called "login to"/"logout from" phone on the Telephony sub-page.

Usually a resource is unavailable (not contained in capabilities) if the corresponding AVAYA service (e.g. AES/CM) is unavailable or if the resource isn't properly configured (1XP Admin Client).

The **Bridge Conference Control** resource shows different behavior. As long as AVAYA Meeting Exchange is connected, the resource is available, even if it is not configured for the user.
A valid configuration has to be provided to allow the user to act as a conference moderator.
Although not technically required, the console page checks the server-name property of the Bridge Conference capability and assumes unavailability of the resource if the name is not set.

Creation of resources is done via the User resource stub object, by calling

myAcpApp.getUser().createResources(<capabilities-to-create>)

where <capabilities-to-create> is a list of **AcpCapability** objects for which resources must be created. Each capability is identifiable by a unique ID value. A created resource has the same ID as the associated capability.

In order to track resource creations, the console page permanently registers as a listener for resource creation events (*myAcpApp.addResourcesCreatedListener(this)*).
Upon successful creations, the console page's handler method *handleResourcesCreated(<list-of-*

resource-ids>) is called; then, the console loads corresponding resource "view" pages into its intrinsic sub-page frames and enables the tab selectors.

Furthermore, the console page is prepared to monitor resource state and configuration changes.
The way to implement this is

- Create an instance of the resource event listener class **AcpResourceEventListener** and register it for any of the created resources. The handler object set in this listener is the console page which must implement a method *handleAjaxEvent(event)*.
- The console's event handler method is sensitive to the events "***ResourceStateChangeEvent***" and "***ResourceConfigChangeEvent***" (see constant definitions ACP_EVENT_RESOURCE_STATE_CHANGE and ACP_EVENT_RESOURCE_CONFIG_CHANGE in *js\waf\acp_client.js*)

Resource configuration changes during a user session are simply announced by a popup notification box.
The current resource status is displayed as an icon in the console tab selectors. Additional text appears if the mouse is moved above the icon.

FIGURE 11: Resource State Signals

Possible resource states are (see constants **AcpResource.STATE_XXX** in *\js\waf\acp_client.js*):

- "available"
- "unavailable"
- "impaired"
- "not initialized"

The Dashboard does not take further action if a resource becomes unavailable. For example, the Dashboard does not disable the resource tab if the resource is unavailable. This is by design, because the Dashboard serves as a training tool. Hence, the user should be able to operate with a resource in any state and observe the behavior.

Note:

Use the "1XP Admin Client" to simulate resource state changes. The "Monitors" tab of the tool offers the ability to "suspend" the connection to a selected AVAYA service.

As mentioned above, the console page ("General" tab) displays a list of resource "properties" (key/value pairs) if you select one of the capabilities in the list.
Properties are taken from capabilities if the associated resource is not created; otherwise, properties are taken from the associated resource (and note that created resources may contain more properties than their associated capabilities).

Many of the properties are fixed values derived from 1XS system configuration or from user "**Group Profile**" configuration (see "1XP Admin Client").

Other properties are user-specific and may be modified via the "1XP Admin Client".
Some of them (the user-specific resource configuration) can be modified by the user via the 1XS API.

The Dashboard's console provides dialogs for those properties which can be modified by a user. Click on the button "**Modify Settings**" (if enabled for a selected capability/resource) to open a dialog for a resource.
Properties may be modified via the 1XS JavaScript API by call of

myAcpAp.getUser().setResourceSettings(capabilityId, ...) for any of the resources or by *myAcpApp.getUser().setUserSettings(...)* for the "User" resource.

For additional details about property modification, refer to the following Chapters below.

4.3.1.1 Telephony Account Settings

Imagine there is more than one extension configured for you in a CM system. Only one of the extensions can be your main extension for 1XS. If you want to set another extension as your main one (change your 1XS extension resource configuration), you can do it by using the Dashboard's telephony account dialog (*/console/accountDialog.jsp*).

The account dialog is multifunctional, it is used in "TELEPHONY" mode.

The following properties can be modified:

- *core.resource.displayname* A display name
- *core.resource.extension* Extension call number
- *core.resource.password.hide* Extension password

Appropriate JavaScript constant definitions are available as **AcpResource.PROPERTY_XXX** or **AcpExtension.PROPERTY_XXX**.

Notes:

Changes do not affect configuration in CM.
The AJAX-Helper object replaces values of properties named "*.hide" by "******". Thus, the dialog is unable to display the old password.

4.3.1.2 Conference Bridge Settings

A single conference bridge can be configured for a 1XS user, but he is able to switch to another bridge via the conference bridge settings dialog (*/console/accountDialog.jsp*).

The account dialog is multifunctional, it is used in "BRIDGECONF" mode.

The following properties can be modified:

- *core.resource.displayname* A display name
- *conf.resource.bridgenumber* Call number of the bridge
- *conf.resource.secondary.bridgenumber* Call number of the secondary bridge
- *core.resource.participantcode* Participant Code for bridge access
- *core.resource.moderatorcode.hide* Moderator Code for bridge access
- *core.resource.pincode.hide* Bridge PIN Code (never used in the Dashboard)

Appropriate JavaScript constant definitions are available as **AcpResource.PROPERTY_XXX** or **AcpConferenceBridgeControl.PROPERTY_XXX**.

Notes:

Changes do not affect configuration in MX.
The AJAX-Helper object replaces values of properties named "*.hide" by "******". Thus, the dialog is unable to display the old moderator and pin code values.

4.3.1.3 Messaging Account Settings

Multiple messaging mailboxes can be configured for a 1XS user. The user is able to modify settings used by 1XS for mailbox access via the dialog */console/accountDialog.jsp*.

The account dialog is multifunctional, it is used in "MESSAGING" mode.

The following properties can be modified:

- *core.resource.displayname* A display name
- *core.resource.mailbox* Number of the mailbox (equal to user call number)
- *core.resource.password.hide* Mailbox password
- *vm.resource.wsourl* Url of the "Web Subscriber Options" page

Appropriate JavaScript constant definitions are available as **AcpResource.PROPERTY_XXX** or **AcpMailbox.PROPERTY_XXX**.

Notes:

Changes do not affect configuration in MM.
The AJAX-Helper object replaces values of properties named "*.hide" by "******". Thus, the dialog is unable to display the old moderator and pin code values.
Modular Messaging can be configured to offer a "Web Subscriber Options" web page which allows to manage a user's mailbox (not supported by the SDK's "single server lab" configuration).

4.3.1.4 Presence ACL Handling Settings

The AVAYA presence service enables a user to subscribe for reception of other users' presence information. If the requested level of presence information exceeds a minimum limit, the monitored user usually has to agree that his personal state may be forwarded to the listener.
These agreements are managed in the form of a user's **Access Control List** (**ACL**), a list of other users who act as listeners and the presence access levels granted to them.

The presence service can handle new subscriptions in several ways:

- Block the subscription automatically, inhibit reception of presence information for the subscriber.
- Automatically set the access level of the subscription, even if the subscriber has requested more.
- Send a notification event to the monitored user. The new subscription will appear as "pending" in his ACL. He then has to decide upon the level of access he will grant to the subscriber.

Presence handling may be configured via the "1XP Admin Client" on a System- or User Group-Profile level, but not per user (look for items "Default Access Type", "Default Access Level", "Minimum Access Level").
A typical client application, like the Dashboard, should offer the option of personal adjustment which overrides system/group profiles.

Presence handling configuration appears as properties of the "User" resource:

- *core.acl.defaultaccess.type* The default type of access handling
- *core.acl.defaultaccess.level* A default access level
- *core.acl.allow.ifrequested* The minimum access level

Appropriate JavaScript constant definitions are available as **AcpUser.PROPERTY_ACL_DEFAULT_XXX**.

The usage of default and minimum levels depends on the "access-type" property which may be one of

- *BLOCK* The user blocks distribution of his presence information
- *ALLOW* The user allows that others monitor him at the default level he has set
- *CONFIRM* The user wants to be notified of presence information requests and decide access-type on a per-user basis

For JavaScript constant definitions, see **AcpUser.ACL_ACTION_XXX**.

The default access level may stay undefined as long as the access type is not *ALLOW*.
As long as a user's access type is *CONFIRM* and another user is not included in his ACL list, the other user is able to receive the minimum level of presence information.

In the "User" resource's properties dialog below, you will recognize multiple "Allow" options.

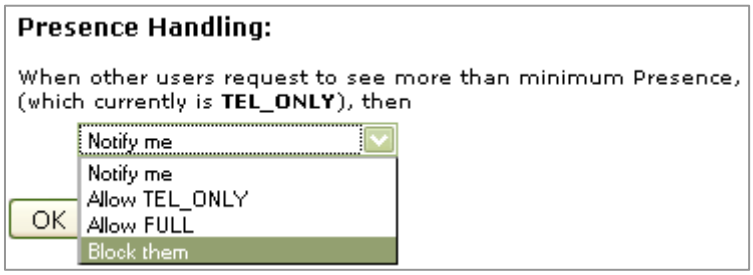

FIGURE 12: Presence Handling Dialog

You are able to "allow" any access level defined in the system. The currently available levels are

- *FULL* Full access to all presence information permitted
- *TEL_ONLY* Access restricted to phone state (which is the min. level per default)

Because the number of defined access levels is likely to be changed with future 1XS versions, the levels have to be determined during runtime (read from server).
In the Dashboard, possible access levels are determined by the "Presence" console sub-page (more precisely: by the "Support" object used there).

The dialog gets the levels from the console window object via *myConsole.getPresenceAccessLevels()*.
The console window script delegates this call to the function *external_GetPresenceAccessLevels()* provided by the presence sub-page.

4.3.2 Additional Functionality

The "General" tab of the console page presents some additional general purpose functionality which may help to visualize the application status.

Debug Window:

A means to enable/disable the "DebugPlus" monitoring feature at any time

AJAX Status:

Shows the current AJAX communication state.

AJAX communication errors and state changes are basically handled by the unique "Application" object. It has set a handler for these type of events in its internal AJAX-Helper object.
If you look into the class **AcpApplication** (*\js\acp_application.js*), you will find that this "Dashboard"-specific version of the class contains handlers which propagate error and state-change events to the console page window (*handleCommunicationError()* and *handleCommunicationStatusUpdate()* in *\console\console.js*).

The console page functions update the current state in the UI resp. open a popup box notification upon AJAX errors.

Client/Server Time Difference:

Many data items received from 1XS contain timestamp values (e.g. a call-log entry). All these timestamps (usually encoded in milliseconds since 1970-01-01 0:00) are based on the server timebase.

It may be annoying to view timestamp information on a client workstation with a reference timebase significantly different from that of the server (e.g. a call just made a minute ago might appear as an hour ago in the call logs).

To overcome this issue, server timestamp values have to be corrected by the time difference between server and client workstation. The console page determines the time difference via *myAcpApp.getClientServerTimeDifference()* after the AJAX communication was started. The delivered value has to be understood from a server perspective.

All Dashboard sub-pages which display timestamps, get the time difference from the console page via *myConsole.getClientServerTimeDifference()* and correct received timestamps before formatting them for display.

Timestamp correction cannot have acceptable results if the delivered timestamps are not based on the 1XS timebase (e.g. message timestamps are based on the MM server timebase).
Therefore, as a general rule, all AVAYA server products connected to 1XS should be time-synchronized.

Runtime Constants: (*\console\runtimeConstants.jsp*)

As mentioned before, some JavaScript constant values have to be determined at runtime of a client user session. The constants are collected in the script file *\js\waf\acp_runtime_constants.js*.
The support function "Show Runtime Constants" (button) of the console page allows to observe the actual constant values.

Component Versions: (*\console\runtimeConstants.jsp*)

Versions of the Dashboard application and of 1X client software components on the WebSphere A/S may be displayed in a separate window.

Application Settings: (*\console\applicationSettings.jsp*)

1XS provides a storage for arbitrary settings (key/value pairs) used in client applications. They can be managed via methods *getApplicationSettings()*, *setApplicationSettings()* of the User resource stub **AcpUser**.
A typical utilization of application settings is to store size and position of certain windows for restoration in the next user session.

The Dashboard doesn't make use of application settings but it supports displaying a list of current settings.

Application settings can be modified but not deleted. A non-existent setting is added upon a modification attempt.

Note that all available settings are established by the "1XP-Client", assuming that an additional application did not add some. **They should not be used or modified**.

You may recognize IP addresses as part of key names of settings. They are inserted to identify settings stored per client workstation.

Any client application which needs some application-specific settings should ensure uniqueness by selection of adequate key-names (e.g. the Dashboard would have taken names like "dashboard.xx.yy.zz").

4.3.3 Text Localization Demonstration

The Dashboard application does not provide a multilingual UI and the demonstration of server-side web page localization techniques is outside of its scope.

But, because of the AJAX and DHTML approaches used in 1XS client applications, text localization is often effected by JavaScript in the client browser.

The 1X Web Application Framework offers support for script-based localization.

The Dashboard demonstrates this support with a single localized greeting on the console pages "General" tab.

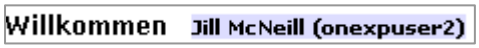

FIGURE 13: Localized Welcome Greeting

The "Welcome" greeting's language is adjusted according to the language setting of the client browser. For the example above, the browser's language list contained "German" at the top position.

How text localization is accomplished:

- The Dashboard web application contains text localization "resource bundle" files (*TextLocalization_<lang>.properties*) for supported languages at its Java classpath root position in *\WEB-INF\classes*.
- The console page *\console\console.jsp* determines the preferred browser language (Locale) from the browser request and uses the WAF framework class *com.avaya.acp.client.AcpWebClientUtils* to include (in the page) initialization code for an associative JavaScript array *localizedStrings* by execution of *AcpWebClientUtils.buildArrayContentFromResourceBundle(...,locale)*.
  Key/localized-text pairs in the array are taken from the resource bundle file which matches the determined language.
- The associative array is stored in the unique application object by *myAcpApp.setLocalizedStrings(localizedStrings)* during script initialization of the console page (*body_OnLoad()*).
- Localized text for a known key may be obtained by calling *myAcpApp.getString(<key>)*.

4.4 Resource View Pages

The Dashboard presents each resource on a single web page which is a sub-page of the console. The console page arranges its sub-pages in the form of tabs.

The pages and associated scripts reside in folders *\pages\<resource>* of the web application.

Although page content is thematically very different, page architecture and software organization is similar. You should be aware of the basic page structure in order to more readily locate developer resources and develop your own applications:

- Pages generally have to include the script library *\js\waf\acp_client.js*.
- Scripts for each page are initialized by a function ***body_OnLoad()***.
- Each page needs references to the parent console page and the "Application" object hosted by the console page. The local references are named ***myConsolePage*** and ***myAcpApp***, they are determined by

  ```
  myConsole = getConsole_Util();
  myAcpApp = myConsole.getApplicationObject();
  ```

  (you find "getConsole_Util()" in *\js\dashboard.js* as a single exception of the rule that utility functions are inside *\js\reusable\utils.js*) .
- A page implements a function *closeOpenSubWindows()*. It is called by the console page's window upon user logout and instructs a sub-page to close any open additional windows (e.g. a dialog).
- A page may offer services to other pages (e.g. the telephony page offers a function which initiates an outbound call). Exposed service functions have names like ***external***_*<function-xxx>()*.
- Most of a page's scripts handle UI events triggered by the user (***button..._OnClick()***), enable, disable or prepare UI elements (***enable...()***, ***prepare...()***) or handle table content (***table...()***). The naming pattern helps to navigate.
- A page usually maintains a resource "stub" object (*xxx****ResourceStub***) which serves to send requests to the remote resource adapter and a suitable "support" object (*xxx****Support***) which handles received AJAX events.
- Page scripts comprise the functions ***handleAjaxResourceResonse(response)*** and ***uiHandle...Notification(uiNotification)***, they handle responses sent by the 1XS API resp. notifications emitted by the support object upon events.
- In order to avoid multiple triggers for the same operation, pages disable UI elements until an operation is completed. It is a good practice to do so, but there is a risk: If serious communication failures occur during the phase of UI disablement (e.g. no response arrives after a resource request), the UI remains frozen. Dashboard pages which include operations during which essential UI elements are disabled use timers to detect communications failures. If the response or event which should re-enable the UI does not arrive, the UI is enabled by force after the timeout period expires(e.g. after 10 sec).

It would be beyond the scope of this document to describe all the details in the Dashboard's pages' JavaScript code. Instead, only the most important code modules will be highlighted. Comments in the code provide additional explanations. The developer is encouraged to view the source code for the pages to gain insight into the details.

4.4.1 Telephony

The following picture shows the Dashboard's telephony page *\pages\telephony\telephony.jsp*. The user has logged in for control of his extension, byt he hasn't set any specific modes. There are two currently active calls.

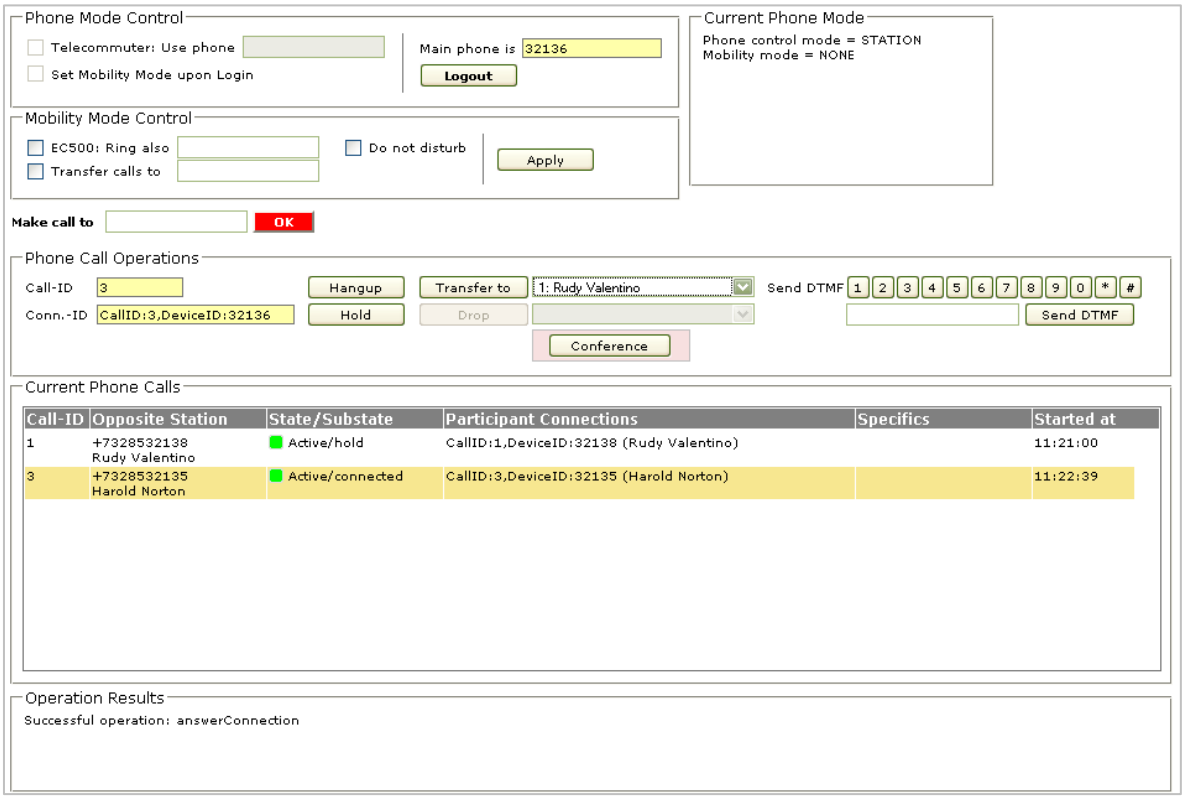

FIGURE 14: Telephony Page

The telephony pages main script file is *\pages\telephony\telephony.js*.

4.4.1.1 Basics

The Dashboard handles creation of the Extension resource in a special way.
It is excluded from the resource creation process of the console page. Instead, the resource may be created/destroyed dynamically by the user. This is called "login"/"logout" on the telephony page.

The Extension resource of a user has to be configured in 1XS (via "1XP Admin Client").
Configuration settings are available as "properties" of the resource:

- *core.resource.extension*
  The extension's call number (displayed as "Main phone is" in the Dashboard)
- *core.resource.displayname*
  A display name set for the extension (not shown by the Dashboard)

See **AcpExtension.PROPERTY_XXX** or **AcpResource.PROPERTY_XXX** for JavaScript constants.

Different modes may be applied to the user's extension. There a 2 types of modes.

Control Modes:

- *STATION* 1XS controls the user's preconfigured main extension (usually his office phone).
- *TELECOMMUTER* 1XS controls the user's extension which is associated to another call number (freely adjustable) than the preconfigured one.
  The main phone is out of order.
- *NONE* 1XS does not control the user's phone. This mode is not supported by the Dashboard (same as not being logged in to the phone).
- *VOIP* The 1XS client web application represents a softphone.
  This mode is not supported by the Dashboard.
  Prerequisites for this mode: A VOIP (H.323 or SIP) softphone component which can be controlled via JavaScript during a browser session (e.g. a browser plugin).

The control mode should be set directly upon login to the phone (after resource creation).

Mobility Modes:

- *NONE* No mobility mode set
- *CALL_FORWARD* Incoming calls are transferred to another station (call number freely adjustable)
- *DO_NOT_DISTURB* Incoming calls are sent to coverage
- *EC500* AVAYA'S "Extension to Cellular" technology used. Upon incoming calls, another phone (e.g. a mobile phone) rings in parallel to the main phone and may be used to answer the call.
  Note: EC500 is not supported by the default configuration of the "Single Server Lab" provided by AVAYA as the development environment for 1X-based web applications.

JavaScript extension mode constants are: **AcpExtension.CONTROL_MODE_XXX** resp. **AcpExtension.MOBILITY_MODE_XXX**.

There are some settings in 1XS **System-** and user **Group Profiles** which affect available modes presented by the 1XP-Client (use "1XP Admin Client" to see and adjust them).

Though not mandatory, other client applications should respect the profile settings.

The Dashboard respects them and takes the properties to inhibit the corresponding mode if the property value is "false".

Property Name (Admin-Client)	Property Key (API)	Meaning
Telecommuter	*tel.resource.feature.telecommuter*	Permission to enable Telecommuter control mode
VOIP	*tel.resource.feature.voip*	Permission to enable VOIP control mode
Mobility	*tel.resource.feature.mobility*	Permission to enable EC500 mobility mode

JavaScript mode feature property constants are: **AcpExtension.PROPERTY_FEATURE_XXX**.

Besides the "Application" object reference *myAcpApp* obtained from the console window, the telephony page utilizes two additional objects of central importance:

- ***myExtensionResourceStub***
  A "stub" object (class **AcpExtension**) for calling methods of the remote extension resource adapter. The instance is obtained via
  myExtensionResourceStub =
  myAcpApp.getUser().getResourceByType(AcpResource.TYPE_TELEPHONY);
  (applicable after creation of the resource).
- ***telephonySupport***
  A "support" object (class **TelephonySupport**) for event handling and call data storage.

4.4.1.2 Login to / Logout from Phone

The resource creation ("login") is combined with the adjustment of the extension's control mode. Two options are available in the UI: Set the *STATION* mode (if the "Telecommuter" checkbox is not selected) or the *TELECOMMUTER* mode.

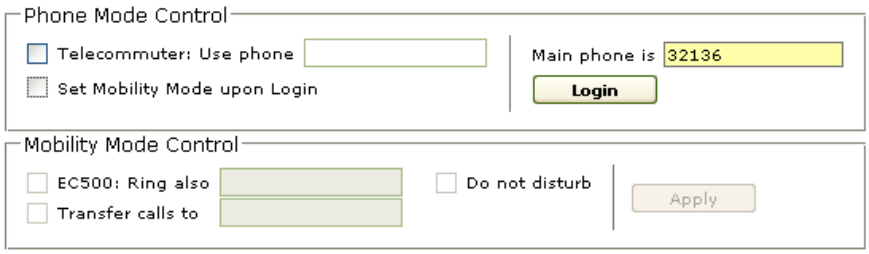

FIGURE 15: Phone Mode Control

Mobility modes can be applied at any time after the user has logged in to the extension, but also as part of the login process directly (checkbox "Set mobility mode upon Login").

Note: The mobility mode stays valid after logout from the extension. Keep that in mind when you login again without setting the mobility mode.

The extension resource is created/removed via the "User" resource stub by

myAcpApp.getUser().createResources(..., this);
resp.
myAcpApp.getUser().removeResource(..., this);

During resource creation/removal, the telephony page acts as a handler which listens for completion of the operations and therefore has to provide handler methods *handleAcpUserResourcesCreated(...)* resp. *handleAcpUserResourceRemoved(...)*.

After creating the resource, adjustment of the extension's control mode and (optionally) of the mobility mode follows immediately. A **Transaction** is started (*myAcpApp.startTransaction(<unique-name>)*) to speed up the possibly two sequential operations (remember that a transaction causes a time-limited increase in AJAX polling frequency).

Control and mobility modes are set by *myExtensionResourceStub.setControlMode(..., this)* resp. *myExtensionResourceStub.setMobilityMode(..., this)*. As usual, the call of the response handler function *handleAjaxResourceResponse(...)* serves as an indicator that an operation is successfully completed. If both types of mode have to be set during the login phase, the operations are chained by their resource responses (*doNextPhoneLoginOperation()*).

The transaction is stopped upon receiving the resource response of the last operation.

After successful login, the "Login" button changes its meaning and works as "Logout" button.

If one of the mode setting commands causes a mode change, 1XS fires the event ***CommunicationResourceAttributeChanged***. The event will be received by the support object which extracts the mode setting for storage in its internal **PhoneMode** object and notifies the UI scripts.

Because of the resource creation/removal feature, the resource stub (*myExtensionResourceStub*) and support (*telephonySupport*) objects have to be handled in a specific way:

- Every time the Extension resource is created, a new resource stub object has to be created and the support object has to be initialized (*telephonySupport.initialize(<stub-obj>)*).
- If the resource is removed, the support object has to be instructed to release its event listeners (*telephonySupport.stopListening()*) and to clear its internal call- and mode-status data (*.removeCalls()*, *.clearPhoneMode()*).

4.4.1.3 Call Handling

Call handling is event driven. The user may operate an UI element to send a resource request (e.g. to make a call), the support object receives events, translates them into objects and stores data of current calls.

Events handled by the support object are

- ***CommunicationStarted*** A new call starts
- ***CommunicationEnded*** A call ends
- ***CommunicationParticipantEnter*** A participant enters a call
- ***CommunicationParticipantExit*** A participant exits the call
- ***CommunicationParticipantAttributesChanged*** The state of a participant changes

A call is represented by a **PhoneCall** object, call participants by a list of **PhoneConnection** objects stored in the call object.
Each time a call is added, modified or removed, the support object sends a notification to the page's scripts by calling the handler function *uiHandleTelephonyNotification(uiNotification)*.
The function parameter *uiNotification* is an instance of **TelephonyUINotification**, it contains a notification type code (**TelephonyUINotification.TYPE_XXX**) which reflects the event and the added, modified or removed **PhoneCall** object.

Upon receiving notifications, the UI's table of current calls will be updated (functions *tablePhoneCalls_XXX()* used). A call is a row in the table.
Each call object has a unique call-id value which serves as identification of the table row (row-id).
Several connections may belong to a call (at least two for a normal call, more than two for a conference). The participant connections are displayed as a list in a column of the table row.

If the user wants to control a call, he has to select it in the table (click on the table row) and possible operations are offered in the UI's "Phone Call Operations" section.
The page automatically selects a call if there is only one call (new or remaining) in the list.
Having the table row-id (which is the call-id), the call object can be obtained from the support object by *telephonySupport.getCall(callId)*.
Possible operations for the call depend on attributes of the call object:

- the call direction (incoming/outgoing)
- the current status (state/sub-state)
- some special call criteria (is it a conference- or a "bridged"- call)
- the current number of other calls (e.g. to join them to a conference)

FIGURE 16: Phone Call Operations

Call operation buttons not only will be enabled/disabled, some of them also change their meaning.
The central point of decision on what can be done with a call is the function *preparePhoneOperationsForCall(callId)*.

Some specifics:

- In the "Single Server Lab" which may be provided by AVAYA as the SDK development and test environment, processing of DTMF requires an additional **Media-Gateway** connected to the server. A media gateway is not included in the AVAYA-provided Single Server Lab (but it can be added).
- Conferences are conferences supported by CM (vs. conferences supported by MX conference bridges).
  If calls are joined to a CM conference, the conference appears as a single call with three or more participant connections. If there are less than three participants, the conference falls back to a normal call.
- A special type of call may appear in case of "bridged call-appearances" (a call-appearance of the user's extension is "bridged" to one of another extension - this must be configured on CM). If another extension (to which the user's extension is bridged) is called and answers, 1XS may signal the call as "**bridged**". This type of call cannot be "answered" or "ignored" like a normal incoming call (since it already exists in an active, connected state), the answer operation becomes the meaning of "join the call". If the user joins the call (API command: *answerConnection()*), the call becomes a conference.

4.4.2 Call-Logs

The Dashboard's contact-log resource view is represented by the page *\pages\contactlog\contactlog.jsp* and the corresponding main script file *\pages\contactlog\contactlog.js*.

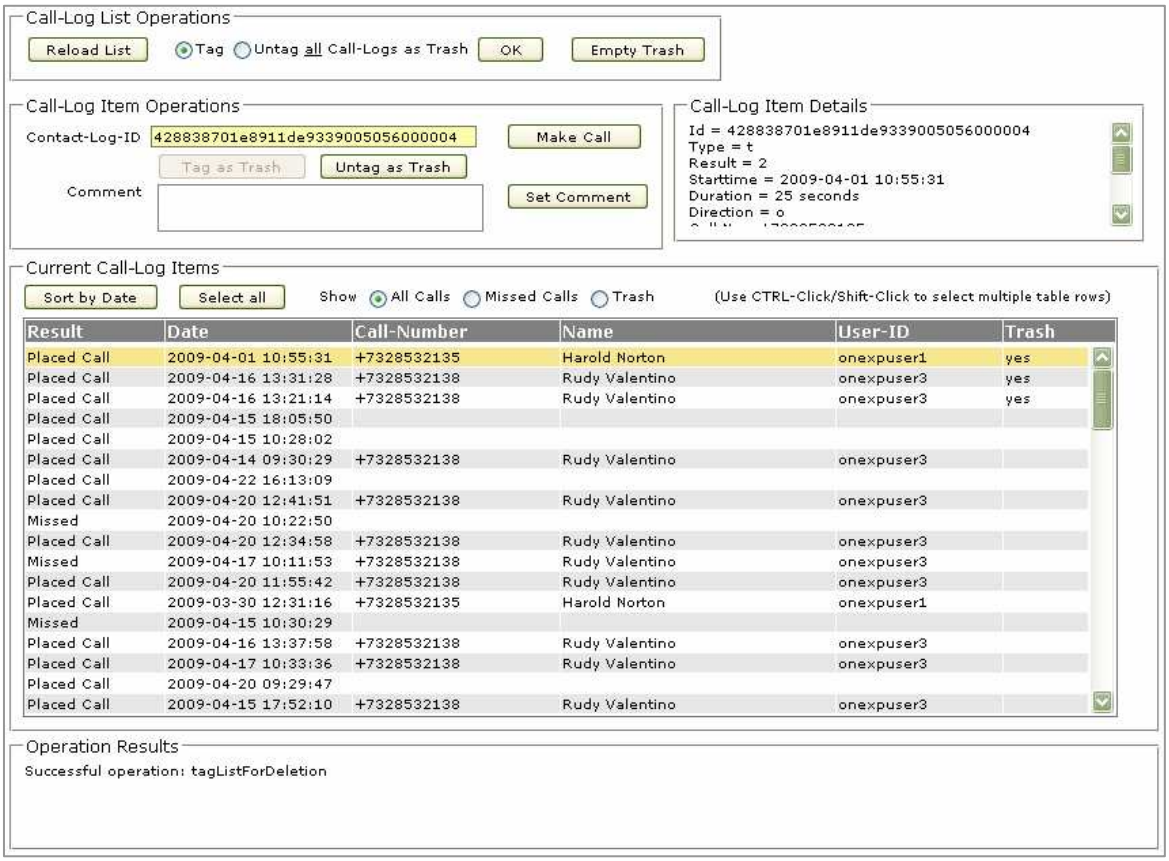

FIGURE 17: Call-Logs Page

4.4.2.1 Basics

Despite its more generic approach, the contact-log resource currently supports telephony call-logs only.

The most important script variables of the page are

- ***myAcpApp***
  The "Application" object reference obtained from the console window
- ***myContactLogResourceStub***
  A "stub" object (class **AcpContactLogger**) for call of methods of the remote contact-log resource adapter. The instance is obtained via
  myContactLogResourceStub =
  myAcpApp.getUser().getResourceByType(AcpResource.TYPE_CONTACT_LOG);
  (applicable after creation of the resource).
- ***contactLogSupport***
  A "support" object (class **ContactLogSupport**) for event handling and call-log data storage.

A call-log item is added after each call (successful or not) on the user's extension.

What can be done with a call-log item:

- It can be tagged for deletion. Tagged call-logs are deleted automatically if the user logs out from 1XS.
  Furthermore, 1XS periodically purges old call-logs (period adjustable via "1XP Admin Client").
- A comment can be stored for a call-log.

Notes:

- Names are resolved via the call number. Thus, it is necessary to provide a valid call number for user data stored in ADS.
- There are two call-log settings adjustable via the "1XP Admin Client" (System profile): The maximum number of stored history records and the number of days to store history records. The settings are available as properties of the contact-log resource: *history.maxrecords* and *history.maxdays*.

4.4.2.2 Call-Log Handling

Events received by the support object are

- ***CommunicationHistoryList*** One of the current call-logs is sent
- ***CommunicationHistoryAdded*** A new call-log item is added
- ***CommunicationHistoryRemoved*** A call-log item is removed
- ***CommunicationAttributeChanged*** A attribute of a call-log is modified

The support object offers the option to load all current call-logs upon initialization or to reload the list at any time (*contactLogSupport.reloadList()*).
The usual handling is to load the list once upon startup of the application. Additional logs are then added after completion of new calls.

A log item is represented by a **ContactLog** object.

Each time a call-log is added, modified or removed, the support object sends a notification to the page's scripts by calling the handler function *uiHandleContactLogNotification(uiNotification)*.
The function parameter *uiNotification* is an instance of **ContactLogUINotification**, it contains a notification type code (**ContactLogUINotification.TYPE_XXX)** which reflects the event and the added, modified, or removed **ContactLog** object.

The notification **TYPE_CONTACTLOGS_RELOADED** appears during reload of the call-logs list. It is important to know that this notification is sent upon reception of the resource response to the API call *<contact-log-resource-stub>.getContactLogs()*.
Call-log list items are sent in form of *CommunicationHistoryList* events (notification **TYPE_CONTACTLOG_LIST_ADD**).
As mentioned before, **there is no guarantee that the response arrives after the last list event**. Therefore not much can be done with this notification.

Call-logs sent by 1XS **currently have no specific sort order**, sorting has to be done by the client application. The support object offers a method which sorts the list of call-logs by their creation time (*contactLogSupport.sortContactLogsByTime()*).
Because of the lack of signal that loading of the list is completed, it is an issue when to apply the sorting. The Dashboard delegates the decision to the user (it offers a "sort" button), other applications may perhaps sort after a timeout.

Upon notifications, the UI's table of current call-logs will be updated (functions *tableContactLogs_XXX()* used). A call-log is a row in the table.
Each log item has a unique contact-log-id value which serves as identification of the table row (row-id).

Most important attributes of a contact-log are

- type Currently all contact-logs have type "Telephony", thus they are call-logs
- direction Contact direction: "incoming" or "outgoing"
- result The result of the contact event, e.g. a call was successfully "placed" or "missed"

JavaScript constants for possible attribute values are defined as members of the support class: **ContactLogSupport.DIRECTION_XXX**, **ContactLogSupport_RESULT_XXX**.

The call-log page is able to apply a filter to log items and display them "all" or those only which are "missed" or "trash" (= tagged for deletion). The filter function is not part of the support object (which holds all current call-logs in its internal list).

The user may select one or more call-logs in the UI's list, possible operations are offered in the "Call-Log Item Operations" section.
The page automatically selects a call-log if there is only a single item (new or remaining) in the list.
Having got the table row-id (which is the contact-log-id), the call-log object can be obtained from the support object by *contactLogSupport.getContactLog(contactLogId)*.
Possible operations depend on (see function *prepareContactLogOperations()*):

- the number of selected call-logs
- the "tagged for deletion" state

The call-log page is a demonstration of some interaction with another sub-page of the Dashboard's console. It determines a reference to the telephony sub-page window via the console page and checks if the user has logged-in to the extension (the telephony sub-page offers the function *external_IsPhoneLoggedIn()* to test this). Once the telephony resource is available, a "make call" button is offered for a selected call-log. If the button is clicked, the call-log page uses function *external_MakeCall(<call-nr>)* of the telephony page to establish a call.

There are other possibilities of cooperation between the call-log- and other resources (not demonstrated by the Dashboard):

- If a valid user-id is available for a call-log, addresses of the user may be read from the contact-list resource. Having obtained the addresses, e.g. a mail client may be started to send a mail.
- The call-log page may register for reception of presence information for each identified 1XS user and then display the presence state.
  Be careful: if there are hundreds of call-logs, sending of presence events may cause a significant load on the server.

4.4.3 Contacts

The Dashboard's contact-list resource view is represented by the page *\pages\contaclist\contactlis.jsp* and the corresponding main script file *\pages\contactlist\contactlist.js*.

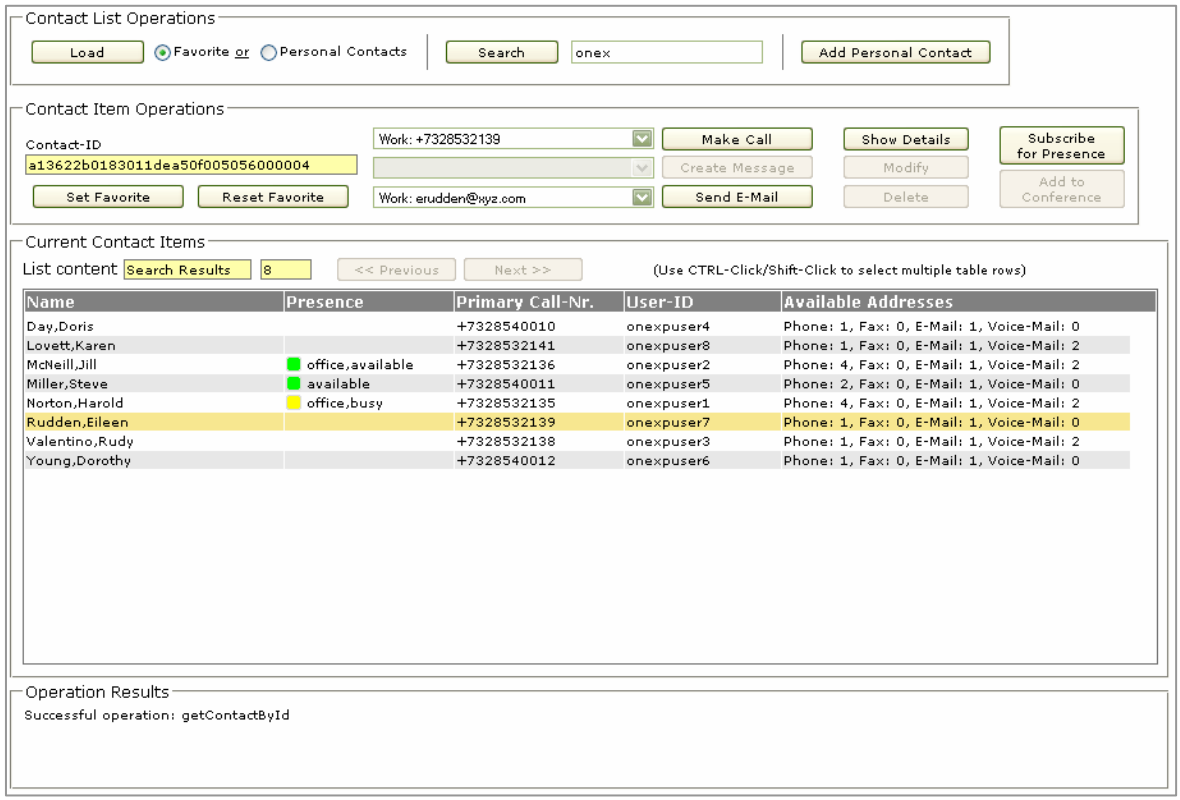

FIGURE 18: Contacts Page

The picture above shows 1XS users preconfigured for the "Single Server Lab" which may be provided by AVAYA as SDK test environment.

4.4.3.1 Basics

Important script variables of the page are

- ***myAcpApp***
  The "Application" object reference obtained from the console window
- ***myContactListResourceStub***
  A "stub" object (class **AcpContactList**) for call of methods of the remote contact-list resource adapter. The instance is obtained via
  myContactListResourceStub =
  myAcpApp.getUser().getResourceByType(AcpResource.TYPE_CONTACT);
  (applicable after creation of the resource).
- ***contactListSupport***
  A "support" object (class **ContactListSupport**) for event handling and contact data storage.

Contacts belong to one of the **Scopes**:

- **Enterprise** contacts scope
  Comprises users configured in ADS. Enterprise contacts cannot be modified, the only available operation is to set a flag which marks the contact as a "favorite".
- **Personal** contacts scope
  Data of personal contacts have to be provided by the user for import into the 1XS database. A personal contact record may be modified by its owner.

Since thousands of contacts may be available in the ADS of a company, contact search operations may have a large number of results. Furthermore, the set of data stored for a contact may alsobe large.

To avoid server performance issues, contacts are read in a "paged" manner. After search requests, only a subset of matching contacts is delivered, the maximum length of the subset is selected in the search command. An application which presents contacts has to provide a mechanism for display of the "next" or a "previous" available block of contacts.

The delivered contact's set of attributes is incomplete. To get all data of a contact, a single selected contact has to be read again from the server.

4.4.3.2 Contact Handling

Events received by the support object are

- ***ChunkedSqlResult*** Delivers a single contact as result of a search operation. Only reduced set of data included.
- ***ContactsList*** Delivers a single contact as result of a load operation. Only reduced set of data included.
- ***ContactFetched*** Delivers a contact as result of a single contact read operation. All available contact data are included.
- ***ContactRemoved*** A contact was removed (can apply to personal contacts only)
- ***ContactAttributeChanged*** An attribute of a contact has changed

The internal list of the support object contains contacts resulting from a search or load operation.

A contact is represented by a **Contact** object. Contact objects contain a list of **ContactAddress** objects, the available addresses. Attributes of an address are:

- The address string
- An address type (e.g. a phone number, an email address)
- An address location identifier (e.g. "office", "home", "mobile")

JavaScript constants for possible address types and locations are defined as **Contact.ADR_TYPE_XXX** resp. **Contact.ADRLOCATION_XXX**. Some of the defined address types are currently not supported by 1XS, as the Dashboard focuses on phone numbers, and email and voicemail addresses.

Contacts in the internal list of the support object are partially populated objects only, they do not contain the addresses. Instead, they contain a list of **ContactAddressCount** objects which represent the count of available addresses per address-type.

Each time a contact is added to, modified in, or removed from the internal list of the support object, it sends a notification to the page's scripts by calling the handler function *uiHandleContactListNotification(uiNotification)*.
The function parameter *uiNotification* is an instance of **ContactListUINotification**, it contains a notification type code (**ContactListUINotification.TYPE_XXX)** which reflects the event and the added, modified, or removed **Contact** object.

The Dashboard contact-list page allows searching for contacts, loading those contacts which are marked as "favorite", and loading personal contacts. Searching by name or phone number and loading of favorites cover both scopes, "Enterprise" and "Personal".

Only indexed operations are applied to load contacts (see function *sendGetContactsRequest_Paged()*):

- *myContactListResourceStub.getFavoritesByIndex()*
- *myContactListResourceStub.getContactsByNameAndIndex*
- *myContactListResourceStub.getContactsByNumberAndIndex*

The command used for searches depends on the given address fragment. If it can be identified as a call number, the search method is *...ByNumberAndIndex*.

Besides search criteria, the parameters "maximum number of records" and "index" have to be set in the request for indexed operations. The "index" is a start index in the set of matching records on the server.

Each contact-list event (and therefore the contact object stored by the support object) contains the start index, the number of contacts in the currently requested block, and the total number of contacts which match the search criteria.
An application is able to implement "paged" contact loading based on these values. The Dashboard requests a maximum number of 30 contacts per block (see function *preparePagedLoadOperations()*).

Upon notifications, the UI's table of current contacts is updated (functions *tableContacts_XXX()* are used). A contact is presented as a row in the table. Each contact has a unique contact-id value which serves as identification of the table row (row-id).
Having obtained the table row-id, a contact can be retrieved from the support object by *contactListSupport.getContact(contactId)*.

The user may select one or more contacts in the UI's list, possible operations are offered in the "Contact Item Operations" section.
The page automatically selects a contact if there is only a single one (new or remaining) in the list.

As mentioned before, listed contacts are partially populated objects which are insufficient for a complete decision on applicable operations. For such a decision, a single, fully populated contact must be available.
In order to obtain the fully populated contact, the contact-list page loads a selected (only if a single item is selected in the list) contact again by *myContactListResourceStub.getContactById(contactId)*.
The request results in a *ContactFetched* event (UI notification **TYPE_CONTACT_FETCHED**).
Based on the received contact object, possible contact operations will be enabled (see function *prepareContactOperations_SSC()*).

Some of the operations available for a contact require support by another sub-page of the Dashboard's console:

- For one of the telephony addresses of a contact, a call may be initiated via the telephony sub-page.
- For one of the voicemail addresses of a contact, creation of a new message may be started via the messaging sub-page.
- If a "primary phone number" is available for a contact, the contact may be called to enter a conference on a MX bridge via the bridge-conferencing sub-page.
- If a contact is in enterprise scope and has a valid 1XS user-id, a registration for reception of presence information may be initiated via the presence sub-page.
  The contact-list page provides the function *external_NotifyUserPresenceInfo()*. The presence sub-page calls this function if updates of presence information are available.
  Thus, the contact-list page is able to display presence information for contacts presented in its UI list.

Note:

Current events do not contain reliable information on a contact's "favorite" status.
Therefore "Set Favorite" as well as "Reset Favorite" operations are offered for contacts in the UI (exception if favorites only are loaded). The only way to check a contacts "favorite" status is to load favorites only.

4.4.3.3 Personal Contacts

The page *\pages\contactlist\contactDialog.jsp* and its associated scripts serve as dialog for entering a personal contact.

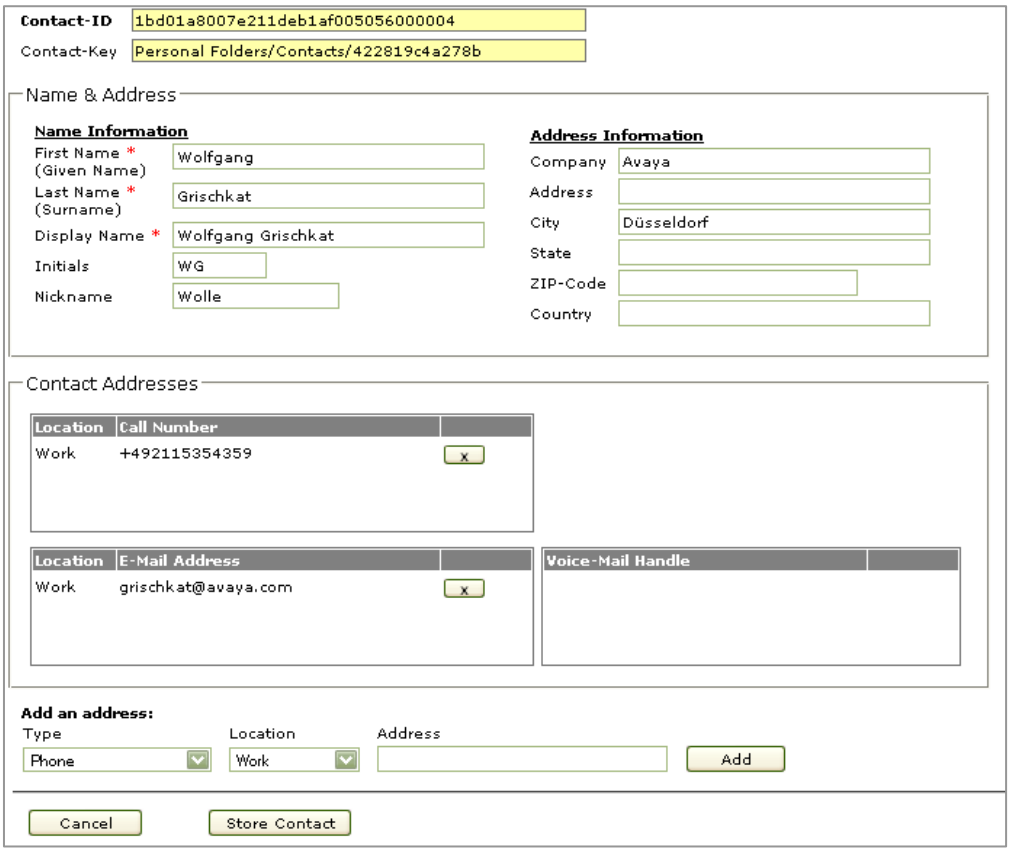

FIGURE 19: Personal Contact Dialog

Upon new definition of a personal contact, a "contact-key" has to be set. The Dashboard generates a random value as key.

The unique contact-id value is generated by 1XS. To see its value, you have to import the contact into the 1XS database and read personal contacts again.

Personal contacts are stored in the 1XS database using the import method *myContactListResourceStub.importContact(contact)*. You find this request if you follow the script function *dialog_StoreContact(contact)* of the contact-list page (in *contactlist.js*).

Notes:

- A personal contact can be modified by importing it again.
- **1XS does not send events if a personal contact is added, modified or removed**.
  In order to see changes, personal contacts have to be loaded again (or a matching search has to be performed).

As a workaround, the Dashboard directly manipulates the support object's internal list upon reception of the resource response to the modification request for an existing personal contact.

4.4.4 Bridge Conferencing

Conferences on a Meeting Exchange (MX) bridge have to be clearly distinguished from conferences supported by CM.

The following picture shows the Dashboard's bridge conference control page *\pages\bridgeconf\bridgeconf.jsp*. Main script of the page is *\pages\bridgeconf\bridgeconf.js*.

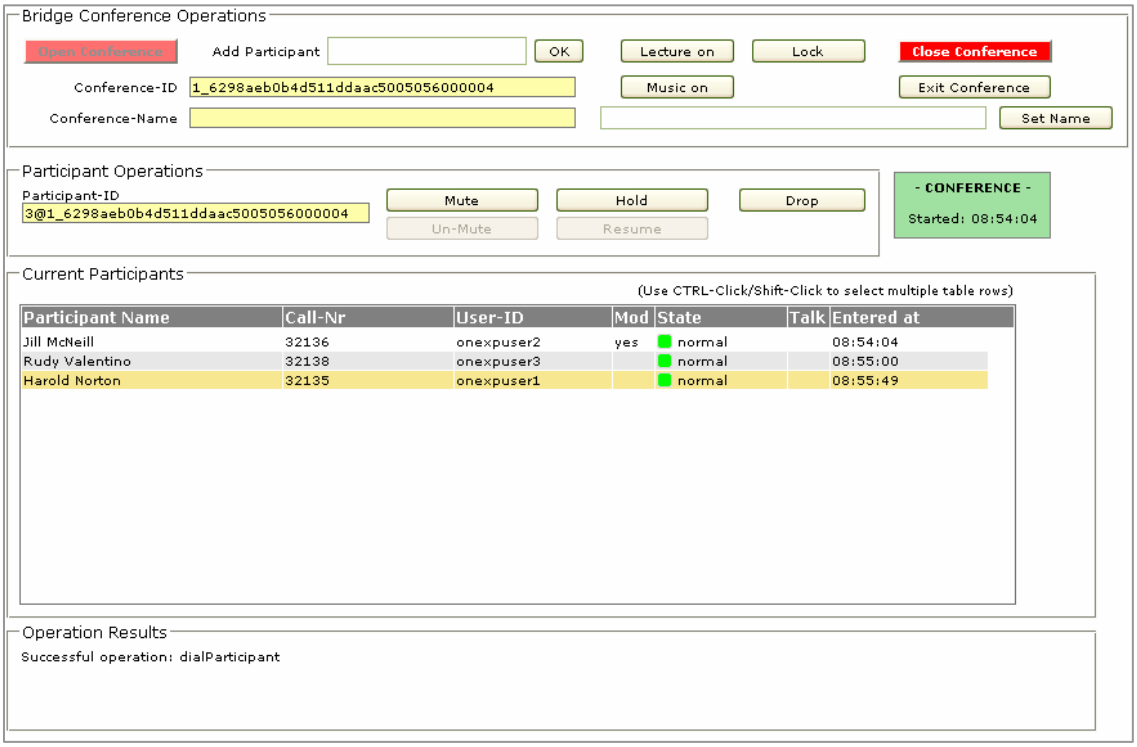

FIGURE 20: Bridge Conferencing Page

4.4.4.1 Basics

It was already mentioned in Chapter 4.3.1, that it is not mandatory to configure bridge conference resource settings for a user (via "1XP Admin Client") in order to get some basic support by the 1XS API.
If a user simply wants to control himself (mute, exit) and see the other participants in a conference he attends, there has to be no specific user configuration at all.

Nevertheless, the Dashboard assumes that at least the conferencing server "handle" has to be set for the user, otherwise the resource will not be created and the bridge-conferencing sub-page remains un-available.

Configuration is mandatory if the user wants to open or enter a conference as a moderator. The bridge's name ("handle") or call number and the moderator code ("host code") have to be known in this case.

A moderator may open a conference in two ways:

- He can call the bridge using his phone by dialing the bridge's call number, afterwards he has to dial the moderator code using DTMF.
  Note that DTMF processing requires a Media Gateway connected to CM.
  Such a Gateway is not available in the "Single Server Lab" optionally provided by AVAYA as test environment (but it can be added).
- He can instruct (via the 1XS API) the bridge to call his extension ("**Call Me**") and then answer.

Similarly, there are two ways for a participant to enter a conference:

- He can call the bridge using his phone by dialing the bridge's call number, afterwards he has to dial the participant code using DTMF (Media Gateway needed).
- A moderator can instruct (via the 1XS API) the bridge to call the participant's extension. The participant has to answer.

The Dashboard focuses on use of API calls. Direct calling of the bridge is implicitly available if the test environment is capable of processing DTMF signals.

User specific conference settings are reflected by properties of the resource:

- *core.resource.server.name* Conferencing server name ("handle")
- *conf.resource.bridgenumber* Call number of the bridge
- *core.resource.participantcode* Participant Code
- *core.resource.moderatorcode.hide* Moderator Code ("host code")
- *conf.resource.allow.call.me* Permission to use "Call Me"

Any client application should respect the configurable permission to use "Call Me". The Dashboard disables this approach to opening a conference if the value of the property is "false".

The most important script variables of the bridge-conferencing page are

- ***myAcpApp***
  The "Application" object reference obtained from the console window
- ***myConferenceResourceStub***
  A "stub" object (class **AcpConferenceBridgeControl**) for calling methods of the remote conference-bridge resource adapter. The instance is obtained via
  myConferenceResourceStub = myAcpApp.getUser().getResourceByType(AcpResource.TYPE_CONFERENCE_BRIDGE_CONTROL);
  (applicable after creation of the resource).
- ***conferenceSupport***
  A "support" object (class **ConferenceBridgeSupport**) for event handling and conference data storage.

4.4.4.2 Conference Handling

Because of the trailing "*.hide*" in the key, framework JavaScript libraries replace the moderator code by "****". Thus, the code is not available in resource properties, it has to be determined by reading it from the server (via method *getResourceSetting(<resource-id>,<property-key>)* of the user resource stub).

The support object may optionally fetch the code during its initialization.

Furthermore, the support object collects property settings important for conference control and offers methods for check of the settings (*isConferenceConfigured()*, *isCallMeAllowed()*).

The functionality "Open Conference" is enabled on the bridge-conferencing page only if the conferencing resource is sufficiently configured for the user and if "Call Me" is allowed.

A conference is opened ("Call Me") by resource request
myConferenceResourceStub.joinConference(<bridge-handle>, <user-call-nr>, <mod-code>,...).

The call-number of the user's extension is represented by the property *core.resource.extension* of the extension resource (the support object determines the value and has stored it as internal attribute).

A way to add a participant to the conference is to instruct the bridge to call him. This is done by the resource request
myConferenceResourceStub.dialParticipant(<conference-id>, <call-nr>, ...).

Events received by the support object are

- ***CommunicationStarted*** A conference was opened
- ***CommunicationEnded*** A conference was closed
- ***CommunicationAttributeChanged*** An attribute of the conference has changed
- ***CommunicationParticipantEnter*** A participant enters the conference
- ***CommunicationParticipantExit*** A participant is removed from the conference
- ***CommunicationParticipantAttributesChanged*** An attribute of a participant has changed

The support object contains a single conference object (an instance of **BridgeConference**).
As long as there is no open conference for the user, this object is undefined.
Besides various state attributes, the conference object holds a list of participants (instances of **ConferenceParticipant**).
A conference, as well as each participant of it, is identifiable by a unique ID value.

If a conference is started resp. ended or its attributes or participants change, the support object sends a notification to the page's scripts by calling the handler function *uiHandleConferenceBridgeNotification(uiNotification)*.
The function parameter *uiNotification* is an instance of **ConferenceBridgeUINotification**, it contains a notification type code (**ConferenceBridgeUINotification.TYPE_XXX)** which reflects the event and an additional object which may be the conference object or an added, modified or removed participant object.
A special notification (**TYPE_MODERATORCODE_READ**) is sent after optional reading of the moderator code from 1XS.
Only a single conference can exist for the user at a point in time. The conference object may be obtained from the support object by calling *conferenceSupport.getConference()*.

Upon notifications, either the "Conference Operations" UI-section is adjusted (see function *prepareConferenceOperations()*) or the UI's table of current participants is updated (functions *tableParticipants_XXX()* used).
A participant is presented as a row in the table, the row-id is equal to the unique participant-id. Having obtained the table row-id, a participant can be retrieved from the support object by *conferenceSupport.getParticipant(participantId)*.
The user may select one or more participants in the UI's list, possible operations are offered in the "Participant Operations" section (see function *prepareParticipantOperations(...)*).
The page automatically selects a participant if there is only a single one (new or remaining) in the list.

Most of the conference or participant operations can be performed by a moderator only, but there are operations which a non-moderator participant may apply to himself.
The support object offers some methods for check of cases:

- *isLoggedInUserModerator()*
- *isParticipantLoggedInUser(<participant-id>)*
- *isParticipantSuitableForDrop(<participant>)*

Notes:

- If a participant (not moderator) is logged in to 1XS and a moderator calls him (via the bridge) to enter an existing conference, then the complete conference becomes visible for the participant but his control options are limited: exit from conference, mute himself, set himself on hold.

- If a moderator opens a conference which already exists (already opened by another moderator), he will join the conference. A conference will be closed if one of the moderators calls the *dropConference()* API command or if the last moderator exits the conference using the command *dropSelfParticipant()*.

4.4.5 Presence

Handling of the Presence resource may be logically divided into 3 independent areas:

- Publishing of one's own presence information
- Subscriptions for reception of presence information of other users
- Control of access to one's own presence information

The Dashboard's presence page *\pages\presence\presence.jsp* organizes the functional areas as tabs. Only one of the tab "windows" is visible at a time, selectable by the user.

FIGURE 21: Presence Page Tabs

Technically speaking, the tab contents are not "windows", they are HTML * * elements. Thus, the presence page is a single window.

The main script of the page is *\pages\presence\presence.js*.

4.4.5.1 Basics

Presence is a service for distribution of a user's **Presence Information** to other interested and authorized users.

Presence information is a collection of state data of the user (personal state) and of associated **Devices** (e.g. his phone or a software application started by him) gathered by the presence server.

A user may either **Publish** manually selected presence state information via the service or he may have the service automatically determine his state (e.g. set status to "busy" if he is engaged in a phone call).

Any user who wants to monitor the presence state of another user must **Subcribe** to receive this information. A subscription contains a **Level** which defines the amount resp. depth of the requested information.

Reception of a minimum level of presence information usually does not require any permission.
A monitored user has to agree upon distribution of a higher level of information.
This can be done in two ways:

- Depending on settings of a user's presence resource or of his 1XS System or Group Profile configuration ("1XP Admin Client"), the presence service can handle new subscriptions automatically.
  The Dashboard's ability to adjust the corresponding settings is presented in Chapter 4.3.1.4.
  New subscriptions by others may be generally denied or allowed at a selected level.
  As a further option, the monitored user has to be notified by the service in order to let him decide what to do.
- Authorization on a per-user basis.
  Each user owns an **Access Control List** (**ACL**) which defines access permissions of other users. Access may be denied (blocked) or granted at a specific level via this list.
  The ACL contains only exceptions to the rule that access of the minimum presence level is free.
  ACL is of relevance if the presence resource is configured to emit notifications upon new subscriptions. The new subscription will be added to the ACL as **Pending**, the ACL owner has to decide on denial or the granted level. As long as a subscription is pending, the listening user receives the free minimum level of information.

As mentioned above, the presence information of a user consists of personal data and several device-specific data. Although the Dashboard displays the complete received presence information, its main focus is the personal presence information.

The most important attributes of the personal presence information are:

- The **mode,** which basically reflects the location of the user.
  It can be one of ***office***, ***home***, ***mobile*** or ***travel***
  (appropriate JavaScript constant definitions are **AcpPresence.MODE_XXX**).
- The **status**.
  It can be one of ***available***, ***busy***, ***offline***, ***out-of-office*** or ***unavailable***
  (appropriate JavaScript constant definitions are **AcpPresence.STATUS_XXX**).
- An arbitrary **message** defined by the user.

The presence page differs a bit from other resource "view" pages, it requires two resources to work. Publications and the logged-in user's subscriptions are handled by the Presence resource, management of the ACL is handled by the User resource.

Main script variables of the presence page are

- ***myAcpApp***
  The "Application" object reference obtained from the console window
- ***myPresenceResourceStub***
  A "stub" object (class **AcpPresence**) for call of methods of the remote presence resource adapter. The instance is obtained via
  myPresenceResourceStub = myAcpApp.getUser().getResourceByType(AcpResource.TYPE_PRESENCE);
  (applicable after creation of the resource).
  Note: The user resource stub (*myAcpApp.getUser()*) serves to send ACL management commands.
- ***presenceSupport***
  A "support" object (class **PresenceSupport**) for event handling and presence data storage.

The support object registers for reception of events from both the presence and user resources.

Received events are:
(Note: Events "UserACLxxx" are emitted by the user resource)

- ***presentityupdate*** Presence info update for a monitored user
- ***subscriptionended*** One of the logged-in user's presence subscriptions ends
- ***subscriptionblocked*** One of the logged-in user's presence subscriptions is blocked (monitored user has denied access to his information)
- ***UserACLList*** Sent after request for current ACL content. Represents one of the current ACL entries.
- ***UserACLUpdate*** Update of an ACL entry. An "action" field in the event decides whether the ACL entry was added, updated or removed.
- ***UserACLPending*** A new "pending" ACL entry is added.

Internal attributes of the support object are:

- A list of the logged-in user's subscriptions (instances of **PresenceSubscription**). Each subscription contains received presence information (a **PresenceInfo** object) and each presence information object comprises a list of presence information of associated devices (instances of **DevicePresence**).
- The ACL, a list for storage of instances of **AclEntry**
- The logged-in user's own presence information (instance of **PresenceInfo**)

If data stored by the support object changes, the object sends a notification to the presence page's scripts by calling the handler function *uiHandlePresenceNotification(uiNotification)*.
The function parameter *uiNotification* is an instance of **PresenceUINotification**, it contains a notification type code (**PresenceUINotification.TYPE_XXX)** which reflects the event and an additional object which may be a subscription, a presence information or an ACL entry object.

Upon subscriptions and handling of ACL entries, possible presence access levels have to be known for presentation in selection elements of the UI.
Currently only two levels are defined: ***FULL*** for reception of all available information and ***TEL_ONLY*** for reception of presence information based on the phone status.
Because levels are likely to be a subject of extension in the future, possible levels should be determined dynamically.

The support object offers the option to read the levels from server during its initialization.
This is done by *<user-resource-stub>.getAllowedAclLevels(..)*. Having received the levels, the support object emits a special notification (**TYPE_ACCESSLEVELS_READ**).

Note:

The console page fetches determined access levels via function *external_GetPresenceAccessLevels()* of the presence page for its presence settings dialog (see Chapter 4.3.1.4).

4.4.5.2 Presence Publishment

The following image shows the Dashboard's UI for presence publication.

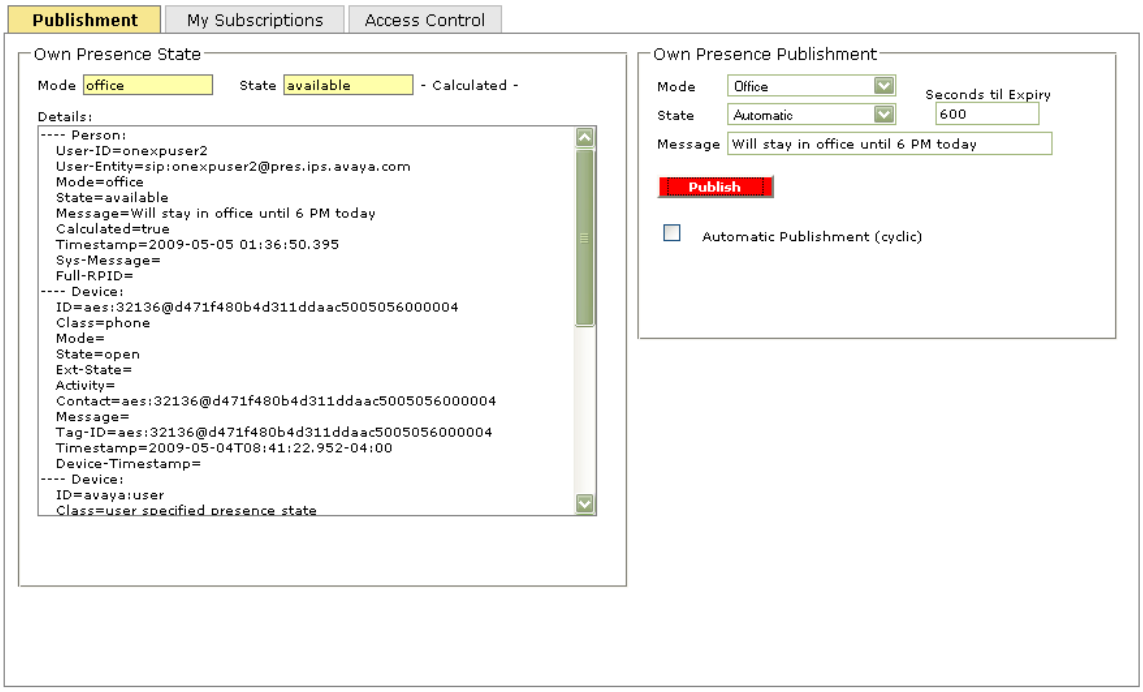

FIGURE 22: Presence Publishment

For publishcation of presence information, the user's identification on the presence server is needed (**User-Entity**). This identification is an attribute of the user's presence information available in the support object.

In order to get the user-entity value, the presence page automatically requests the user's presence information from the server after being loaded (see *myPresenceResourceStub.getUserPresence(..)* in *loadMyPresenceAndAcl()*). The support objects checks the 1XS user-id in received *presentityupdate* events for detection of the user's own presence information. It is stored separately and propagated to

the presence page's scripts via notification **TYPE_OWNPRESENCE_UPDATE**.
The presence page displays all details of this information (see Figure above).

Publication is done by call of
myPresenceResourceStub.publishUserPresence(publishData).
The published data (instance of **AcpPresencePublishInfo**) contains various information describing the user's personal state and that of a specific device which represents the application to which the user is logged in (see function *publishMyPresence()*)

- Personal mode and state of the user.
  A specific state ***calculate*** (**AcpPresence.STATUS_CALCULATE**), displayed as "Automatic" in the UI, means: The presence system has to determine the state automatically (e.g. derive it from the user's phone status).
  Presence information based on automatic state detection is distributed with the "calculated" attribute set "true".
- A text message
- The state of the "application" device (which, in case of the "Dashboard", belongs to the device class "AVAYA Application"). It should be declared as ***open*** (**AcpPresence.DEVICE_STATUS_OPEN**).
  Note: Please follow conventions for definition of a unique **device-id**.

A duration of validity has to be set for a presence publication.
An application usually will publish information periodically (e.g. every 10 minutes) and apply an expiration time which is larger than the period. The Dashboard supports this as an option.

Personal mode and state of the user will be frozen on the presence server upon publication expiration, the published device information becomes invalid.

4.4.5.3 Subscriptions

The UI "My Subscriptions" (see image below) demonstrates the management of subscriptions for reception other users' presence information.

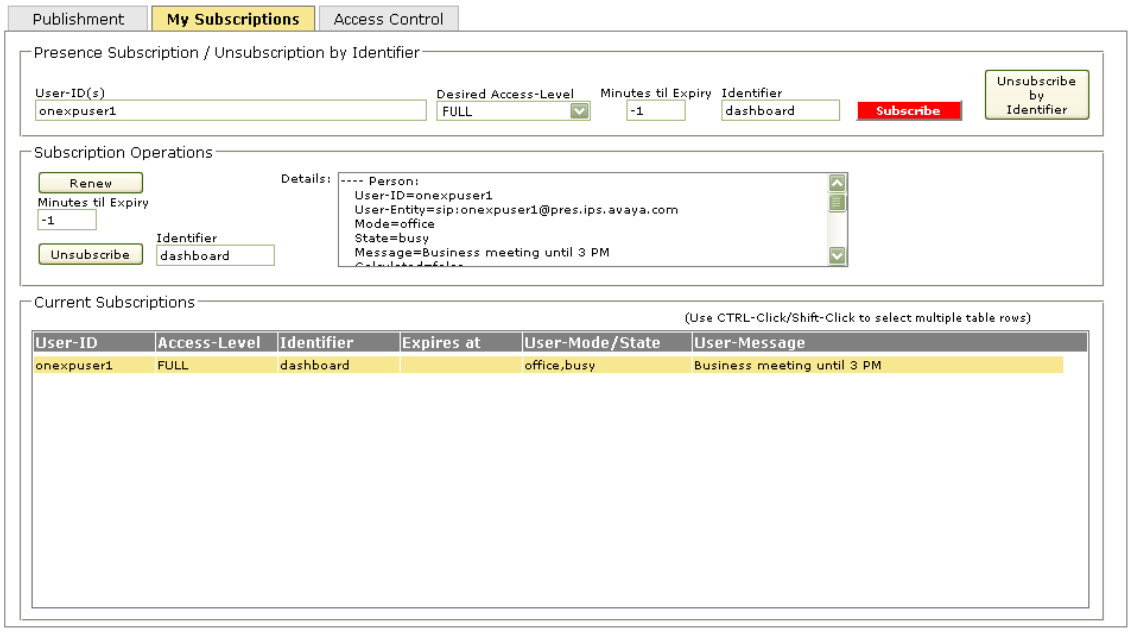

FIGURE 23: Presence Subscriptions

A subscription is based on the 1XS user-id of the user to be monitored. One or multiple (comma-separated) user-id values may be supplied (simultaneous start of subscriptions). Additional parameters of a subscription are

- The desired level of information
- A duration of the subscription (no expiration if value = **-1**)
- A arbitrary "**Identifier**" string

Subscriptions are started via *myPresenceResourceStub.subscribeMultiple(...)* (see function *doSubscribe(...)*).

Besides the request to add the subscription on the server, a new subscription object has to be created (instance of **PresenceSubscription**) and to be programmatically added to the support object's internal list (*presenceSupport.addSubscription(subscription)*), no specific event is fired by the resource.

Furthermore, the subscription has to be added to the UI's table. Each subscription is represented by a table row. The user-id of the subscription serves as a unique table row identification.

After subscription, the resource immediately begins to fire *presentityupdate* events for the monitored user. The support object updates the presence information attribute (**PresenceInfo** object) in the stored subscription object from events and emits notifications (**TYPE_SUBSCRIPTION_UPDATE**) upon changes. The notifications cause updates of displayed presence data in the UI's table.

If one of the subscriptions is selected in the table, possible operations and the details of received presence information become available in the "Subscription Operations" section of the UI (details evidently cannot be shown for selection of multiple subscriptions in the table).

Operations are:

- Renew the subscription (optional modification of the expiration time)
- Unsubscribe
  For removal of a subscription, the same identifier as used for the establishment must be provided.
  The Dashboard prevents use of an improper value. The input field "Identifier" should be understood as additional criterion if multiple subscriptions are selected in the list. Only matching subscriptions will be removed.

The usual way to remove subscriptions is to unsubscribe all subscriptions with the same identifier in a single step by calling *myPresenceResourceStub.unsubscribeByIdentifier(identifier, ..)*.

Imagine an application displays contact search results and intends to show the presence status of each contact. It will subscribe for each of the found contact items. If a new search operation is started, the existing subscriptions have to be removed by a single command (for convenience).

Now imagine that the application contains different windows which both subscribe to presence information of potentially the same users. The windows should work independently, their presence handling should not interfere with each other.

This is accomplished by using different subscription identifiers.

If concurrent subscriptions for the same user's presence are established using different identifiers, then unsubscription is accomplished by sending an unsubscription command for each identifier.

In other words:
If you send an unsubscription command and the resource response signals "success", there is no guarantee that the subscription is removed on the server.
A *subscriptionended* event may appear (to indicate the removal) or not.

For some part of the application which established subscriptions, it is not important to know if the concurrent subscriptions were established by another part of the application. If it has removed "its" subscriptions then it should consider them as actually removed.

How is this handled by the Dashboard:

Having sent the "unsubscribe" command, the subscriptions are programmatically removed from the support object's list (*presenceSupport.removeSubscription(...)*) and from the UI table upon reception of the resource response.

If *subscriptionended* events are fired by the API, the subscriptions will be removed also (depending on what occurs first). This event especially occurs if a subscription expires.
The corresponding notification type sent by the support object is
TYPE_SUBSCRIPTION_REMOVE.

The concurrent use of different identifiers cannot be demonstrated by the Main Dashboard application. A tool to explore the behavior is the **AJAX-Tester** (see Chapter 4.5).

Note: Subscriptions are not persistent, they exist during the user's login session only.

As a demonstration of combined use of different resources and inter-page communications, the presence and contact-list pages of the Dashboard contain mechanisms for close cooperation.
A presence subscription can be established for selected contacts directly from the contact-list page.
The contact-list page delegates the operation to the presence page.
The presence page propagates received presence information to the contact-list page. Thus, the presence status of contacts can be displayed.

4.4.5.4 ACL Handling

The ACL ("Access Control List") UI (see image below) is a means to control other user's access to the logged-in user's presence information.

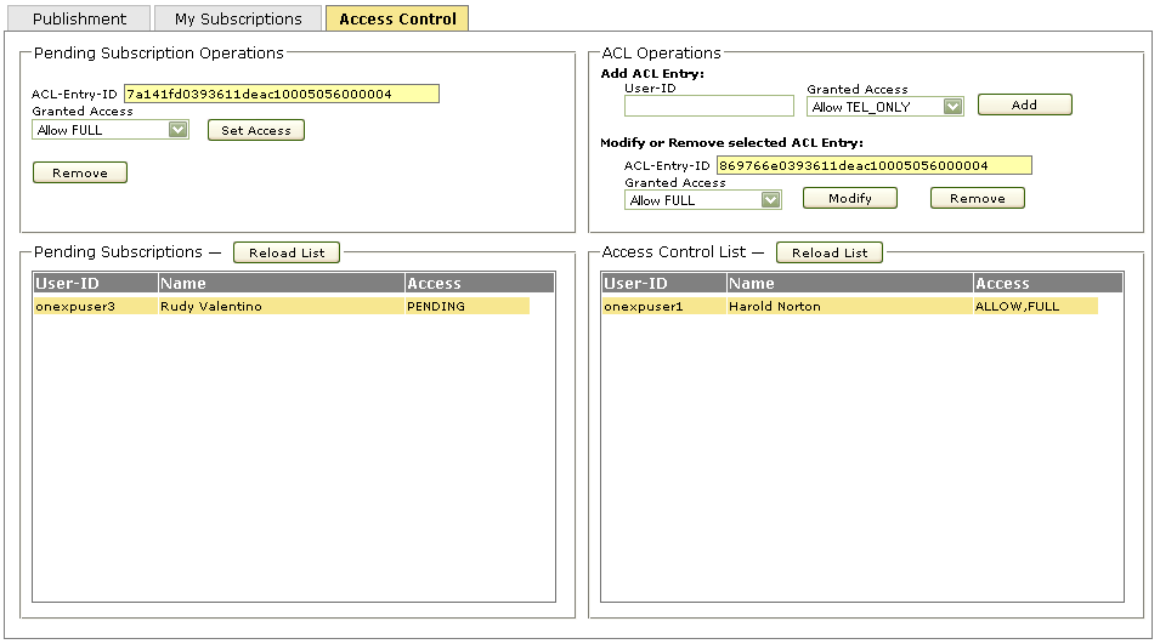

FIGURE 24: Presence ACL Handling

A user's ACL entries are persistent, they remain after the user has logged out from 1XS.

The presence page automatically fetches the initial content of the list from the server after being loaded (see *myAcpApp.getUser().getAclList(..)* in *loadMyPresenceAndAcl()*).
List items are delivered in the form of *UserACLList* events which are received by the support object for data extraction and storage as **AclEntry** objects.

Each time ACL entries are added, removed or modified, the support object notifies the page's scripts (notification types **TYPE_ACL_XXX** resp. **TYPE_PENDING_XXX**).

A user's ACL determines how subscriptions of other users to receive the logged-in user's presence information are to be handled. There is a default (see Chapter 4.3.1.4) which can be overridden by the ACL on a per-user basis.
Handling of subscriptions is defined by an "access type" (you may also find terms like "access name" or "action" in the software) and an "access level".
Via the ACL, the following access types may be set for another 1XS user:

- *BLOCK* Distribution of presence information to this user is inhibited.
- *ALLOW* The user is enabled to receive presence information at a given access level (which may be one of the levels defined in the system).

The Dashboard's presence page provides the ability to add, modify or remove ACL entries.
As an example, the operation to add an entry is:

var aclEntry = new AcpUserAclEntry(<1XS-user-id>, "", <access type name>, <access level>);
myAcpApp.getUser().addAclEntry(aclEntry, ..);

As mentioned before, the special access type *CONFIRM* may be set for default subscription handling. If this is in effect, 1XS fires a *UserACLPending* event for every new subscription at an access level above the minimum.
The "access name" (access type) attribute of the ACL entry added by this event is ***PENDING***.
That means: The user who established the subscription receives the minimum presence level, the monitored user has to decide whether to permit a higher level or to deny access.
He may remove the entry as well, in which case the listening user would continue to receive minimum presence.

The Dashboard presents pending subscriptions in a separate table. Actually there is a single list only stored in the support object. Separation into two lists by the UI is enabled by different notification types emitted by the support object (**TYPE_ACL_XXX** resp. **TYPE_PENDING_XXX**).
This is an option selectable upon creation of the support object. If an application opts for ACL presentation in a single table, notifications **TYPE_PENDING_XXX** will be replaced by **TYPE_ACL_XXX**).

An info box is popped up by the presence page upon arrival of a new pending ACL entry.

4.4.6 Messaging

AVAYA Modular Messaging does not only support distribution of voice audio messages (which may be generated via phone if a user's phone call is redirected to the voice mailbox of another user upon coverage). Also text, images (e.g. in case of FAX) or any kind of attachment file can be transported.

A message may be considered to be a container of **message parts**, each part being a file containing data. Reception, sending and manipulation of messages is managed via a messaging resource but transfer of message parts (up-/download between the user's workstation and some intermediate storage on the server) is done via the **File-Transfer-Servlet**.

4.4.6.1 Main Page

Basic parts of the Dashboard's messaging resource view are represented by the page *\pages\messaging\messaging.jsp*. and the corresponding main script file *\pages\messaging\messaging.js*.

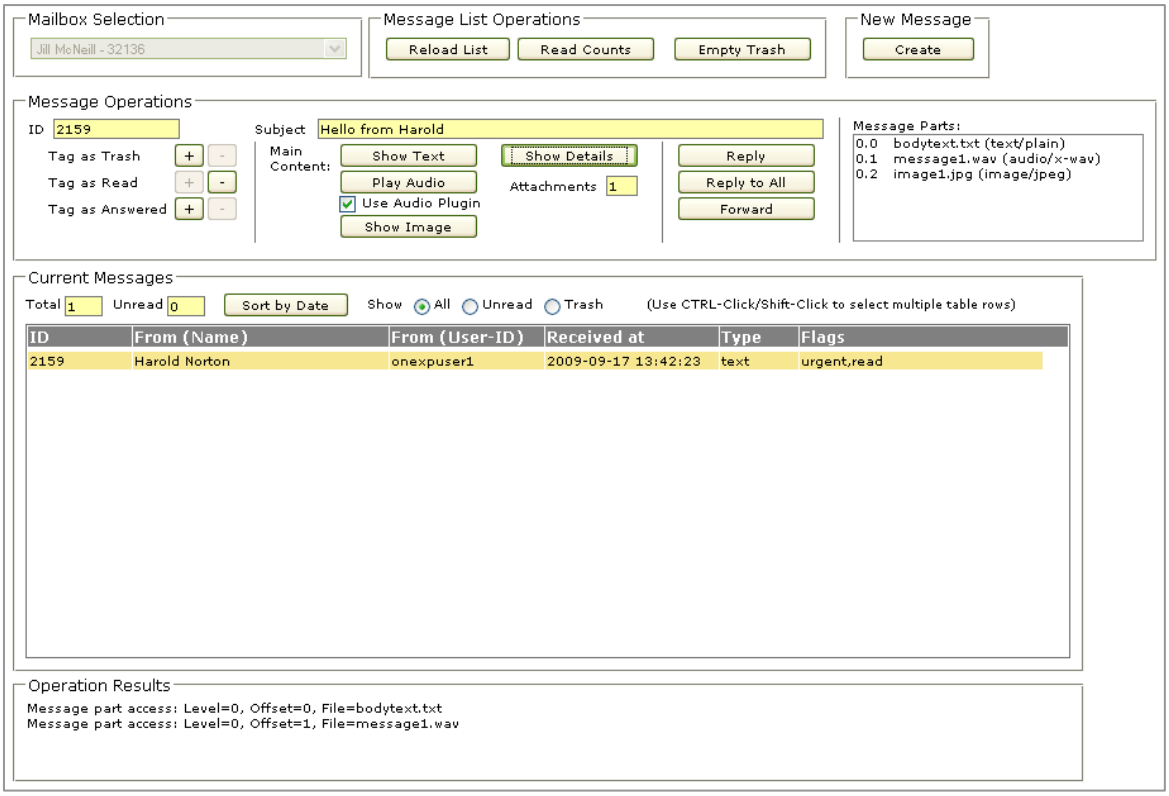

FIGURE 25: Messaging Main Page

The most important script variables of the page are

- ***myAcpApp***
  The "Application" object reference obtained from the console window
- ***currentMessagingResourceStub***
  A "stub" object (class **AcpMailbox**) for call of methods of the remote messaging resource adapter.
  Messaging supports multiple mailboxes for one user (although unusual). All available resource stub references are stored in the array **messagingResourceStubObjects**. The instances are obtained via
  messagingResourceStubObjects =
  myAcpApp.getUser().getResourcesByType(AcpResource.TYPE_VOICE_MESSAGING);

(applicable after creation of the resource(s)).
Resource stubs can be selected by the Dashboard user on the messaging main page ("Mailbox Selection"). The selected stub reference is stored in *currentMessagingResourceStub*.

- ***currentMessagingSupport***
  A "support" object (class **MessagingSupport**) for event handling and message data storage. Because of the "multiple mailbox" feature, a support object is created for each of the resource stubs and stored in the array ***messagingSupportObjects***.
  Upon selection of one of the available resource stub objects, the corresponding support object reference is stored in *currentMessagingSupport*.

4.4.6.2 Main Page Message Handling

Events received by the selected support object are

- ***CommunicationList*** Delivers a single message as result of a load operation
- ***CommunicationStarted*** A new message arrived
- ***CommunicationAttributeChanged*** Some message attribute has changed
- ***CommunicationEnded*** A message is removed

The internal list of the support object contains messages resulting from the mentioned events.
The support object offers the option to load all current messages upon initialization or to reload the list at any time (*currentMessagingSupport.reloadList()*).
The usual handling is to load the list once upon startup of the application. Messages are added then upon reception.

A message is represented by a **Message** object.

Each time a message is added, modified or removed, the support object sends a notification to the page's scripts by calling the handler function *uiHandleMessagingNotification(uiNotification)*.
The function parameter *uiNotification* is an instance of **MessagingUINotification**, it contains a notification type code (**MessagingUINotification.TYPE_XXX)** which reflects the event and the added, modified or removed **Message** object.

The notification **TYPE_MESSAGES_RELOADED** appears after completion of the message list reloading process.
Reloading is requested by the API call *<messaging-resource-stub>.getAllMessages()*.
The message items are sent in form of *CommunicationList* events (notification **TYPE_MESSAGE_LIST_ADD**).

Messages sent by 1XS **currently have no specific sort order**, sorting has to be done by the client application. The support object offers a method which sorts the list of messages by their creation time (*currentMessagingSupport.sortMessagesByTime()*).

Upon notifications, the UI's table of current messages will be updated (functions *tableMessages_XXX()* used). A message is a row in the table.
Each message has a unique ID value assigned which serves as identification of the table row (row-id).

Attributes of a **Message** object are

- A message type. This is a main classification of the message, it can be "**text**", "**audio**" or "**image**" (see constant definitions **Message.MESSAGE_TYPE_XXX**).
  The message type cannot be set by the sender, it will be set by the server depending on available message parts (e.g., a message which contains an audio part only will be of type "audio", a message which contains text only or text- and audio parts will be of type "text").
- Various state attributes like "message is urgent", "message was read", "message is tagged for deletion", ...
- Identification of the sender: 1XS user-id, sender mailbox address

- Address lists of recipients:
  Main recipients (**To**- addresses), recipients which receive a copy (**Cc**- addresses) and recipients which receive a "blind" copy (**Bcc**- addresses)
- A subject string
- A list of message parts, each represented by a **MessagePart** object.
- The count of all available messages and the count of unread ones.

A message part object's attributes are

- The name of a file which contains message part data
- The **MIME-Type** of the data (e.g. "text/plain").
  (see http://www.iana.org/assignments/media-types/)
- **Level**- and **Offset**- values

The **Level** identifies a level of retransmission of a message part. If a message is initially sent, all parts in it have level 0. Assume the recipient forwards the message and decides to include all original message parts in the forwarded message. Then all original parts will appear with level 1 for the next recipient. A further inclusion will set their level to 2, ... and so on.
The **Offset** is a running identification of a part inside of a level (starting with 0).

Message parts with the lowest offset in level 0 and text- resp. audio- MIME-types are handled as the message's **main text- or audio parts**. All other message parts are considered to be **attachments**.

The user of the Dashboard application may select one of the messages available in the UI's list, possible operations are then offered in the "Message Operations" UI section.
The page automatically selects a message if there is only a single one (new or remaining) in the list.

Having obtained the table row-id (message ID) upon user selection of a message in the UI' list, the message can be retrieved from the support object by
currentMessagingSupport.getMessage(messageId).
According to contents of the message object, possible message operations in the UI will be enabled (see *prepareMessageOperations()*).

Operations are:

- Dependent on availability of main text-, audio- or image- message parts, these parts can be displayed/played in separately opened window
- The user may reply to a message, he may reply to all recipients of the message (if there is more than 1 recipient in the message's "To"- and "Cc"- lists) or he may forward the message (if it is not marked to be "private").
  Note that replying to a message of type "image" is inhibited because this type usually represents a FAX.
  There is a resource property ***vm.resource.feature.forwardtoinbox*** which may be adjusted via the 1XP-Admin Client on a system- or user-group- level. If the property is set "false", forwarding of a message should be inhibited.
  You can test the value of the property via *currentMessagingResourceStub.allowForward()*.
- According to the message's flag status, it can be marked as "read", "answered" or "tagged for deletion".
  A message will be automatically set "read" by the server upon access of one of its message parts.
  A message which is "tagged for deletion" (in other words: marked as trash) will be removed from the user's mailbox if he logs out from 1XS or if he explicitly emits the API command *currentMessagingResourceStub.purgeDeletedMessage()* (see UI section "Message List Operations").
  The "answered" flag serves as optional reminder that a reply was sent for a message. It is completely under control of the client application.

If a message is selected, the available message parts are shown as a list in format

<level>.<offset> <filename> (MIME-type).

As to the shown number of available attachments, note that all parts except an existing main text- or audio part are regarded as attachments.
As told before, each message contains the count of total available messages in the user's mailbox on the server and the unread ones. These values are valid at the moment the server has sent the message as event to the client browser. If afterwards a new message arrives in the user's mailbox, the values stored in already existing message objects on the browser become invalid.

The messaging main page of the Dashboard application shows the latest determined total/unread-values at the top of the message list. The API offers a method to access the values directly (*currentMessagingResourceStub.readMessageCounts()*), this API may be called via the "Read Counts" button in the "Message List Operations" section of the UI.

One of the most important operations on a received message is the **presentation of** its **message parts**. This is basically a 2-step process:

- A script on the client browser instructs the server to place the selected message part file in a intermediate storage on the server by calling *currentMessagingResourceStub.getMessagePart(message ID, message part level, message part offset)*.
  The script then waits for the resource response (*handleAjaxResourceResponse()* will be called) which contains the full path to the file on the server.
- A new opened or already existing window (e.g. in an <iframe>-element positioned on the current page) is loaded by request to the **File-Transfer-Servlet**, with the file path added as request parameter. The servlet gets the file content for transfer to the browser's window.

The easiest way to present the retrieved message part data is to let presentation up to the browser. Dependent on default or user-defined browser settings and on the MIME-type of downloaded file data, the message parts will be shown in the browser window directly, by a plugin hosted by the browser (e.g. Windows Media Player or Apple Quicktime) or by a separately started application.

This way of presentation may be implemented by simply calling *myAcpApp.showAttachment(message part file path)* (see */js/waf/acp_browserclient.js*). It is utilized by the Dashboard application to show available main text-, audio- or image- message parts.

If the application should have more control (e.g. show additional information, adjust the window size, set media player parameters), a specialized page has to be set up.
The Dashboard demonstrates this for audio playback with its page */pages/messaging/audioPlayer.jsp* and the related script */pages/messaging/audioPlayer.js*.
The page will be opened if you check the option "Use Audio Plugin" in the UI.

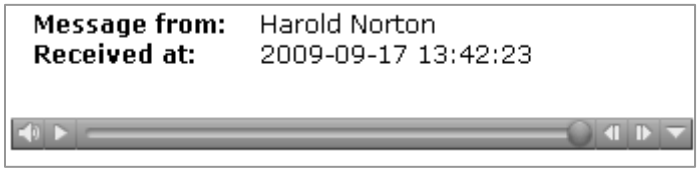

FIGURE 26: Audio Player Page

The page contains a ****- HTML-element (*id="mediaPlayerSpan"*) which is dynamically (JavaScript) populated with an **<embed>**- element upon page load. The <embed>- element contains a **src=...** parameter which has to point to the url of the File-Transfer-Servlet and which has to include the path of the message part file as parameter.
The url can be obtained by *myAcpApp.buildAttachmentUrl(filePath,...)*.

If the <embed>- element is utilized, the browser tries to select a suitable plugin for presentation of the file according to its MIME-type (an Apple Quicktime plugin in the image above). If there isn't any

plugin available, the browser displays a note at the plugin position in the page and guides the user to install some proper software.

If you look into the methods mentioned above (myAcpApp.showAttachment, ...), you will recognize an additional request parameter for the File-Transfer-Servlet:
ACP_FT_REPLY_DISPOSITION_XXX.
According to this request parameter, the servlet will put a "**Content-Disposition**" HTTP header parameter into the response. The content disposition instructs the browser to present the file content ("Open") or to provide a dialog which enables the user to store the file locally on his machine ("Save").

4.4.6.3 Details Dialog

The message details dialog (*/pages/messaging/messageDetails.jsp* and */pages/messaging/messageDetails.js*) presents all the details of a selected message.

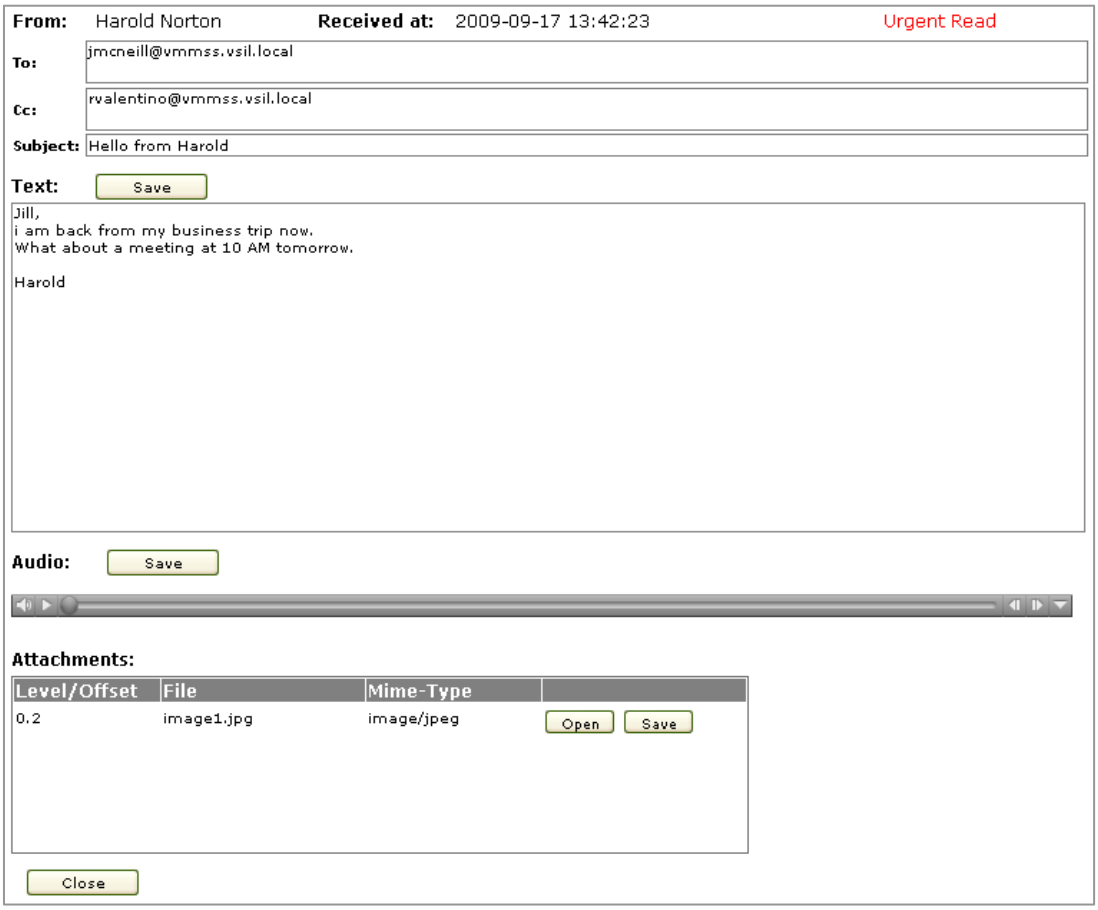

FIGURE 27: Message Details Dialog

The most important script variables of the page are

- ***messagingWindow***
  Reference to the main messaging window, the parent window of the dialog.
- ***myAcpApp***
  The "Application" object reference obtained from the main messaging window via *messagingWindow.dialog_GetApplicationObject()*.
- ***myMessagingResourceStub***
  A "stub" object (class **AcpMailbox**) for call of methods of the remote messaging resource

adapter. It is obtained from the main messaging window via *messagingWindow.dialog_GetResourceStubObject()*.

- ***message***
  The current message (class **Message**) obtained from the main messaging window via *messagingWindow.dialog_GetSelectedMessage()*.

At the top of the details page, basic message parameters like sender name, status flags, "To"- and "Cc"- recipient addresses (comma separated) and the message subject string are displayed.
The rest of the page depends on availability of main text- and audio parts and of attachment parts.

The details dialog presents main text- and audio message part content directly upon opening of the dialog, attachment message parts can be shown/played by user request.

The dialog may request various message part files from the server via *myMessagingResourceStub.getMessagePart(...)*. The response to this API request contains a file path only, an inconvenient situation to make decisions which part of the UI has to be handled.
The much better way is to use special response handler objects instead of providing a single *handleAjaxResourceResponse()* function in the dialog page (and thus use the page window itself as response handler).

The support library defines some handler classes for convenience, they are utilized in the message details dialog.

- **GetTextMessagePartHandler**
  A special resource response handler for support of the main text message part access.
- **GetAudioMessagePartHandler**
  A special resource response handler for support of the main audio message part access.
- **GetMessagePartHandler**
  A generic resource response handler for support of any message part access (especially attachments). This handler object stores level and offset of the message part and an arbitrary operation identifier in order to ease handling of the response.

All the message part handlers call a well-known function of another handler object given upon instantiation of the handler. This second handler object usually is the window which utilizes the handlers. The called function is message part specific, e.g.

- *handleGetTextMessagePartResponse(file path)*
- *handleGetAudioMessagePartResponse(file path)*
- *handleGetMessagePartResponse(file path, level, offset, operational id)*

The following description outlines handling details for the various message parts.

Main text message part handling:

If a main text message part exists, the server is requested to provide the file in its intermediate storage immediately upon page load. A **GetTextMessagePartHandler** object handles the response.

As explained in the previous chapter, it would have been possible to include an <iframe>- element in the details page and load the <iframe>- window by requesting the File-Transfer-Servlet (pointing to the file in the server's intermediate storage) to deliver the message's main text content.

Another solution was chosen in the details page for demonstration purposes: A simple <textarea>- element is used instead of an <iframe>. As a consequence, the text has to be downloaded from the server by means of the **XMLHttpRequest** object available in JavaScript.

The WAF- and support- libraries offer mechanisms to perform the download.
Look into the function *handleGetTextMessagePartResponse(file path).* It uses an instance of **AvayaHttpRequestHelper** (see */js/waf/avaya_ajax_helper.js*) and one of

GetTextMessagePartContentHttpRequestHandler (see */js/reusable/messaging_support.js*) to download the text content.
The **AvayaHttpRequestHelper**- object can be created via *myAcpApp.createHttpRequestHelper(...)*. You have to provide the download url (*myAcpApp.buildAttachmentUrl(filePath)*) and a handler to be called upon reception of the downloaded text (the **GetTextMessagePartContentHttpRequestHandler**- object). If the handler object receives the text, it forwards it to the handler function *handleGetTextMessagePartContextReceived(text)* of your page. This function can update the content of, for example, a <textarea>- element.

Main audio message part handling:

If a main audio message part exists, the server is requested to provide the file in its intermediate storage immediately upon page load. A **GetAudioMessagePartHandler** object handles the response. The details page contains a ****- HTML-element (*id="mediaPlayerSpan"*) which is dynamically (JavaScript) populated with an **<embed>**- element upon response to the message part request. The <embed>-element contains a **src=...** parameter which has to point to the url of the File-Transfer-Servlet and which has to include the path of the message part file.
The url can be obtained by *myAcpApp.buildAttachmentUrl(filePath,...)*.

A parameter **autostart=false** in the <**embed**>- element causes the browser-selected plugin not to play the audio automatically, the user has to click a start button provided by the plugin.

Attachment message part handling:

All attachment message parts are presented in a tabular form, the user may click a button to open one of the attachments. The method applied here was explained in the previous chapter:

- Instruct the server to place the selected message part file in an intermediate storage on the server by calling *myMessagingResourceStub.getMessagePart(...)*.
  A **GetMessagePartHandler** object is set as response handler. The script waits for the resource response (*handleGetMessagePartResponse()* for further processing.
- A new opened window is loaded by request to the **File-Transfer-Servlet**, with the file path of the message part added as request parameter. The browser decides how to present the downloaded file content according to its MIME-type.

All message parts can be saved. This operation requires the "reply disposition" parameter **ACP_FT_REPLY_DISPOSITION_SAVE** set in the request to the File-Transfer-Servlet. File saving is accomplished by *myAcpApp.saveAttachment(file path)* (see */js/waf/acp_browserclient.js*).

There is a resource property ***vm.resource.feature.savetofile*** which may be set on system- or user-group level using the 1XP-Admin Client. If the property is set "false", saving of message part files should be inhibited.
You can test the value of the property via *myMessagingResourceStub.allowSave()*.

4.4.6.4 Composition Dialog

The message composition dialog (*/pages/messaging/messageComposition.jsp* and */pages/messaging/messageComposition.js*) is opened if you want to create a new message or if you want to reply to or forward an existing message.

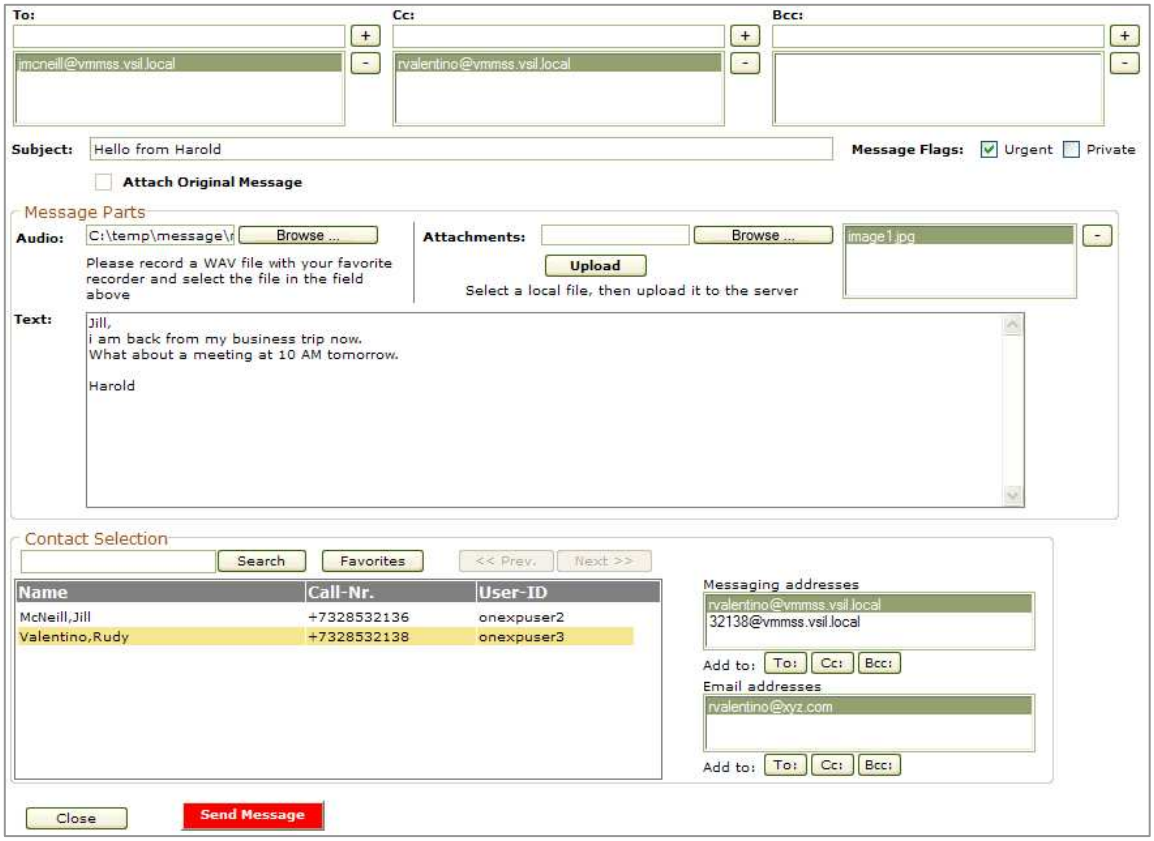

FIGURE 28: Message Composition Dialog

Messaging is closely related to contact-list handling. Each creation of a message implies the tasks of searching for contacts and determination of their messaging addresses (voicemail handles).

Therefore the message composition page maintains objects for access of the user's Messaging- as well as the ContactList- resource. The Dashboard application's concept to present "views" on each of the resources almost separated cannot be maintained in this case.

The most important script variables of the page are

- ***messagingWindow***
  Reference to the main messaging window, the parent window of the dialog.
- ***myAcpApp***
  The "Application" object reference obtained from the main messaging window via *messagingWindow.dialog_GetApplicationObject()*.
- ***messageCompositionMode***
  A mode which defines how the dialog has to work. It is obtained from the main window via *messagingWindow.dialog_GetMessageCompositionMode()*. Possible values are "new", "reply", "replyall" and "forward".
- ***myMessagingResourceStub***
  A "stub" object (class **AcpMailbox**) for call of methods of the remote messaging resource adapter. It is obtained from the main messaging window via *messagingWindow.dialog_GetResourceStubObject()*.

- ***originalMessage***
  The original message (class **Message**) obtained from the main messaging window via *messagingWindow.dialog_GetSelectedMessage()*. This message object is invalid in case of a new message creation.
- ***messageToSend***
  A helper object (class **MessageToSend**, see */js/reusable/messaging_support.js*) which assists in collection of all data for a message to be sent.
- ***myContactListResourceStub***
  A "stub" object (class **AcpContactList**) for call of methods of the remote contact-list resource adapter.
- ***contactListSupport***
  A "support" object (class **ContactListSupport**) for contact event handling and data storage.

Communication with the contact-list resource and handling of contact data will no be described here in deepness. It is quite similar to the processing on the Dashboard's contact-list page (see Chapter 4.4.3).

Please remind that **Contact**- objects received in a list- or search- operation are partially populated only. In order to get the full contact information (including the addresses), a particular contact has to be loaded again via *myContactListResourceStub.getContactById(contact-id)*.

Available addresses of a contact are displayed in UI lists, the user may select one of them for addition to the "To"-, "Cc"- or "Bcc"- address lists of the message.
Both, voicemail addresses (handles) as well as Email addresses are shown because Modular Messaging can be configured to forward Email to a dedicated mail server.

Please note that a voice mailbox usually has 2 addresses, one with the mailbox number as leading part of the address (e.g. "32136@mailserver") and a more expressive one like "jillmcneill@mailserver".

Initialization of the message composition page largely depends on the "composition mode" and is done in several steps (*initializationStep<n>()*).
Several information has to be collected:

- The logged-in user's own contact information may have to be determined (especially his voicemail addresses).
  Why? Upon sending a new message, a voicemail address of this user (the sender) has to be supplied. In a reply to all recipients of a message, the logged-in user's own addresses have to be removed from the "To"- and "Cc"- lists of the reply message.
  The *ownerContact* is determined via
  ownerUserId = myAcpApp.getUser().getUserId();
  myContactListResourceStub.getContactByUserId(ownerUserId,..)
  As a response to this request, the contact-list resource delivers the contact object and the messaging script receives it from its *contactListSupport* object via call to its handler *uiHandleContactListNotification(...)*.
  The determined sender address should match the currently selected voicemail resource. This currently can be accomplished for a number-based address only. Having the mailbox number (*messagingResourceStub.getPropertyValue(AcpMailbox.PROPRTY_MAILBOX)*), a suitable address can be obtained from the contact object via *ownerContact.getVoiceMailAddressForMailbox(mailbox number)*.
- The contact information (voicemail addresses) of the sender of an original message (reply, forward) may have to be determined.
  Why? The sender address included in the original message in most cases is the mailbox-number based address (e.g. "32136@mailserver"). In order to get a more expressive target address, the *senderContact* is determined via the sender's 1X user-id (if available) included in the original message.
  The address may be obtained via *senderContact.getExpressiveVoiceMailAddressString()*.

Note: If the sender owns multiple mailboxes, then currently there is no means to determine an expressive voicemail address for the mailbox which was the source of the original message.

- For a new created message, a recipient address may be preset (Collaboration with the Dashboard's contact-list "view": An address is selected there and the messaging page is instructed to create a new message for this address via *externalCreateNewMessage(address)* in *../messaging.js*. The composition dialog gets the preset address by calling *messagingWindow.dialog_GetTargetMessageAddress()*.
- In a reply to a message or if a message has to be forwarded, the main message text (if there is any) of the original message should be automatically attached to the text to be sent.
  In order to accomplish this, the original message's main text part content has to be loaded.
  Helper objects of classes **GetTextMessagePartHandler** and **GetTextMessagePartContentHttpRequestHandler** (see */js/reusable/messaging_support.js*) are utilized to get the text (the previous "details dialog" chapter contains a description how it works).

Since gathering of all the information requires several consecutive resource requests, a "**Transaction**" is used to speed up Ajax communication.
All the parameters and attributes of the message to be sent are collected in the *messageToSend* object (class **MessageToSend**).

Similar to the access of message parts of a received message, sending of a message is a 2-step process as well:

- Each of the message parts has to be uploaded to a file in an intermediate storage on the server via the **File-Transfer-Servlet**.
- Finally the messaging resource is requested to sent the message, message part filenames are parameters of the request.

The common way to upload a message part to the server is the utilization of a HTML <**form**>-element which has to be populated with some data and then submitted as a request to the server.
The <form>- element's **action**- parameter has to point to the url of the File-Transfer-Servlet.
Its **enctype**- parameter and the inner HTML content of the <form>- element control the data upload.

- For text upload:
  The encoding-type is set to **enctype="application/x-www-form-urlencoded"**.
  An inner element <**textarea name="bodyText"**> contains the text to be uploaded as its "value"- parameter.
  An inner element **<input type="text" name="fileName">** contains the filename for storage on the server.
- For file uploads:
  The encoding-type is set to **enctype="multipart/form-data"**.
  An inner element **<input type="file" name="fileName">** represents the file to transfer to the server.
  Elements **<input type="file">** work autonomous, the have their own user interface which consists of a button (opens a file selection dialog) and a text field (displays the path of the selected file).

For proper collaboration with the File-Transfer-Servlet, ensure to exactly set "bodyText" and "fileName" as "name"- parameters of the inner HTML elements.

Upload forms are provided by the pages */pages/messaging/textUploadForm.jsp* (for text) and */pages/messaging/fileUploadForm.jsp* (for arbitrary files). They contain scripts for access of the form data and submission of the form (*doUpload()*).

These pages are loaded into 3 <iframe>- elements of the message composition dialog, an invisible one for upload of a main message text and visible ones (because of the UI of the file selection <input>- element) for upload of a main audio file and attachment files.

To track the upload process, the composition dialog has "onload"- handlers set for the <iframe>- elements (*iframeOnload(..)*).
If an upload form is successfully submitted to the File-Transfer-Servlet, the servlet's response is a blank page. Therefore, if the form might be needed again for another upload, the upload page has to be reloaded (has to be done for the attachments upload page).
The <iframe>- element's "onload"- handler discriminates the cases "upload completed" and "upload page reloaded" by check if a well-known JavaScript function is defined in the loaded window.

After selection of a file, attachment file upload is started by the user via operation of the "Upload" button. Having the upload completed, the filename is added to a list in the *messageToSend* helper object.
It doesn't matter to remove a file from this list while it is still available in the intermediate storage on the server. Only those message parts will be taken for the message which are specified in the final "send message" request to the messaging resource.

The process of sending the message starts if the user operates the button "Send Message", it comprises multiple steps (*messageSendingStep<n>()*).

- Settings for the message are retrieved from the UI and collected in the *messageToSend* object. Some plausibility checks are performed.
- If a main message text exists, the hidden text upload form is prepared (copy of the text and set the fixed target filename) and submitted to the File-Transfer-Servlet.
- If a main audio file is selected by the user, the audio upload form is submitted to the File-Transfer-Servlet.
- The messaging resource is requested to send the message. This request depends on the *messageCompositionMode*, it may be *myMessagingResourceStub.messageNew(...)*, *myMessagingResourceStub.messageReply(...)* or *myMessagingResourceStub.messageForward(...)*.
  Parameters of the request are:
  - the origin mailbox address (new message only)
  - ID of the original message (replied or forwarded message only)
  - a message subject text
  - "To"-, "Cc"- and "Bcc"- address lists
  - message flags (e.g. "urgent")
  - the list of message part files
  - a flag which instructs the server to include all message parts of the original message
- In case of a reply: Mark the original message as "answered" via *myMessagingResourceStub.markMessageAsAnswered(...)*.
- close the dialog

4.5 AJAX-Tester

Although the Main Dashboard application demonstrates most of the API and its usage, it is impractical to have it show all combinations of operations in every possible situation.

The AJAX-Tester is a tool which may help in additional investigations. It is a means to explore the 1XS API manually.

There are two ways to run the Tester:

- As a standalone tool;
- Concurrently with the Main Dashboard's console. This mode enables additional manipulations which may be impossible to do with the Main Dashboard in a specific state.

In standalone mode, the Tester has its own "Application" object, and it has to manage initialization of the AJAX communication and creation of resources.
When running concurrently with the console, all initialization tasks are handled by the console, and the Tester gets an "Application" object reference from there.

The AJAX-Tester is activated by attachment of a launch parameter to the Dashboard's normal start URL. The parameter is **ajax-tester=standalone** for the "standalone" mode or **ajax-tester=console** for the "parallel to console" mode (see Chapter 4.2.1).

If started as a standalone tool, you have to do initialization manually via the Tester window's "Lifecycle" section.

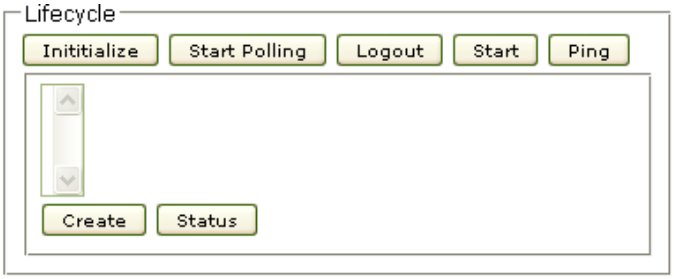

FIGURE 29: AJAX-Tester Lifecycle Section

Perform the following steps:

- Click the button "Initialize" to start the AJAX communication. A list of "Capabilities" (creatable "Resources") will appear.
- Click the button "Start Polling" to enable polling for messages.
  Note: Don't forget to do this. If polling is not started, the browser's session will expire after a while and you will receive curious error messages upon subsequent server requests.
- Select all resources of interest in the list and click "Create".
  Note: Include the "user" resource in creation. Although it is implicitly created, this has to be done due to some idiosyncracies of the tool.

Now you are able to send requests to a selected resource and the Tester will display responses and events.
In order to send a request, you have to provide the API method name and a set of parameters.

There is a basic difference when compared with a WAF-based application. The Framework offers resource-adapter stub objects and you call methods provided by the stub. The AJAX-Tester does not use stubs, it calls resource-adapter methods directly.

For almost all of the resource adapter methods, the method name as well as the parameters and their sequence is identical to the stub methods, but, there are some exceptions. If you make use of the

AJAX-Tester, you will quickly find the rare differences by running the "DebugPlus" monitor concurrently with the Tester.

The following image shows an initialized AJAX-Tester.
A request to search enterprise contacts with a leading "onex" in their names or in their 1XS user-names (user-id) was sent to the contact-list resource adapter.

FIGURE 30: AJAX-Tester

If you run the AJAX-Tester as a companion of the Dashboard's console page, most of the buttons in the "Lifecycle" section are disabled since initialization tasks are up to the console.

Simply click the button "Status" and the Tester will receive all resources currently created by the console. Afterwards, resources may be used as in the standalone mode.

An example demonstrates the power of the AJAX-Tester.

Remember the "Identifier" used in presence subscriptions (Chapter 4.4.5.3). The Main Dashboard is not able to show the effect of concurrent subscriptions (with different identifiers) for reception of the same user's presence information.

Perform the following steps:

- Start the Dashboard and activate the AJAX-Tester to run in parallel to the console.
  Also, enable the "DebugPlus" message monitor (launch parameter "debug-plus=true".
- Let's assume: You want to access the presence information of a user with 1XS user-id "onexpuser1".
- Establish a subscription for "onexpuser1" via the Dashboard's presence page. The identifier is "dashboard".
- Remove the subscription (unsubscribe) and you will recognize an event *subscriptionended* thrown by the resource.
- Establish a subscription for "onexpuser1" again via the Dashboard's presence page.
- Use the AJAX-Tester for establishment of a similar subscription but with a different identifier "tester".
  Select the presence resource.
  Enter *subscribe* as operation. Enter the following parameters (a line for each of them):

onexpuser1, *""*, *FULL*, *true*, *false*, *-1*, *tester* ("" indicates a blank parameter).
Click the button "Resource Request".

- Remove the subscription (unsubscribe) via the Dashboard's presence page (identifier "dashboard") and you will recognize that the event *subscriptionended* is not thrown by the resource.
- Now unsubscribe via the AJAX-Tester.
  Enter *unsubscribeByIdentifier* as operation and *tester* as parameter.
  Click "Resource Request", the *subscriptionended* event is thrown now.