www.jBilling.com

The Open Source Enterprise Billing System

Integration Guide

Copyright

Author

Emir Calabuch and others

Revision number and software version

This revision is 1.4, based on **j**Billing 2.1.0

Table of Contents

Chapter 1
Introduction

Thanks for your interest in **j**Billing!

In this document, you'll find detailed information on how to make use of some of **j**Billing's less known features: the ability to connect the billing infrastructure provided by **j**Billing with your own application programs.

jBilling provides a comprehensive environment for entering, managing, and keeping track of all billing and invoicing operations your organization requires. This, however, is just the tip of the iceberg. Along with its core functionality, **j**Billing provides a rich integration layer that enables your system to tightly interact with the billing infrastructure and streamline all billing-related operations into your organization's work flow.

Scope

This document contains valuable information for application developers that need to integrate **j**Billing into their own systems, or simply wish to explore the possibilities offered by the integration layer.

It also provides a comprehensive reference to the Application Program Interfaces (APIs) available for communication and integration with **j**Billing, and an introductory section in tutorial form that explains how many of these integration tasks can be accomplished.

Requirements

jBilling provides its integration services in three distinct flavors: SOAP (Simple-Object Access Protocol), Hessian/Burlap (lightweight web service protocols) and Java RMI. Explaining how SOAP itself works is beyond the scope of this document. Therefore, in order to fully acquire, and take advantage of, the information contained herein, you'll need to have at least a basic grasp of what SOAP is and how it works. To learn about Hessian and Burlap, visit their homepage. Also, in order to take advantage of Java RMI, you'll need to have knowledge of, and work with, the Java language and platform.

SOAP is not programming language-dependent. Actually, its most interesting feature is that of being understood in almost every system thanks to a large number of libraries or frameworks. Since **j**Billing implements its services through SOAP, your client application is not limited to a specific language or library, as long as it has SOAP support.

Hessian is a fast binary web services protocol that works over HTTP, with implementations available in a number of languages. Like SOAP, it has the advantages of being firewall friendly and the ability to make use of authentication and encryption, yet is similar in speed and bandwidth performance to Java RMI. Burlap is closely related to Hessian, but uses human readable XML instead of binary messages. Neither protocols require an externally defined WSDL file, as required by SOAP.

While there's a small section dedicated to implementing **j**Billing integration calls in other languages/platforms, such as C#.NET and PHP, most examples in this document are provided in Java, so some knowledge of this programming language would prove very useful. Please refer to your language/platform's documentation in order to determine what specific SOAP support it provides, and how to make use of it (specifically, how to call remote SOAP services, and how to pass parameters and decode the returned values).

You'll also need to have a basic understanding of **j**Billing's functionality and usage, which is beyond the scope of this document. You can find this information in other documents available from **j**Billing, such as "**j**Billing's User Guide" and "Getting Started".

Knowing how the program works is of course necessary to understand the scope and usage of each service call.

Last but not least, you will need a running copy of **j**Billing in order to execute the examples contained in this document. Following the “Trend” tutorial on the “Getting Started” document is also necessary to generate some of the data required for the examples to work.

The Need for Integration

A self-contained billing application, while perfectly valid, has an important drawback: to keep it updated, you would need a separate data-entry process that feeds the billing data from your system into the billing application. This is both time- and resource-consuming.

Let’s suppose you own a website with a member’s area. In order to access the member area, the customer would need to pay a monthly fee. If the billing data is kept separately, when a new user subscribes to the website, the billing data would need to wait until somebody enters data into the billing application, and after the payment is cleared the same person would need to update the website in order to grant access to the user.

The manual authorization process would have a heavy impact on the site’s functionality. Even if there are instant payment options, such as credit cards, the user would have to wait for a manual operation to take place before being granted access to the service he paid for.

jBilling solves this problem by rendering common billing and data retrieval operations available to your system via service calls. With a relatively small programming effort, you can integrate the billing application with your own system, so that when the new user signs in, you can instantly pass the data to the billing application, have the payment cleared and grant access without the need for manual operations.

As we’ll see in the following sections, integration services in **j**Billing are almost as comprehensive as the billing functionality provided through the standard web interface. Using the integration layer, you can retrieve lots of useful data from **j**Billing, as well as create, modify or delete users, items, orders and invoices.

Preparing jBilling for Integration

Configuring a Remoting Method

jBilling is distributed with SOAP, Hessian and Burlap already configured, ready to accept connections from clients. To enable Java RMI edit the following file:

```
jbilling\conf\jbilling-remoting.xml
```

Look for the `<!-- RMI Service -->` comment and uncomment the bean definition XML below it. Because RMI is an unauthenticated remoting method, a default caller username and password must be provided. See the “Enabling/Disabling authentication for jBilling web services” section below for information regarding this.

If other remoting methods are not being used, their definition beans may be commented out from this file to give a small improvement in jBilling startup time and memory footprint. However, other files within the `jbilling\webapps\billing.war` archive will need to be modified. For Hessian/Burlap, delete the appropriate mapping from `WEB-INF\`

`jbilling-server.xml`. For SOAP, search for and delete any sections of XML that reference 'cxf' in `WEB-INF\web.xml`.

Securing Integration Access

Overview

While integration services could be an extremely useful feature, they do bring up some security concerns you'll need to take into account. It is critical that the services are not exposed to external parties, otherwise it could be possible for them to invoke the same functionality you're using (and, most probably, you don't want an outsider to be able to create a payment or invoice).

Therefore, the integration services should be exposed only to properly authorized parties. This is, at least in part, guaranteed by **j**Billing, since by default it requires a client application to identify itself via a user name/password pair, before servicing any of its requests. It would also be important to transmit all data over a secure channel, which can be accomplished by using the SOAP or Hessian/Burlap calls over an SSL tunnel, something we'll cover shortly.

Ideally, the web services would be exposed only to the server(s) that require it, and any other parties should be excluded. This can be accomplished by using a firewall application that limits the IP addresses that have access to **j**Billing's integration services. Please refer to the documentation of the Operating System or platform in which your copy of **j**Billing is to run, in order to have some information on how to restrict access to specific TCP ports in your system.

It is also recommended that all service calls are performed by means of an encrypted channel, as provided by the SSL (Secure Socket Layer) protocol. This effectively avoids any threats related to unauthorized interception or decryption of the service calls. SSL also ensures that the party you're engaging communication with is actually your intended recipient, nullifying any impersonation attempts.

In order to determine if the party engaged in communication is actually who it is pretending to be, SSL uses certificates. Therefore, in order to establish an SSL connection to **j**Billing, it could be required to have a copy of **j**Billing's certificate in your development and production systems.

Enabling/Disabling authentication for jBilling web services

By default, all calls to jBilling must be authenticated. In most cases, this is the desired behavior. If Java RMI is the remoting method being used to make calls to jBilling, authentication must be disabled. If the jBilling server is called by clients within an internal network secured from outside access, it maybe preferable to disable authentication to give a slight increase in performance.

To disable authenticated calls, a default username and password will be set on the server. Open the following file for editing:

```
jbilling\conf\jbilling-remoting.xml
```

Uncomment the bean definition with `id="webServicesCallerDefaults"`. Set the default username and password. The user must still be setup according to the section below, "Setting up jBilling to accept external calls".

If SOAP or Hessian/Burlap are being used, an extra step must be performed. Extract the jBilling web archive:

`jbilling\webapps\billing.war`

Open the following file for editing:

`WEB-INF\web.xml`

Comment out or delete the XML that refers to '`Web Services Authentication Filter`'. This includes one `<filter>` tag and two `<filter-mapping>` tags. Save the file and recreate (zip) the web archive.

Enabling/Disabling company security checks

If only one company is using the jBilling installation, unnecessary security checks, which make sure one company is not accessing another company's data, can be disabled to further increase performance. To disable the check, edit the following file:

`jbilling\conf\jbilling-remoting.xml`

Comment out or delete the two XML beans under the comment `<!-- Security Advice for WebServicesSessionBean -->` (ids `webServicesSecurityAdvice` and `webServicesSecurityAdvisor`).

Enabling SSL for jBilling web services

Since web services run on HTTPS, for **j**Billing this is actually managed by Tomcat. Tomcat is the web server that comes with the **j**Billing distribution.

Thus, the first step is to enable SSL in the Tomcat configuration. This is explained in the 'Security' chapter of the jBilling's user guide. Please refer to that document. Once you can use your standard web client over SSL, you can immediately start doing all the web services communication through HTTPS, effectively securing your web services.

A further step is to *force* all the web services communication over SSL. To achieve that extract the billing web archive:

`jbilling\webapps\billing.war`

and edit the file:

`WEB-INF\web.xml`

Notice that there is already a section at the bottom to force HTTPS, but it is commented. Uncomment the XML tag `<security-constraint>` and its contents to force HTTPS. It will use the SSL certificate that Tomcat is using to secure the standard web client. Recreate the web archive (compress with a zip utility program) when done.

Setting up **j**Billing to accept external calls

As mentioned earlier, **j**Billing will not accept all external calls it gets, but will require the caller to identify itself with a user name and password in order to service the request. Otherwise, the request will receive an error in response. This is a simple but effective measure that improves overall security. Of course, this also means you (or the system administrator) will need to set up an account in **j**Billing that will be authorized to perform external calls.

Once you've followed the “Getting Started” tutorial and created the initial billing entity, the user name and password you entered in the entity setup screen will represent the user name and password of **j**Billing's administrator account. You can grant access to this administrator account (or any other account you create for this purpose) to connect remotely and perform service calls.

In order to do so, you'll need to enter **j**Billing's database, and annotate the User ID of the account you wish to authorize for external connection. This number can be obtained from the "BASE_USER" table, you can retrieve it with a simple SQL query:

```
SELECT USER_ID FROM BASE_USER WHERE USER_NAME = 'your-username';
```

Once this number is known, you can proceed to activate external calls permission by adding a line in the PERMISSION_USER table. The PERMISSION_ID contains the code for web services, which is "120" by default. The SQL query that does the insertion is:

```
INSERT INTO PERMISSION_USER(ID, PERMISSION_ID, USER_ID, IS_GRANT) VALUES
(SELECT MAX(ID)+1 FROM PERMISSION_USER, 120, <your-user-id>, 1);
```

The ID for the PERMISSION_USER table is obtained by finding the largest ID number already in use and adding 1. The IS_GRANT field contains a boolean value, if it equals "0", the permission is not granted, if it is assigned "1", the permission is granted. Assuming the User ID obtained in the previous step was "1", the query would be:

```
INSERT INTO PERMISSION_USER(ID, PERMISSION_ID, USER_ID, IS_GRANT) VALUES
(SELECT MAX(ID)+1 FROM PERMISSION_USER, 120, 1, 1);
```

Now the administrator user is allowed to perform external calls. Note that you'll need to specify the user name and password in the caller program.

Note: Since **j**Billing is not tied to a specific database program, it could be possible (but highly unlikely) that your specific database does not support the queries indicated above. Please consult your vendor's manual on how to query and insert data in your database, if the above SQL instructions do not happen to work in your specific system.

Connecting to **j**Billing

SOAP

Once it is properly configured, you'll need to address all integration calls to **j**Billing's web service endpoint. This endpoint is accessible at the same server where **j**Billing was deployed, in a specific URL address, which follows this form

```
http://localhost/billing/cxf/soap.service
```

You can query the service's WSDL (Web Service Description Language) file. It consists of an XML containing a description of the available service calls and parameters used by each call, and could be useful if your programming platform provides a way of automatically generating code for service calls from a WSDL, or you have a tool that can perform test calls (such as **SoapUI**).

To query the WSDL file, you'll need to append the `?wsdl` parameter to the call, for example:

```
http://localhost/billing/cxf/soap.service?wsdl
```

Hessian/Burlap

The Hessian service URL is in the following form:

```
http://localhost/billing/services/hessian.service
```

Similarly for Burlap:

```
http://localhost/billing/services/burlap.service
```

Java RMI

The Java RMI service URL is in the following form:

```
rmi://localhost:1199/RmiService
```

Chapter 2
Integrating jBilling to your Application

Overview

This section discusses the different approaches available for integrating **j**Billing to your application. While **j**Billing provides a unique web services integration layer, there are several ways of interacting with that layer, based on the technology available to the application developer in the target system.

The simplest way of integrating **j**Billing to your application is by making use of the **j**Billing Client API. This however is available only to Java programs, therefore your application needs to be written in Java, or at least you need to have a way of interfacing with Java applications, in order to perform the calls to the integration library.

The **j**Billing client API can perform service calls to **j**Billing in three distinct ways: by means of standard SOAP calls, through Hessian/Burlap calls, and through Java RMI calls. The specific calling protocol is hidden in the library's implementation, so in order to use either protocol, you'll just need to change a parameter, and provide the required libraries.

Service calls can also be performed by means of plain SOAP or Hessian/Burlap calls. This type of interaction is useful for those situations in which you cannot use Java as your programming language (either because your server does not support it or because your application has already been written in another programming language). If you're using Java, you will probably find it easier to use the **j**Billing API than hard-coding SOAP or Hessian/Burlap calls directly.

We'll now examine each of the possibilities and provide some sample code for each instance.

The jBilling Client API

The **j**Billing Client API is a library that contains a set of Java classes that wrap the actual calls that are made to **j**Billing. This simplifies the interaction with **j**Billing as it completely hides the low level connection logic, which you would otherwise need to deal with either directly or via a third party library (such as Axis or CXF). Being Java classes, you must be working in Java or have a way of invoking Java functions from your programming language.

The exact same function calls exposed as web services are available through the API. It simply acts as a wrapper.

As a subproject of **j**Billing, the **j**Billing client API has a few extra advantages. It provides a standardized interface to **j**Billing; so, even if in the future the SOAP interfaces change or new parameters are added, your code will require little or no modification at all to adapt to those changes. Last, but not least, its code is open source, so that you can get down to the implementation details if needed.

The **j**Billing client API provides four different methods of invoking the underlying **j**Billing functionality: as standard SOAP calls, as Hessian calls, as Burlap calls and as Java RMI calls. Choosing which version to use depends on your project's requirements and constraints. Hessian generally has the best features on offer: a fast binary protocol comparable in speed to Java RMI (for Hessian 2), HTTP based for access through restrictive firewalls, the possibility of using HTTP authentication and SSL encryption, and library implementations in a number of languages. Multiple programming language support, however, is probably still best with SOAP.

The advantages of using jBilling client API

The **j**Billing client API is convenient over direct methods (such as directly placing the SOAP or Hessian calls) for a number of reasons.

First of all, your implementation is **cleanly separated from the underlying transport protocol** used. Your code does not change if you switch from SOAP calls to Hessian calls. Also, if other protocols are added in the future, you'll be able to use them as needed without changing your integration code. An example was jBilling 2's introduction of Hessian/Burlap support.

Secondly, the API **absorbs most of the housekeeping activities** you need to perform when using a SOAP library (such as setting up the call parameters and data types). You will just need to instantiate and populate the correct data structures that will contain the input and output data, and call the API to take care of the rest.

A third good reason is that using this API will allow you to use advance deployment features, such as **clustering, load balancing and fail over**. Since the API is a layer in between the client (your application), and the server (**j**Billing), it is the ideal place to abstract different lay-outs of the server deployment, keeping a simplified view from the client. Using these features will simply mean a change of the API's configuration files, without changing any code.

To have an idea of how simple it can be to perform the integration calls, just take a look at the following code. The first example calls **j**Billing (using the **j**Billing client API) to perform a simple customer login sequence (more on this later):

```
import com.sapienter.jbilling.server.util.api.JbillingAPI;

import com.sapienter.jbilling.server.util.api.JbillingAPIFactory;
import com.sapienter.jbilling.server.user.UserWS;

Integer userId = null;
JbillingAPI api = JbillingAPIFactory.getAPI();
try {
       userId = api.getUserId(username);
       UserWS userData = api.getUserWS(userId);
} catch (Exception e) {
       System.out.println("Invalid username: the user does not exist!");
}
```

Compare the above code with the following example, which performs exactly the same calls but using the Apache Axis library:

```
import javax.xml.namespace.QName;

import javax.xml.rpc.ParameterNode;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.ser.BeanDeserializerFactory;
import org.apache.axis.encoding.ser.BeanSerializerFactory;

Service service = new Service();
Call call = (Call) service.createCall();
call.setTargetEndpointAddress("
```

```
        http://localhost/billing/cxf/soap.service");
call.setUsername("myusername");
call.setPassword("mypassword");
call.setOperationName("getUserId");
call.setReturnClass(UserWS.class);
QName qn = new QName("http://www.sapienter.com/billing", "UserWS");
BeanSerializerFactory ser1 = new BeanSerializerFactory(
        UserWS.class, qn);
BeanDeserializerFactory ser2 = new BeanDeserializerFactory(
        UserWS.class, qn);
call.registerTypeMapping(UserWS.class, qn, ser1, ser2);
try {
       Integer userId = call.invoke(new Object[] { username });
       UserWS userData = call.invoke(new Object[] { userId });
} catch (Exception e) {
       System.out.println("Invalid username: the user does not exist!");
}
```

As you can probably notice, a good part of the Axis example deals with the setup of the SOAP calls, so the actual logic of the call is somewhat obscured by the amount of housekeeping activities that the code needs to perform. Remember also that the **j**Billing Client API example works just as well with Hessian (you'll just need to change an XML file, no coding is needed), whereas you'll need to entirely rewrite the Axis example in order to change transport protocol.

We advise you to take advantage of the Client API. Unless you do not work in Java, you'll probably welcome its simplicity and convenience.

Using the jBilling Client API

To use the **j**Billing Client API, you'll need to copy some library files and render them visible in your application's class path.

All of the API classes are located in the `jbilling_api.jar` file located in your **j**Billing distribution.

The API also makes use of several third-party libraries, such as the Log4j library and Commons Logging, which provides a powerful logging infrastructure; Spring, which handles configuration and remoting; CXF, a SOAP library; and Hessian, for Hessian/Burlap support. You'll therefore need to provide the `log4j.jar`, `commons-logging.jar` and `spring.jar` files in your class path, if your project does not already include them. These files are in the `WEB-INF\lib\` directory of the `jbilling\webapps\billing.war` web archive, as well.

In order to work, the API also requires a configuration file, the `jbilling\conf\jbilling-remote-beans.xml` file, which needs to be added to your project. This file defines some important parameters that the API will later retrieve and use. More on this file, its purpose and available settings in the next section.

Depending on the underlying transport protocol you choose to use (Hessian or SOAP), you'll also need to include several other libraries. These requirements are explained in detail in the sections below.

Once you have set up your environment for using the **j**Billing Client API, you can use the library by using a factory method to retrieve an API interface, which you'll later use to

place the integration calls. You can retrieve an interface to the API in the following manner:

```
JbillingAPI api = JbillingAPIFactory.getAPI();
```

The `JbillingAPI` object allows you to place integration calls directly. Each call performs a specific functionality, for example `api.getUserId()` retrieves the **j**Billing User ID of a user, given the user name. Obviously, each call requires different parameters and returns different data, according to its use.

The rest of the API library contains classes that define the parameters you can use as input and receive as output from the service calls. For example, the `UserWS` class contains a set of data regarding the **j**Billing user, such as the user name, password, date of creation, etc. As usual in Java, most of these properties are accessible using the getter and setter methods, such as `getUsername()` or `setPassword()`.

Most of the integration effort goes into setting values into these structures and passing them to the appropriate service call. For example, you'll fill a `UserWS` structure and pass it to the `createUser()` service call. Most of the API follows this simple logic.

Configuration

The `jbilling\conf\jbilling-remote-beans.xml` file contains some fundamental parameters that define the connection to the **j**Billing server you wish to communicate with. This file is required by the **j**Billing API library for all connection alternatives.

Each connection option is contained within XML bean configuration tag with `id="apiClient"` as an attribute. There are four example configuration beans – one for each of the four different remoting protocols. Only one bean should be uncommented at any given time, which is the configuration the API will use. The comment above each bean indicates which protocol it configures.

Common properties to the SOAP, Hessian and Burlap protocols:

`username` and `password`: these values must correspond to a valid account in **j**billing that has been granted permission to execute web service calls.

Properties and setup specific to each remoting method are detailed in the sections below.

The SOAP (CXF) Web Services Method

When using this protocol for service calls from the **j**Billing Client API, you will need to include the CXF library (`cxf.jar`), which provides support for SOAP. A few support libraries are also required: `XmlSchema.jar`, `wsdl4j.jar`, `FastInfoset.jar` and `neethi.jar` You can find all the required `.jar` files in your **j**Billing distribution, in the `WEB-INF\lib\` directory of the `jbilling\webapps\billing.war` web archive. All these libraries will need to be added to your application's class path in order to be usable by the API.

While SOAP is a solid way of communicating with remote applications, it requires the parties to exchange XML files when the service call takes place. These files can be quite massive, and a great deal of time is wasted in serializing parameters into an XML, and

deserializing the response. For this reason, SOAP is relatively inefficient when it comes to implementing fast service calls. Therefore, you're advised to use Hessian or RMI, since they are much faster.

Properties

`address`: The URL defined in this parameter points to the **j**billing web services endpoint, as defined in the “Connecting to **j**Billing” section of the “Preparing **j**Billing for Integration” chapter of this document. You can specify the request protocol (either http:// or https://) and the port number, if it differs from the standard port (which is 80 for HTTP and 443 for HTTPS). For example,

```
address="http://localhost:8080/billing/cxf/soap.service"
```

The Hessian/Burlap Method

The Hessian and Burlap versions of the Client API provides a different communication protocol. Hessian is free from the overhead derived from SOAP's extensive use of XML, and therefore provides faster calls. Burlap allows XML messages to be used in the cases where human readability is needed.

If using Hessian or Burlap, `hessian.jar` is required, and can be found in the `WEB-INF\lib\` directory of the `jbilling\webapps\billing.war` web archive

Properties

`serviceUrl`: The URL Hessian should try to connect to, as defined in the “Connecting to **j**Billing” section of the “Preparing **j**Billing for Integration” chapter of this document. For example:

```
<property name="serviceUrl"
 value="http://localhost:8080/billing/services/hessian.service"/>
```

`hessian2:` A Hessian specific property. It indicates whether Hessian should use the newer version 2 protocol, which is faster than version 1. Although Hessian 2 is still a draft specification, a value of `true` is generally recommended for performance.

Java RMI

RMI is the standard Java binary remoting protocol, which provides excellent performance and required not extra libraries. As it is unauthenticated, some setup is required on the server. See the “Enabling/Disabling authentication for jBilling web services” section of the “Preparing **j**Billing for Integration” chapter for more information.

Properties

`serviceUrl`: The URL RMI should try to connect to, as defined in the “Connecting to **j**Billing” section of the “Preparing **j**Billing for Integration” chapter of this document. For example:

```
<property name="serviceUrl"
 value="rmi://localhost:1199/RmiService"/>
```

Integration with non-Java Applications

If your application is not programmed in the Java language, you still have the opportunity to integrate your system to **j**Billing via direct SOAP calls. The beauty of SOAP is that of being independent of any particular language or framework in order to be usable. SOAP is available for many, if not all, of the languages commonly used for web programming.

Covering all possible programming languages is impossible, so we've limited ourselves to provide two of the most used programming languages for web applications, besides Java: C#.NET and PHP. Both provide SOAP support and therefore are perfectly capable of integrating with **j**Billing. If your system of choice is not one of these, please refer to your language's documentation for clues on how to use SOAP service calls effectively. You can however use these examples as a guideline on how to perform specific operations with **j**Billing.

The provided examples perform the same login sequence as seen in the Integration Tutorial section of this document, but implementing it in each of the languages and platforms. Once the implementation in your specific language is clear, we advise you to follow the Integration Tutorial as well, since it provides useful insight into the purpose of some of the most used web service calls. This information should be of interest even if your system is not built on Java, as it explains the purpose of the calls, rather than their specific Java implementation.

Integration with C# .NET

C# .NET provides seamless SOAP support, which requires no extra resources to be added to your application's execution environment. Refer to the MSDN resources for more information on what support options you have.

The difficult part in implementing SOAP on C# is to map the input and output parameters of the web call into appropriate data structures. Since there are many different calls, it quickly gets difficult to create all the necessary classes.

A handy tool, which is distributed as part of the Visual Studio® distribution, is the `wsdl.exe` utility, which takes care of converting a WSDL file into appropriate data structures and creates an output file that contains a class that performs all web service calls. You can find this command line tool in the "`Visual Studio 8\SDK\v2.0\Bin`" directory of your Visual Studio installation. Consult your Visual Studio documentation or just call "wsdl.exe" with no parameters to obtain some information on how to invoke this tool and what parameters are acceptable.

If you use **SoapUI**, an Open Source tool for testing and handling web services, you can also set this program to generate the necessary .NET files (it supports generation of Managed C++, VisualBasic and J# code, as well as C#). Simply indicate the location of the above mentioned `wsdl.exe` tool in your system in the Preferences dialog, connect to the **j**Billing web service url (as defined earlier) from SoapUI, and click on "Tools -> .NET 2.0 Artifacts" in the main menu. You'll be presented with a set of generation options (WSDL file to use as input, output directory, the user name and password to pass to the service, etc.).

Once the class containing all the web service definitions has been generated (which will be named `WebServicesSessionLocalService.cs`), it is a simple matter of using the generated class in your code, in a way that is akin to the jBilling Client API for Java:

```
using WebServicesSessionLocalService;


// create the interface instance of the class.
WebServicesSessionLocalService service = new WebServicesSessionLocalService();
int userId = service.getUserId(username);
if (userId > 0) {
       UserWS userData = service.getUserWS(userId);
}
```

Integration with PHP

PHP provides SOAP support via an add on library (distributed with PHP itself). You'll need to activate the library support in your `PHP.INI` file and probably need a library file (`libxml`) to be present in your system in order to use the SOAP support (refer to the PHP manual for details on how to activate SOAP support).

While PHP provides SOAP support, the hardest part of implementing remote SOAP calls is, as with most other programming languages, to define all the input and output parameters that are required for the SOAP call. There are automated ways to accomplish this part of the job, such as the PEAR library, which comes with the default PHP installation (unless you've used the --without-pear configuration option during installation).

PEAR can generate the PHP code for the data classes or it can generate the classes on the fly for immediate usage on the program. While the first option is the recommended procedure (this way you can avoid parsing and generating the WSDL file each time your code executes), for simplicity we'll demonstrate the use of the PEAR module with the second option, using the classes on the fly:

```
require_once 'SOAP/Client.php';

$requestParams = array ('user'=>'admin', 'pass'=>'asdfasdf');
$wsdl = new SOAP_WSDL('http://localhost/billing/cxf/soap.service?wsdl',
                           $requestParams);
$client = $wsdl->getProxy();
$userIdParams = array('in0' => $username);
$result = $client->getUserId($userIdParams);
$userId = $result->getUserIdReturn;
$userDataParams = array('in0' => $userId);
$result = $client->getUserWS($userId);
$userData = $result->getUserWSReturn;
```

This is quite straightforward, as the PEAR library has done most of the dirty work of parsing the WSDL and setting up the calls. The `$requestParams` is an associative array containing the user name and password for the **j**Billing authentication. Parameters for the web service calls are passed in associative arrays as well.

If you wish to generate the code for the web service calls for use in the program without having to parse the WSDL every time, you can explore how to generate the code and

save it to a file with the `$wsdl->generateProxyCode()` function call of the `SOAP_WSDL` object.

Chapter 3

An Integration tutorial

Overview

This section provides an introduction to the use of **j**Billing integration calls. It builds on the “Trend” tutorial found in “**j**Billing's Getting Started Guide”. As such, if you wish to follow the examples in this section, you'll need to follow the Getting Started Guide tutorial, creating the Trend entity and adding the item and orders.

The examples in this section will mainly use the **j**Billing client API. If you're unable to use this API (most probably because your system does not make use of the Java programming language), you'll need to substitute the API calls with direct SOAP requests. Please, refer to the “Integrating **j**Billing to your Application” chapter, which explains the options for performing service calls to **j**Billing in more detail and provides examples for some programming languages and SOAP APIs available.

The Trend Website

In the “Getting Started” guide, you were introduced to **j**Billing's capabilities by means of a small tutorial centered around a hypothetical “Trend” company, which runs a website. Following the tutorial, you were able to create the Trend entity, added an Item and created an order and invoice.

We'll build on this example, examining how **j**Billing's integration features could be applied to the Trend website, which needs to integrate to **j**Billing.

Controlling Entrance to the Paid Area of the Website

Being a paid service, the Trend website obviously needs to determine whether each user that enters the site is a paying customer or not, in order to provide its services only to those that have paid for it. This can be accomplished by the use of the usual user name/password pair, which will be provided to each customer at the end of the subscription process, once the first payment has cleared.

Of course, the invoicing and payment parts are handled by **j**Billing, so the website needs a way of knowing when a given user name corresponds to a paying customer.

A solution could be to maintain a local record of customers and check each login attempt against it, but this actually represents a problem: how would the program know when a customer has paid or not? Without integration calls, someone would have to keep the data in this record updated, a manual operation that could introduce errors and requires time.

Another important consideration is that the website charges its customers a monthly fee. This means the status of customers could change at any time, if the user, for example, has not paid the fee for the current month (it would fall back to non-paying). A local record of customers would not have this information, at least not until the next manual update.

The solution is throwing in an integration call to **j**Billing to ensure the customer at hand has paid his fee and is authorized to access the website's resources. **j**Billing takes care of checking its internal data and determines if the client's payment has been processed and whether his subscription is still valid.

The website's login form retrieves the customer's username and password, performs any local validations and checks the user's status in **j**Billing (which will indicate his current payment status and therefore whether he should be authorized to login or not).

The first step (regarding integration calls and not local validation) is therefore to retrieve the **j**Billing's User ID number, and request the user's data, so that the current status can be determined. The sequence would be:

```
import com.sapienter.jbilling.server.util.api.JbillingAPI;

import com.sapienter.jbilling.server.util.api.JbillingAPIFactory;
import com.sapienter.jbilling.server.user.UserWS;
/*
 * We assume the String variable “username” contains the username
 * string as entered by the user in the login form.
 */
Integer userId = null;

// Create and initialize the jBilling API.
JbillingAPI api = JbillingAPIFactory.getAPI();
boolean canLogin = false;

try {
    userId = api.getUserId(username);
    if (userId == null) {
       /*
        * This shouldn't happen as the API should issue an exception if the
        * user does not exist, but test it anyway. In this case, we simply
        * generate an exception that is catched later in this block
        * of code.
        */
    }

    // With the user id just retrieved, we can query the user's data.
    UserWS userData = api.getUserWS(userId);

    /*
     * The user data contains many information about the user, but in
     * this case we're mostly interested in the statusId field of the
     * UserWS class.
     * This field values are: 1=active, 2=Overdue 1, 3=Overdue 2,
     * 4=Overdue 3, 5=Suspended 1, 6=Suspended 2, 7=Suspended 3,
     * 8=Deleted. Status Ids from 1 through 4 indicate the user is able
     * to login, all other codes cannot login.
     */
     int statusId = userData.getStatusId().intValue();
     if (statusId > 0 && statusId <= 4) {
       /*
        * The user can login to the system, as his current status is ok.
        * Just print a notice in case he's late with his last payment.
        */
       canLogin = true;
       System.out.println("User can log in");
       if (statusId != 0) {
             System.out.println("The user's payment is overdue!");
       }
    }
} catch (Exception e) {
    /*
     * The user does not exist in Jbilling's records. The login request
```

```
        * should be denied, and perhaps an error message issued back to the
        * caller. Here, we just print an error message to stdout.
        */
        canLogin = false;
        System.out.println("Invalid username: the user does not exist!");
    }

    if (canLogin) {
          // Here you can grant access to the reserved area.
    } else {
          // Here you can deny entrance to the reserved area.
    }
```

As you have probably noticed from this example, using the **j**Billing API is very straightforward and simple. The burden of setting input and output parameters for the call has been hidden in the implementation of the library, and calls seem simpler. Take a look at the sample code in the “Integration with non-Java applications” section to see why we strongly recommend this approach for Java applications. And, as a bonus, you can easily switch between SOAP calls and Hessian calls with just a change in an XML file!

The code in itself is quite simple. We acquire the user's ID with the call to `api.getUserId()` and use that ID to retrieve the user's account data via `api.getUserWS()`. With the user data at hand, we can check if the user's status in **j**Billing warrants his entrance to the paid area. The flag `canLogin` is set so that we can later determine what response to give to the user.

There's even a simpler way to login into the system: the `authenticate()` function call requires two String arguments (user name and password) and returns a 0 if the user is in a state that allows entrance to the paid area, whereas a non-zero response indicates the user does not exist or is in disabled or suspended state.

It however does not provide information about what specific state the user is in, or if it is overdue or not. A quick example using this function would be:

```
import com.sapienter.jbilling.server.util.api.JbillingAPI;

import com.sapienter.jbilling.server.util.api.JbillingAPIFactory;
import com.sapienter.jbilling.server.user.UserWS;

// Create and initialize the jBilling API.
JbillingAPI api = JbillingAPIFactory.getAPI();
Integer result = api.authenticate(username, password);
if (result.intValue() == 0) {
      // The user is able to loging.
} else {
      // The user cannot login. The return value indicates why.
}
```

Trial Period Management

You can easily set up trial accounts so that would-be customers can enter your paid area for free before actually being charged. Since a trial period in **j**Billing is managed through a normal order, this will also give a practical demonstration of how orders can be entered via the integration calls.

jBilling allows you to practice two different trial period policies: with or without pre-authorization. Pre-authorized trial periods require the user to enter payment information (usually in the form of credit card data), which will be validated before the trial period can begin. Non pre-authorized trial periods do not require this validation phase.

Trial periods are managed by simply delaying the “Active Since” date of the purchase order by a number of days. Let's say Trend wishes to grant a free 15 day trial period to new customers, so we're going to enter orders with an “Active Since” date of TODAY+15.

This means the login code we just saw will work correctly for trial customers as well. The user appears as “Active”, but the invoicing will not take place until the order becomes active. In addition, when pre-authorization is required, you'll need to provide some payment information, that will be validated before the order insertion ends correctly.

When the trial period ends, the order will become active and **j**Billing will automatically start invoicing the customer. If the customer changes his mind and wishes to cancel his membership to the site (either before the trial expires or after actual invoicing has taken place), you'll have to provide a cancellation page that will take care of deleting active orders, so that no further automatic invoicing takes place. We'll take care of the cancellation code in the next section.

We will need to request all necessary data from the user. Since we will be creating a new user in **j**Billing, we will require the user to enter the user name/password pair she intends to use to access the site in the future. This is the minimal information we'll need to successfully call the user creation service for a non pre-authorized trial, but in production code you would probably need to add contact information to the user record, and if you're implementing pre-authorized trials, you'll need to request some payment information as well. These can be added as parameters to the user creation process.

Once the new user is created, we can add an order to that user with the “Active Since” date set to TODAY+15, and “Active Until” date set to “null”, which indicates the order is permanently active and will continue to invoice until it is canceled. We are assuming the Item has been created as indicated in the “Getting Started” tutorial, so that we have a “Banners” category and a “Front Page Banner – Monthly Fee” item created under this category. In the example, we assume the Item ID code is “1”, in a more realistic situation you could have several item types to choose from, or you would otherwise need to determine what the Item ID is from the **j**Billing user interface.

We also assume the periodicity code for the order is “2”, which maps to monthly payments. This however depends on the setup of the specific instance of **j**Billing, you can determine what specific code each period has by consulting the **j**Billing User Interface under “Orders -> Periods”.

One small notice: the item we're creating an order for is, according to the Getting Started guide, the fee for displaying a banner in Trend's front-page. To keep the example consistent, let's say the login procedure illustrated above allows the customer to login to the area where he can upload and update the banner itself, and perhaps see some statistics for his banner's exposure. The code that performs the banner rotation would probably need to determine what banners to show (maybe a local database, which of course needs to be kept in sync with billing data, something we'll cover in another section).

```
/*
 * We assume the String variables “username” and “password” contain the
```

```
 * new user's login data as entered by the user in the trial registration form.
 */
Integer userId = null;

// Create and initialize the jBilling API.
JbillingAPI api = JbillingAPIFactory.getAPI();

try {
       // Create the user's record.
       UserWS newUser = new UserWS();

       // Fill in the new user's data.
       newUser.setMainRoleId(new Integer(5));  // Role “5” = “Customer”.
       newUser.setStatusId(new Integer(1));    // Status “1” = “Active”.
       newUser.setUserName(username);
       newUser.setPassword(password);
       // Refer to Appendix A for language codes.
       newUser.setLanguageId(new Integer(1));
       // Refer to Appendix A for currency codes.
       newUser.setCurrencyId(new Integer(1));
       newUser.setCreateDatetime(new java.util.Date());  // = now
       // If you're entering credit card information, you'll need to create
       // a CreditCardDTO object and fill it with data, and assign the data
       // to this user via the setCreditCard() method of UserWS. Same goes for
       // Contact info, a ContactDTO object can be assigned via setContact().

       // Call the “create” method on the api to create the user.
       Integer newUserId = api.createUser(newUser);

       // Now let's create a new order line.
       OrderLineWS line = new OrderLineWS();
       line.setPrice(new BigDecimal(10));      // Order price = 10.
       line.setTypeId(new Integer(1));    // Type 1 = “Item”.
       line.setQuantity(new Integer(1));  // Quantity = 1
       line.setAmount(new BigDecimal(10));     // Total amount = 10
       line.setDescription("Banner on Front Page");
       line.setItemId(new Integer(1));

       // Now create an order that contains the order line just created.
       OrderWS newOrder = new OrderWS();
       newOrder.setUserId(newUserId);
       newOrder.setPeriod(new Integer(1));
       // Now add the order line created above to this order.
       newOrder.setOrderLines(new OrderLineWS[] { line });
       newOrder.setBillingTypeId(new Integer(1));  // Prepaid order.
       GregorianCalendar activeSinceDate = new GregorianCalendar();
       activeSinceDate.add(Calendar.DATE, 15);
       newOrder.setActiveSince(activeSinceDate.getTime());

       // Now create the order.
       Integer newOrderId = api.createOrder(newOrder);

} catch (Exception e) {
       /*
        * There was an error during the user or order creation. Just print an
        * error message.
        */
```

```
        System.out.println("Error in user or order creation");
}
```

A few details need to be pointed out: first, we create the user and order in two separate steps, since we wanted to show how both operations worked on their own, but the API provides a way of doing both operations in one step. Instead of calling `api.createUser(newUser)` and `api.createOrder(newOrder)`, you can just call `api.create(newUser, newOrder)`.

This call will also charge the user credit card, but not as a pre-authorization, but as a full charge (capture). The method 'create' does four steps:

1. Creation of the user
2. Creation of the purchase order
3. Generation of an invoice based on this purchase order
4. Processing of the payment to get the invoiced paid.

In the example, we only got the first two steps, which is consistent with the requirements of a free trial: the invoice and payment should only happen when the trial is over and the 'active since' of the order reached. If you do the single-step 'create' call, be sure to set the User ID of the new order to “-1”, since you're creating the user at the same time you're adding the order and therefore the correct User ID is not yet known.

If you wish to do pre-authorization, you'll need to fill credit card information and add it to the new user data. Then, for the second step (order creation) you should call

`api.createOrderPreAuthorize(newOrder)`

instead of calling the `createOrder()` method. Note that in this case you cannot use the `create()` method indicated above, as it does not do pre-authorization, so you'll need to perform the two separate steps. The `createOrderPreAuthorize()` method returns a `PaymentAuthorizationDTOEx` structure, which provides info on the authorization released. At this point, you're probably most interested in knowing if it was successful, so just test the boolean result by calling `getResult()`. The following snippet shows how:

```
PaymentAuthorizationDTOEx payment = api.createOrderPreAuthorize(newOrder);

if (payment.getResult() == false) {
        System.out.println("Your payment was not accepted! Trial not active!");
        // Since the user has been already created, you might want to delete it
}
```

Keep in mind that if you create the user and the trial order in two separate steps, you could have a situation in which the user gets created but the order cannot be submitted. So, if the code catches an exception, test if the user exists (if you received a User ID from the first call that is non-null, for example) to intercept such a situation.

Finally, this does not test whether the user existed previously with that user name. When creating a user, you can first validate if the user exists, by querying its User ID, as done in the login section above, and display an error or request a different user name. It also does not verify whether the user has already created other trial accounts.

To intercept situations in which a user creates trial accounts just to cancel them before billing takes place to create another trial account, ask for some contact information (such as, for example, a valid e-mail address or just save the IP address of the caller) and verify that this information is not repeated in previous trial accounts.

Contact Information

The users data structure contains fields to keep Contact information along with your **j**Billing's user account. Adding or updating contact data is quite simple: you just add it to the user's data structure whenever you call `createUser()` or `updateUser()`. There's a shortcut call that allows you to edit contact information as well:

`updateUserContact()`

To add contact information to an account, create the `ContactWS` data structure and fill it as necessary. Most data fields in this structure are self-explanatory, such as `postalCode` or `faxNumber`. For example, the following code snippet creates a `ContactWS` structure and puts some data into it, and later it passes it to the user creation process:

```
// Create the ContactWS structure
ContactWS contactInfo = new ContactWS();

// Put some data into it
contactInfo.setPostalCode("12345");
contactInfo.setFaxNumber("555-123456");
contactInfo.setEmail("foo@bar.com");

// Pass the contact info to the user creation call
// This assumes userData is an already filled UserWS structure.
userData.setContact(contactInfo);
// Now create the user
api.createUser(userData);
```

Here, we pass the contact information in the user creation process, but you could as well pass this information in an `updateUser()` call when updating user data, or in a separate call to `updateUserContact()` if you wish to only update contact information and leave the user data as it is.

Custom Contact Fields

Two fields of the `ContactWS` structure deserve further analysis: `fieldNames` and `fieldValues`., which represent custom contact data. For example, let's say you need to store the user's website address. A quick look at the `ContactWS` structure shows there's no website field, so you would be left with the option of storing that info in another field not currently used, but **j**Billing offers a better alternative: custom contact fields (CCF).

To use CCF you will need to add it to the system through some configuration. This is out of scope for this document, but it boils down to two simple steps:

1. add the field to the the `contact_field_type` table:

```
insert into contact_field_type (id, entity_id, prompt_key,
data_type, customer_readonly)
values (432, 1, 'ccf.web_site', 'string', 1);
```

2. add the name of the field to the file `ApplicationResources.properties`:

```
ccf.web_site=Web site URL
```

Customizing your contact info is quite straightforward: you need to provide two arrays of strings. The first array, `fieldNames`, contains the IDs for each of the custom contact fields you wish to specify a value. The second array, `fieldValues`, holds the actual values for the fields.

When entering custom contact data, you need to initialize both arrays with data and add them to the `ContactWS` structure. A quick example explains how:

```
// The CCF id to set is 432, following the above example
public static final String CCF_WEBSITE = "432";


// Now we create two string arrays for the Ids and values.
String customContactIds[] = new String[1];
String customContactValues[] = new String[1];

// Put the values into the arrays.
// this holds the unique ID for the field.
customContactIds[0] = CONTACT_WEBSITE;
// this holds the actual value.
CustomContactValues[0] = “www.fifa.com”;

// Set the custom fields in the ContactWS structure.
// We assume contactData is an already created and filled ContactWS
structure.
contactData.setFieldNames(customContactIds);
contactData.setFieldValues(customContactValues);

// We now proceed with the contact data creation process.
```

We defined a constant that holds a unique identifier for the custom contact fields. It will have to be equal to the ID of the row in the `contact_field_type` table that defines this field. In this case, ID 432 will contain website information. Of course, this constant would ideally be located in a separate class or externalized to a properties file, to keep things simple we just declared it on the fly.

We then create two `String` arrays (each containing just one element) and assign the ID number and field value, respectively. We then assign these arrays to the `fieldNames` and `fieldValues` fields in the `ContactWS` structure. When the api call is performed, the custom data will be saved along with the rest of the contact data.

A fixed length array is of course not very intuitive, but you can easily convert these two arrays into a HashTable and use the ID value as an index to the actual data.

Updating and Deleting Users and Orders

There are many situations in which you would need to update the user's information, such as for example when the user changes passwords or address. It is also common

that the user subscribes to a service, or cancels a subscription. These type of events are reflected through purchase orders in **j**Billing, you will need a way to update orders programmatically. This section provides some insight into these operations.

Updating a user or order is easy enough. You just need to **query its current data, change the fields you need to update, and call the proper update method**. For example, let's say a user changes his password, the sequence to perform would be:

```
// Get the user information from jBilling
UserWS userData = api.getUserWS(userId);

// Change the password in the user data and submit it back to the system.
userData.setPassword(newPassword);
api.updateUser(userData);
```

Here we assume you already have the User ID handy after the used logged in, perhaps as a session variable.

There are similar functions to update orders (`updateOrder`), order lines (`updateOrderLine`), credit card information (`updateCreditCard`) and contact information (`updateUserContact`). They are used much in the same manner. You can of course populate the `UserWS` data from scratch, instead of querying it from **j**Billing, if you have all the necessary information handy.

You can also delete an user or order, if necessary. The methods are, as you have probably guessed, `deleteUser` and `deleteOrder`. You need only the User ID or Order ID to perform the deletion, not the entire `UserWS` or `OrderWS` structures. For example:

```
api.deleteUser(oldUser);
```

This will delete the user. Keep in mind that **j**Billing does not actually delete the user from the database, it just marks it as deleted. Of course, when a user gets deleted, so does the contact and credit card information that was associated to the account, and orders are deactivated (they will be no longer invoiced).

Deleting an order deletes the order lines associated with it, as well. When you query the status for a user that has been deleted (for example, in the login page), you'll get the “Deleted” status code as response.

The user's “main” order

jBilling introduces the concept of a main order for each user. This simply tells **j**Billing which order should be the target when executing automated operations such as recurring charges. “**j**Billing's User Guide” covers this concept in greater detail, here we're mainly interested in how the user's main order can be specified or modified by means of API calls.

The main order of a user can be found on the `UserWS` structure (returned by `getUserWS()`), in field `mainOrderId` (which contains the unique identifier of the order that acts as main order for this user). So, it is easy to access it with user API calls:

```
// Get the user information from jBilling
UserWS userData = api.getUserWS(userId);

// Read the main order ID into a variable and set
// another order (retrieved previously) as main.
Integer mainOrder = userData.getMainOrderId();
userData.setMainOrderId(newMainOrder);
api.updateUser(userData);
```

It is also possible to determine if an order is the current main order by examining the order's record (there's a property named `isCurrent` that indicates whether the order is the main order or not). When creating a new order, you can also set this property to “1” to tell **j**Billing that the order being created is to be considered the main order for its user:

```
OrderWS newOrder = new OrderWS();
// Set order parameters here ...
newOrder.setIsCurrent(Integer.valueOf(1));
// Create the order:
api.createOrder(newOrder);
```

A word on pricing

Pricing is the process that performs calculations to the order data to find the correct price for the order. While possibly an “quantity * price per item” scenario is sufficient, many companies have a much more complicated process to calculate prices. You could, for example, apply a discount for large orders, or have preferential rates for select customers or items.

jBilling solves this problem by providing a business rule-based mechanism for price formation. While the inner workings of the business rule engine lies outside the scope of this document (please refer to the “Extension Guide” for details on how to integrate Drools-based rules into **j**Billing), let's briefly describe how to interact with this component, and use it in your integration calls.

Pricing is usually based on the characteristics of the item (for example, “apply a 15% discount on item 'X' if quantity of item is above 15”) or order (“apply a 10% discount to any orders entered by user 'Y'”). There are situations, however, when the order or item data is not sufficient to describe the formation of price.

For example, consider a situation in which we have created a “click on a banner” item (as exemplified in the Trend example of the “Getting Started” guide) with ItemID = 2. We need to apply different prices depending on the country of origin of the customers clicking on the banner. Without rule-based pricing, the only solution is to create as many items as there are countries, and charge the right item when the server has decoded the country of origin of the visitor. By using rule-based pricing, you can create one single item and tailor the price this item will have according to external conditions.

To address these situations, **j**Billing introduces the concept of “pricing fields”. A pricing field is simply a value that is passed verbatim to the pricing engine, to be used if required during the price formation phase.

The pricing fields are useful in our example because the country of origin is known only by the server that is creating the order on **j**Billing, and only after the event has taken place. So, the only way we can specify variable data that alters the way rules behave at runtime is passing these values to the rule engine for evaluation.

Now, lets code the rules that set different prices for our “click on a banner” item according to the country of origin. To keep things simple, we're assuming here that the server that originates the call to **j**Billing passes the country as a string containing the ISO code of the country of origin.

```
rule 'pricing rule for the US'
when
       exists(PricingField( name == “countryFrom”, strValue == “US” ))
       $line : OrderLineDTO( itemId == 2 )  #the “click on a banner” item
then
       $line.setPrice(10.0F);  # price for US customers is 10$.
end

rule 'pricing rule for Canada'
when
       exists(PricingField( name == “countryFrom”, strValue == “CA” ))
       $line : OrderLineDTO( itemId = 2 )
then
       $line.setPrice(9.0F);  # price for canadian customers is 9$.
end
```

These rules can be read as: “apply a price of 10$ to order lines with Item 2 for users connecting from the US, and 9$ for order lines with Item 2 for users connecting from Canada”. We can now code the API client that passes customer country in a new PricingField variable with name “countryFrom”:

```
JbillingAPI api = JbillingAPIFactory.getAPI();
String country = “US”;  // your application code determines the origin
PricingField pf = new PricingField(“countryFrom”, country);
PricingField[] pfields = new PricingField[] { pf };
OrderWS order = new OrderWS();
// Initialize order parameters here...
order.setPricingFields(PricingField.setPricingFieldsValue(pfields));
// now, let's rate the order
OrderWS result = api.rateOrder(order);
```

This sample code creates a pricing field containing “countryFrom” = “US”, so, if run, all order lines containing Item 2 are assigned a price of 10$ in the output OrderWS structure.

A PricingField can hold one of a String, Date, Integer or Double value, a name that identifies the PricingField (and can be used as demonstrated above). For more details, see the PricingField structure definition below.

If you examine the structure of the OrderWS class, you'll see that its pricing fields are contained in a String, for serialization purposes. To simplify access to this data, the PricingField class offers utility methods to decode the serialized string into an array of PricingField values and encode an array of PricingField classes into a string. This of course can only be done if you're using the API, SOAP clients should encode the string by hand.

Encoding is actually very simple, pricing fields are separated by commas, and each element of the pricing field is separated from the others by colons, in the following order: “name:position:type:value” (name is a string value corresponding to the name of the pricing field, position is an integer indicating the ordering of pricing rules, type is one of “string”, “integer”, “double” or “date”, and value has the string formatted value for the field). Correct examples of encoded strings are: “countryFrom:1:string:US” or

“newPrice:1:double:5.5,oldPrice:1:double:6.0” (the second example has two pricing fields encoded, notice the comma separating each element).

In this manner, it is possible to quickly add pricing rules that do not apply to specific items or orders, but are rather applied based on external events (that translate to appropriate PricingField values) or other constraints not directly determined from the Order or Item data structures.

Keeping jBilling in sync with your data

While you can place many service calls to **j**Billing and expect it to update its information directly, there are times in which asynchronous events take place in either your system or in **j**Billing that change the data in one way or the other. For example, a scheduled payment could have been rejected by the processor, or new invoices have been generated in a specific date, or users could change their status due to payment events (they become overdue or suspended, etc.).

When it is your data that changes, it could be a simple matter of calling the appropriate update methods in the Client API to sync **j**Billing to your changes. When it is **j**Billing that changes its data, you'll need to resort to specific API calls or setup callback functions.

The Client API provides several query functions that can be used to update your data. Most of these functions have the “`get`” prefix. For example, if you keep a list of all users and their current status in your system, you could simply fire up an update job every night or so and call the `getUserWS()` method for each user in the list to update their data. This, however, is both time- and resource-consuming. It will work for small volumes of data, but it will not scale up.

Some API query methods return sets of results and not just the data that corresponds to one element. The `getUsersInStatus()` function returns a list of the User IDs of all users that are in a specific status, for example, `getUserInStatus(new Integer(1))` returns all users that are currently active. As another example, `getOrderByPeriod()` returns a list of all orders that have the same periodicity (in our previous example, `getOrderByPeriod(new Integer(2))` returns all orders that recur monthly).

Subscription status

The `UserWS` structure maintains a field named `subscriberStatusId` which contains the status code for subscribers. This status field is kept updated internally by **j**Billing to reflect the overall status of a user as a subscriber. Event such as a failed payment, or a successful payment, produce changes to this status. It is easy to retrieve the current user's status by calling `getUserWS()`.

Another important query method is `getUserTransitions()`. This method returns a list of all users that underwent a transition in their subscription status in a given time period. You can call this method periodically (daily, for example), to keep your application updated with who is subscribed and who is not to your services.

The method receives a pair of date values as arguments, indicating the starting and ending date of the period where transitions occurred. You can assign a null value to either or both of these parameters. The query **remembers the last time you called it**, so if you specify a null starting date, it will extract all transitions made from the moment you last called this function.

If the ending date is null, there's no upper limit to the transition dates (i.e., the upper limit is the current date). The query returns a set of `UserTransitionWS` structures that contains all transitions of subscription status registered in the given period of time.

Setting up Call Backs

As seen, the API provides a rich set of queries that allow your program to determine the status and information about most of the events your program needs to be informed about. You can therefore use these calls to update your program's data from **j**Billing, by creating recurring tasks that periodically query your **j**Billing installation and update your data accordingly.

Some situations, however, warrant a finer degree of control. User status changes are the most hard to manage with API calls, since they occur in a rather asynchronous manner, and checking the status for all your users could prove a resource hog. Also, you would ideally need to know of the change quickly, so that you can suspend services to a user that has not paid your invoices.

> *Note that, although related, the 'ageing status' is not the same as the 'subscription status'. The first one is related to the ageing process, while the second is not, it is only affected by recurring orders and payments. Call backs are only done for changes in the 'ageing status'.*

Ideally, you would need a way to receive a notification when a user status changes, so that you proceed to update only the user that has been affected by the change. This is where the HTTP Call Back (HCB) interface comes to the rescue.

The HBC feature of **j**Billing is capable of performing a web call to your application whenever a user status changes. HBC will notify your application what user incurred in the change, the old and new status, the user's login name, and whether the new status allows the user to login or not. This information can come handy to update your data on the fly.

HBC performs a normal web request, passing the information about the status change to your application via POST parameters. So, your application needs to publish a web page that takes care of receiving the notification, get the parameters from the request form and act accordingly.

> *The drawback of this method is that, if the page that listens to the call backs is not responding, the call will fail and your application will be out-of-sync.*

Setting up HBC requires you to enter **j**Billing's User Interface as an administrator, and set the callback URL in the “Ageing” screen. When a valid URL is placed in this field, **j**Billing will POST a request to that URL whenever a user changes status. This callback can be redirected to a task in your program that registers the status change in your data structures.

The POST callback performed by **j**Billing provides the called routine with the following information pertaining the event that has occurred:

`cmd`: A string that identifies **j**Billing's call. It is always set to “`ageing_update`” by **j**Billing, and allows you to uniquely identify a **j**Billing call (in case the callback's url is used for other purposes).

`user_id`: The User ID of the user that is undergoing a status change in **j**Billing. You can use this ID to identify the user directly, or otherwise query **j**Billing via the API to obtain the data for the user, should you need further information.

`login_name`: The user name of the user that is undergoing a status change.

`from_status`: The status the user is transitioning from. This is the status the user had before the transition took place.

`to_status`: The status the user is transitioning to. This is the status the user has once the transition took place. Subsequent calls to `getUser()` in the API will return this status code.

can_login: Indicates whether the user's new status allows him to have access to the website's member's area or not. A value of “1” indicates the user can login to your system, a “0” means he cannot. This information can be safely ignored, if you check the user's status during the login phase, as explained in a previous chapter.

A quick example of a callback routine handler is given below. This routine simply prints a status change notice to the console, but ideally your application would use this data in a more useful manner. This code constitutes the `doPost()` method of a Java servlet class that services the callback inside your application:

```
public void doPost(HttpServletRequest request, HttpServletResponse resp)
            throws ServletException, IOException {
      // The doPost() method of the servlet class. This services all callbacks
      // from jBilling into your app.
      String userName;
      Integer userId;
      Integer fromStatus;
      Integer toStatus;
      boolean update = false;
      // Test whether this is a jBilling call by checking the "cmd" parameter.
      if (request.getParameter("cmd").equals("ageing_update")) {
            userName = request.getParameter("login_name");
            try {
                  userId = new Integer(request.getParameter("user_id"));
                  fromStatus =
                     new Integer(request.getParameter("from_status");
                  toStatus = new Integer(request.getParameter("to_status");
            } catch (NumberFormatException e) {
                  // If the values passed were not numbers do not
                  // perform the update.
                  update = false;
            }
            if (update) {
                  // Here we just print out a notice. Your app should
                  // do something more sensible, such as updating your data.
                  System.out.println("User " + userName +
                        " has changed status!");
            }
      }
      else {
            // This was not a jBilling call. You can process it normally
            // or return an error, as needed.
      }
```

```
}
```

Conclusion

This document provided a hands on guide to using **j**Billing's integration services. While the Client API provides much more service calls and features than what was covered here, we hope this tutorial has provided some useful insight into how to put the integration API to good use in your programs.

The sections that follow cover all service calls and data structures provided by the Client API in more depth, and explain what parameters they require and what information you get in return, where applicable. Please refer to these sections for further details.

The **j**Billing integration layer provides a rich set of features that allow your application to seamlessly integrate the billing process into your business logic. The Client API provides a flexible and intuitive layer of abstraction that further simplifies the integration process and makes your application independent of the specific technology that takes care of communicating with the **j**Billing server. The Client API is easily available to any Java application.

The SOAP layer provides further support for non-Java applications that nevertheless need to integrate with **j**Billing. Most modern web programming frameworks provide support for SOAP services, which greatly expands the set of programming languages and production environments that **j**Billing can integrate with.

Chapter 4
Client API Reference

Overview

The following contains a detailed reference of the **j**Billing Client API. All service calls provided by the API are covered here.

The chapter is divided into sections. Each covers the functions pertinent to a specific functional area of **j**Billing: user management, orders, items, etc. Each section is further divided into two subsections, one covering the service calls and one that details all data structures used as parameters or returned by the service calls.

All data contained in the Client API data structures can be accessed by means of the "getter" and "setter" methods of each property (i.e., `getFaxNumber()` and `setFaxNumber()` provide access to the `faxNumber` property). When describing data structures, this guide always provides the property name.

Aside from these data structures, the Client API makes use of several standard Java data types (notably, `java.lang.String` and `java.lang.Integer`), whose use and interfaces are not covered here.

This reference is equally useful for those planning to use the web services directly, or for those that will be using the **j**Billing Client API. For non-Java languages, the data structures will look slightly different: they are described in the WSDL descriptor and how that is translated to each language varies. For those using the **j**Billing Client API from a Java application, the data structures can be used directly as classes.

For a finer coverage of the SOAP support, refer to the WSDL file or use a SOAP introspection tool (such as **SoapUI**). All the SOAP methods, however, share the same name, parameters and functionality as their API counterparts.

User Management Calls

The user management calls provide methods for creating, modifying and deleting users, retrieve user information, user authentication, status and contact management.

Data Structures

The following data structures are used by user management service calls.

UserWS

This data structure holds information about a **j**Billing user.

Property Name	Type	Description
`balanceType`	`Integer`	The type of dynamic balance for this user. Refer to Appendix A for acceptable values.
`blacklistMatches`	`String[]`	Lists any blacklist matches for this user. See the "jBilling User Guide" for more information on blacklists.
`childIds`	`Integer[]`	The identifiers of any sub-accounts for this user.

contact	ContactWS	The primary contact information for this user.
createDateTime	Date	Creation date of this data record.
creditCard	CreditCardDTO	Credit card information for this user. Not required for the user creation process.
creditLimit	Double	The credit limit. Only valid if balanceType is of credit limit type.
currencyId	Integer	Contains the currency code for this user. Refer to Appendix A for acceptable values.
deleted	Integer	If the record has been deleted, this field contains “1”, otherwise it contains “0”. Note that deletion cannot be carried out by simply setting a “1” in this field.
dynamicBalance	Double	The dynamic balance. If balanceType is credit limit, this represents the amount of credit used on the account. If balanceType is pre paid, this represents the pre paid balance remaining.
failedAttempts	Integer	Number of login attempts that have been failed by this user (i.e., the user has entered the wrong password).
invoiceChild	Boolean	“true” if this is a sub-account (child of a parent account), but this user will still receive invoices.
isParent	Boolean	“true” if this record is a “parent” user. A parent user can have sub-accounts (children).
language	String	Name of the language (i.e. “English”).
languageId	Integer	Contains the preferred language code for this user. Refer to Appendix A for acceptable values.
lastLogin	Date	Date of the last login performed by this user.
lastStatusChange	Date	Date of the last status change incurred by this user.

mainRoleId	Integer	The level of privilege granted to the user when logged into the system. See Appendix A for acceptable values.
mainOrderId	Integer	The id of the main order for this customer. This is the order that sets the customer's billing cycle for the management of 'current' orders'. Current orders gather usage. See section: The user's “main” order for a description and examples of use of main orders. You can set this value to an order and call 'updateUser' to set a new order as the main order.
owingBalance	Double	A real-time calculated owing balance.
parentId	Integer	If the user belongs to a parent record, this field contains the identifier of the parent record.
partnerId	Integer	Identifier of the partner this user belongs to.
password	String	Authenticates the user's identity during login. This could be meaningless if the password is encrypted.
role	String	The name of the role (i.e. “Clerk” or “Customer”).
status	String	Name of the current status (i.e. “Suspended” or “Active”).
statusId	Integer	Current status of the user. See Appendix A for acceptable values.
subscriberStatusId	Integer	Subscriber status for this user. See Appendix A for acceptable values.
id	Integer	A unique number that identifies the customer
userIdBlacklisted	Boolean	“true” if the user id is blacklisted. See the “jBilling User Guide” for more information on blacklists.
userName	String	Identifies the user during login

autoRecharge	String	Amount by which the customer's account will be auto-recharged when depleted (the amount can be handled as a BigDecimal Java type via the “setAutoRechargeAsDecimal()” and “getAutoRechargeAsDecimal()” methods.

CreditCardDTO

This data structure holds information about credit cards.

Property Name	Type	Description
deleted	Integer	If the record has been deleted, this field contains “1”, otherwise it contains “0”. Note that deletion cannot be carried out by simply setting a “1” in this field.
expiry	Date	Expiration date of the credit card. Usually, card expiration dates are expressed in month/year form, such as “05/11” or “May 2011”. This field contains the last day the card is valid, in this example, “05/31/2011”.
id	Integer	Unique identifier for this record.
name	String	Credit card owner's name. This is the name that appears physically on the credit card.
number	String	Credit card number. Usually, a 16 digit number.
securityCode	Integer	CCV (Credit Card Verification) code of the credit card.
type	Integer	Credit Card type. See Appendix A for acceptable values.

ContactWS

This data structure holds contact information.

Property Name	Type	Description
address1	String	First line for the address.
address2	String	Second line for the address.
city	String	City of this contact.

countryCode	String	Country code for this contact (Appendix A contains a list of acceptable country codes).
createDate	Date	Date this contact record was first created.
deleted	Integer	If the record has been deleted, this field contains “1”, otherwise it contains “0”. Note that deletion cannot be carried out by simply setting a “1” in this field.
email	String	E-Mail address of this contact.
faxAreaCode	Integer	Area Code for the fax number, if any.
faxCountryCode	Integer	Country Code for the fax number, if any.
faxNumber	String	Fax number.
firstName	String	First name for the contact.
id	Integer	Unique identifier of this contact.
include	Integer	“1”, if this contact is marked as included in notifications.
initial	String	Middle name initials, if any.
lastName	String	Contact's surname.
organizationName	String	Name of the organization the contact belongs to.
phoneAreaCode	Integer	Phone number Area Code.
phoneCountryCode	Integer	Phone Number Country Code.
phoneNumber	String	Phone Number.
postalCode	String	ZIP Code for the contact's address.
stateProvince	String	State or Province of the contact's address.
title	String	Title for the contact, such as “Mr.” or “Dr.”.

`fieldNames`	`String[]`	The name of each of the customized contact fields contained in this contact record.
`fieldValues`	`String[]`	The values for each of the customized contact fields contained in this contact record.

UserTransitionResponseWS

This data structure holds information about subscriber status transitions.

Property Name	Type	Description
`fromStatusId`	`Integer`	Status of the subscription before the transition took place. See Appendix A for acceptable values.
`id`	`Integer`	Unique identifier for the transition record.
`toStatusId`	`Integer`	Status of the subscription after the transition took place.
`transitionDate`	`Date`	Date and time the transition took place.
`userId`	`Integer`	Identifies the user account that suffered the subscription status change.

Methods

authenticate

This method provides user authentication. It is a shorthand for calling `getUserId()` and `getUserWS()` in sequence, while also checking the password in the process.

Input Parameters

`username` (String): the user's login name.

`password` (String): the user's password.

Return Value

`Integer`:

- 0: The user was successfully authenticated and his current status allows him entrance to the system.
- 1: Invalid credentials. The user name or password are not correct.
- 2: Locked account: The user name or password are incorrect, and this is the last allowed login attempt. From now on, the account is locked.

- 3: Password expired: The credentials are valid, but the password is expired. The user needs to change it before logging in.

If the input data is null or missing, this method generates a `JbillingAPIException` that signals the problem.

createUser

This method creates a new user.

Input Parameters

`newUser` (UserWS): the user data that will be used to create the new user record. The `username` field of this structure must have a valid and unused user name string. The `userId` field can be set to `-1` since the unique identifier has not been generated yet (**j**Billing generates it during this call).

If the contact or credit card information are present, they will be created for the new user as well.

Return Value

`Integer`: If the user has been successfully created, the return value is the newly created user's ID. If the user name has already been used by another user record, it returns `null`. If the input data is null or missing, this method generates a `JbillingAPIException` that signals the problem.

deleteUser

Deletes an existing user. It will only mark the user as deleted, the record will remain in the database. It will do the same for all the orders, but will leave the invoices and payment untouched.

Input Parameters

`userId` (Integer): the **j**Billing identifier for the user that is being deleted. This id is retrieved either from your application's data or by a previous call to `getUserId()`.

Return Value

None. If the `userId` provided is null or inexistent, a `JbillingAPIException` is generated.

updateUserContact

Updates the user contact information.

Input Parameters

`userId` (Integer): the identifier of the user whose contact information is being updated.

`typeId` (Integer): the contact's type. This is typically a '2' for the primary contact type in an installation with only one company. You would need to query the table `contact_type` to find out the IDs of all the type available in your system.

`contact` (`ContactWS`): Maintains the contact data that is being updated.

Return Value

None. If the parameters provided are null or inexistent, a `JbillingAPIException` is generated.

updateUser

This method updates the user account information in **j**Billing. This includes the contact and credit card information.

Input Parameters

`user` (`UserWS`): the user's data. Can be obtained by a previous call to `getUserWS()` or generated directly by your application (be careful, if you don't put values into fields that haven't changed, you'll loose those values, so **it is best to first retrieve the data record with `getUserWS()`** and just change the required fields and resubmit the `UserWS` structure). The supplied `UserWS` structure must contain a valid user identification number in its `id` field.

Three fields of the `UserWS` input parameter can be null. If the following fields are null, they will simply be ignored: `password`, `creditCard` and `contact`. If you do not want those fields to be updated, simply set their value to null.

This is useful on occasions where the jBilling configuration encrypts or does not make certain values available. The password and credit card are among them. If that is the case, a call to `getUserWS()` will return values for those fields that won't pass the validations of a subsequent call to `updateUser`.

Return Value

None. If the user identifier supplied is incorrect or the parameter is null or invalid, this method throws a `JbillingAPIException` to signal the problem.

getUserWS

This method returns the user data contained in **j**Billing.

Input Parameters

`userId` (`Integer`): the identifier for the user whose account data is being retrieved.

Return Value

`UserWS`: The account information, or null if the supplied `userId` is not assigned to an existing user. If the parameter is null or inexistent, a `JbillingAPIException` is generated.

getUserContactsWS

Returns the contact information for the user. A user can have several contact fields assigned to it (one primary contact plus any number of secondary contacts). This function returns all known contacts for the given user.

Input Parameters

`userId` (`Integer`): The identifier for the user whose contact information is being retrieved.

Return Value

`ContactWS[]`: An array of `ContactWS` structures containing all known contact records assigned to the user given as input. If the input parameter is null or inexistent, a `JbillingAPIException` is thrown.

getUserId

Returns the unique identifier associated to a user in **j**Billing. This method is meaningful specially during login, where you use it to retrieve the user's ID from the supplied user name.

Input Parameters

`username` (`String`): the user's login name.

Return Value

`Integer`: The user's identification number. If the parameter provided is null or there isn't a user with the user name give as a parameter, a `JbillingAPIException` is thrown..

getUsersInStatus

Returns a list of all users in a given status. Useful for making synchronization calls between your application and jBilling, or to otherwise process this information (for example, for a report).

Input Parameters

`statusId` (Integer): Indicates what status will be used for extraction. See Appendix A for a list of acceptable status codes.

Return Value

`Integer[]`: An array containing the identifiers of all users that were in the status given as input in the moment the method was called (status transitions could occur after the method was called, so this information should not be used to grant access to the site or to substitute the `authorize()` or `getUserWS()` methods for authentication).

getUsersNotInStatus

The opposite of `getUsersInStatus()`, this method provides a list of all users that are not currently in the given status.

Input Parameters

`statusId` (`Integer`): Indicates what status will be used for extraction. See Appendix A for a list of acceptable status codes.

Return Value

`Integer[]`: An array containing the identifiers of all users that were in any status different to the one given as input in the moment the method was called (status transitions could occur after the method was called, so this information should not be used to grant access to the site or to substitute the `authorize()` or `getUserWS()` methods for authentication).

getUsersByCustomField

Returns all users that share a common custom contact field with a specific value. See the section Custom Contact Fields for an explanation on how to use these fields.

Input Parameters

`typeId` (`Integer`): Identifier of the custom contact field ID.

`value` (`String`): The value that will be tested for all users in order to determine which users should be extracted.

Return Value

`Integer[]`: An array of all users that have the indicated custom field set to the specified value.

getUsersByCreditCard

Returns all users in the system that share a common credit card number. Could be useful to determine if other users in the system have already registered a given credit card number.

Input Parameters

`number` (`String`): The credit card number used to filter users.

Return Value

`Integer[]`: An array of all users that have a credit card registered with the number passed as parameter.

updateCreditCard

Updates credit card information for a user. A user can have more than one credit card record. After calling this method, the user will only have this credit card. This method will remove all the existing credit cards and then assign the one given as a parameter.

Input Parameters

`userId` (`Integer`): The identifier of the user whose credit card information is being updated.

`creditCard` (`CreditCardDTO`): the credit card's data with the updated values.

Return Value

None. If the parameters provided are null or incorrect, a `JbillingAPIException` is thrown.

getUserTransitions

Returns a list of subscription transitions that have taken place in a given period of time. See the section Subscription status for an explanation on how this function can be used.

Input Parameters

`from` (`Date`): Starting date of the extraction period. Can be `null` (in which case, the extraction period starts from the last time this function was called, or from the first transition if the function has not yet been called).

`to` (`Date`): Ending date of the extraction period. Can be `null` (in which case the extraction has no upper limit, i.e. It extracts all records that have happened to this moment).

Return Value

`UserTransitionResponseWS[]`: Array of transition records containing the transition information for all registered changes in subscription status during the specified period.

getUserTransitionsAfterId

Returns a list of subscription transitions that have taken place after a specific transition (whose id is given as a parameter) has taken place. See the section Subscription status for an explanation on how this function can be used.

Input Parameters

`id` (`Integer`): Identifier of the transition marking the start of the extraction. Can be `0` (in which case the extraction has no lower limit, i.e. it extracts all transitions recorded).

Return Value

`UserTransitionResponseWS[]`: Array of transition records containing the transition information for all registered changes in subscription status that have taken place after the specified transition.

isUserSubscribedTo

Returns the quantity of the given item the user is subscribed to, that is, has recurring orders for.

Input Parameters

`userId` (`Integer`): The identifier of the user whose subscription is being checked.

`itemId` (`Integer`): The identifier of the item the user is being checked for subscription to.

Return Value

`Double`: The quantity of the item the user is subscribed to.

Order Management Calls

The order management calls provide methods for entering orders and querying information about orders.

Data Structures

OrderWS

This data structure contains the data for an order.

Property Name	Type	Description
`activeSince`	`Date`	The point in time when this order will start being active, reflecting when the customer will be invoiced for the items included. A null value indicates that the order was active at creation time (see field createDate).
`activeUntil`	`Date`	The point in time when this order stops being active. After this date, the order will stop generating new invoices, indicating that the services included in this order should stop being delivered to the customer. A `null` value would specify an open-ended order. Such order never expires; it is considered on-going and will require explicit cancellation for it to stop generating invoices.
`anticipatePeriods`	`Integer`	How many periods in advance the order should invoice for. Leave with a '0' unless you have configured the system to work with anticipated periods.
`billingTypeId`	`Integer`	Indicates if this order is to be paid for before or after the service is provided. Pre-paid orders are invoiced in advance to the customer, while post-paid are only invoiced once the goods or services included in the order have been delivered. “1” means “pre-paid”, while “2” means “post-paid”.
`billingTypeStr`	`String`	(read only).This will show the word that represents the billing type. It is ignored when you submit the object.

createDate	Date	(read only). A time stamp with the date and time when this order was originally created.
createdBy	Integer	The id of the user that has created this order.
currencyId	Integer	Currency code. Refer to Appendix A for a list of acceptable values.
deleted	Integer	A flag that indicates if this record is logically deleted in the database. This allows for 'undo' of deletions. Valid values are 0 – the record is not deleted 1 – the record is considered deleted.
dfFm	Integer	Only used for specific Italian business rules.
dueDateUnitId	Integer	If this order has a specified due date, this will the the units (days, months, years). See Appendix A for valid values.
dueDateValue	Integer	How many units will be used for the due date.
id	Integer	A unique number that identifies this record.
lastNotified	Date	When the order has expiration notification, this field tells when the last one was sent.
nextBillableDay	Date	The date when this order should generate a new invoice. Meaning that until that date (and excluding that date), the customer has been invoiced for the service included in this order.
notes	String	A free text field for your notes.
notesInInvoice	Integer	"1" if this order's notes will be included in the invoice, or "0" if not.
notificationStep	Integer	What step has been completed in the order notifications.
notify	Integer	If this order will generate notification as the 'active since' date approaches.
orderLines	OrderLineWS[]	The order lines belonging to this order. These objects will specify the items included in this

		order with their prices and quantities. See the OrderLineWS specification for more information.
ownInvoice	Integer	A flag to indicate if this order should generate an invoice on its own. The default behavior is that many orders can generate one invoice.
period	Integer	Indicates the periodicity of this order. In other words, how often this order will generate an invoice. Examples of periods are: one time, monthly, weekly, etc. Period codes can be seen in **j**Billing's User Interface under “Orders -> Periods”.
periodStr	String	(read only). The description of the order period.
statusId	Integer	An order has to be on status ‘Active’ in order to generate invoices. An order usually starts in active status, and only goes to suspended or finished when the customer fails to make the required payments. The steps and actions taken due to late payments are part of the ageing process. See Appendix A for a list of acceptable order status codes.
statusStr	String	(Read only) The description of the current order status.
userId	Integer	This order belongs to the user specified by this field.
pricingFields	String	An array of pricing fields encoded as a String. To encode, use the PricingField.setPricingFieldsValue() static method, which takes as parameter an array of PricingField structures and returns the encoded string. To decode, use the PricingField.getPricingFieldsValue() static method, which takes as parameter an encoded string and returns an array of PricingField structures. Pricing fields are descriptors that provide further information to the pricing engine and aid in forming the price for the order itself. See section “A word on pricing” for a more detailed explanation of the use of pricing fields.
cycleStarts	Date	The date at which the billable cycle starts. A

		recurring order will generate invoices at the day of the month specified in this field. The date provided in this field must comply with the following condition: cycleStarts ≥ createDate For example, for an order created on 01/13/2009, setting this field to 01/28/2009 will make the system generate the invoices the 28th of each month (for orders with monthly periodicity). If set to null, the system will calculate the appropriate billing date based on the date of creation of the order and its activity period.
`isCurrent`	`Integer`	Indicates whether the order described in this structure is the current main subscription. See section: The user's “main” order for a description and examples of usage of this field. Acceptable values are: 0 – The order is not the current main order. 1 – The order is the current main order.
`timeUnitStr`	`String`	(Read only) The description of the time unit used for billable periods.

OrderLineWS

This data structure maintains the information for a single line of an order.

Property Name	Type	Description
`amount`	`Float`	The total amount of this line. Usually, this field should respond to the formula `price * quantity`. This amount will be the one added to calculate the purchase order total. The currency of this field is the one specified in its parent order. The amount can be also set and obtained as a Java BigDecimal, using the “getAmountAsDecimal()” and “setAmountAsDecimal()” methods.
`createDateTime`	`Date`	A time stamp applied when this record is created.
`deleted`	`Integer`	A flag that indicates if this record is logically deleted in the database. This

		allows for ‘undo’ of deletions. Valid values are 0 – the record is not deleted 1 – the record is considered deleted.
description	String	A descriptive text for the services being included. This usually copies the description of the item related to this line.
editable	Boolean	Indicates whether this order line is editable or not (i.e., it cannot be submitted for update).
id	Integer	A unique number that identifies this record.
itemDto	ItemDTOEx	Contains information of the item this order line refers to.
itemId	Integer	The id of the item associated with this line, or null if this line is not directly related to an item. It is consider a good practice to have all order lines related to an item. This allows for better reporting.
isCurrent	Integer	Indicates whether this order is the user's main order (i.e., the “default” order). To set an order as the main order for a user, set this flag to “1” and update the order via API calls. Acceptable values: 0 – This order is not the current order. 1 – This order is the current order for its associated user.
price	String	The price of one item, or null if there is no related item. Can also be manipulated as a Java BigDecimal using the “getPriceAsDecimal()” and “setPriceAsDecimal()” methods.
priceStr	String	The price of the item as a string.
provisioningRequestId	String	The provisioning request UUID for this order line, if it exists.
provisioningStatusId	Integer	The provisioning status id for this order

		line. See Appendix A for valid values.
quantity	String	The quantity of the items included in the line, or null, if a quantity doesn't apply. It can also be handled using the "getQuantityAsDecimal()" and "setQuantityAsDecimal()" methods.
typeId	Integer	An order line usually has items. However, some lines are used for additional charges, like taxes. See Appendix A for a list of acceptable order line type codes.
useItem	Boolean	If true, when submitted, this line will take the price and description from the item. This means that you would not need to give a price and description for the line. Instead, you only provide the id of the item. See the createOrder section for details.
orderId	Integer	Identifier of the order that contains this order line.

CreateResponseWS

Property Name	Type	Description
invoiceId	Integer	Identifier of the invoice that was generated.
orderId	Integer	Identifier of the order that was created.
paymentId	Integer	Identifier of the payment that was generated to pay the invoice.
paymentResult	PaymentAuthorizationDTOEx	Outcome of the payment operation.
userId	Integer	Identifier of the new user created and for which the order, invoice and payment were created.

PricingField

This data structure describes heterogeneous data that will be passed to the rules engine in order to calculate prices and/or flag specific conditions that affect pricing:

Property Name	Type	Description
name	String	Identifier of the pricing field.
position	Integer	
strValue	String	Optional string value of the pricing field, which can be used as necessary by the rules (for example, use the string value as a new description for the item).
intValue	Integer	Optional integer value of the pricing field, which can be used as necessary by the rules.
doubleValue	Double	Optional double value of the pricing field. Provided for backwards compatibility only, API users are advised to use “decimalValue” instead.
dateValue	Date	Optional date value of the pricing field, which can be used as necessary by the rules (for example, to set the activeUntil date of the order).
calendarValue	Calendar	Optional date value of the pricing field expressed with a Calendar object. It can be used by the business rules as necessary.
decimalValue	BigDecimal	Optional decimal value of the pricing field, which can be used as necessary by the rules (for example, to set a different price or apply as discount).
floatValue	Float	Optional float value for the pricing field. Provided for backwards compatibility only, API users are advised to use “decimalValue” instead.

Methods

create

This method facilitates the sign-up of a new customer. It will create the new user, and all the related objects: purchase order, invoice and payment.

The payment will only be created and processed if a credit card is included in the `UserWS` parameter. Therefore, the caller will always get values for the fields `userId`, `orderId` and `invoiceId`, but `paymentId` and `paymentResult` would remain null if not credit card information is found in the `UserWS` parameter.

The payment will be submitted for immediate, real-time processing to the payment processor. The value in `paymentResult` will reflect the response of the payment processor. An email notification will also be sent to the customer with the result of this transaction.

To summarize, these are the tasks involved in this method:

1. Creation of a new user
2. Creation of a new purchase order for this new user.
3. Generation of an invoice based on the purchase order.
4. Real-time processing of a payment for the invoice through a payment processor.
5. Notification via email of the result of this payment to the customer (if applicable, see the Notifications documentation for more information).

Steps 4 and 5 only take place if a credit card is present in the UserWS parameter.

See the description of the `createUser` and `createOrder` methods for more information.

Input Parameters

`user` (`UserWS`): the account information for the user that is being created.

`order` (`OrderWS`): The purchase order information to assign to the new user.The `userId` of this object can be null and will be ignored, since this value will be assigned automatically with the `userId` of the newly created customer.

Return Value

`CreateResponseWS`: A data structure containing data about all the objects created.

createOrderPreAuthorize

This method creates an order. Then it submits pre-authorization payment request to a payment gateway. The result of this request is returned as a `PaymentAuthorizationDTOEx` object. Regardless of the pre-authorization result, the order remains in the system after this call.

This method is typically called with an order whose 'active since' is set to a future date: this would represent a 'free trial', where the paying period starts on the 'active since' date, but you need to verify the paying capabilities of the potential customer.

The billing process will generate the invoice at the 'active since' date, and then 'capture' the pre-authorization for the first payment. This guarantees that the payment will be successful, since the payment gateway had already pre-authorized it.

Input Parameters

`order` (`OrderWS`): data for the order whose payment data is to be validated.

Return Value

`PaymentAuthorizationDTOEx`: Data structure containing the outcome of the payment verification process. If you need to know the ID of the new order, you will have to call '`getLatestOrder`'.

createOrder

Creates an order. When creating an order, you can indicate that the information of an order line should be fetched from the properties of an existing item. This is done through the field `OrderLineWS.useItem`. If this flag is set to 'true', then the price of the order line will be the price of the item designated in `OrderLineWS.itemId.` The behavior of some fields change depending on this flag:

Field	`useItems` is true	`useItems` is false
`itemId`	This field is required.	It is not required to specify an item id, but it is considered a good practice.
`description`	If left blank, the description of the item will be used. Otherwise, the description provided in this field	This field is required.
`amount`	This field will be calculated by the system, using the formula quantity * price. Any value in this field will be overwritten by this calculation, and therefore ignored.	This field is required.
`quantity`	This field is required, and has to be grater than 0	This field is optional.
`price`	The price will be taken from the price that the item has. This follows the item's pricing rules: multiple currencies, special customer prices, etc.	This field is optional.

Input Parameters

`order` (`OrderWS`): the order data.

Return Value

`Integer`: Identifier for the newly created order, if any, or `null` if the order data supplied was incorrect or insufficient.

createOrderAndInvoice

Creates a new order and contextually generates the corresponding invoice. Typically, you only create the order and then let the billing process take care of the invoice generation.

However, at times you might want to subscribe an **existing** user (if the user does not exist, then it's better to call '`create`') and immediately process the payment. In that case, you will call this method and then call '`payInvoice`' using the return value of this method.

If you need to generate an invoice based in **many** orders, you will need to call '`createOrder`' many times, followed by '`createInvoice`'.

Input Parameters

`order` (`OrderWS`): The order data.

Return Value

`Integer`: Identifier for the newly created invoice, if any, or `null` if the order data supplied was incorrect, insufficient or with attributes that do not generate an invoice (for example, an active since in the future). If you need to retrieve the ID of the order generated by a call to this method, call '`getLatestOrder`'.

updateOrder

Updates an order's data. Calling this method will modify an existing purchase order. Since this method updates all the order fields and all the order lines, normally the following steps are followed:

1. A call to `get` to retrieve the current information
2. Modify the fields to update. Add, update or remove order lines.
3. Call update

The existing order lines will be deleted, and new ones will be created with those provided in the `OrderWS` object passed as a parameter. In the end, the order identified by the field `id` will look exactly the same as the parameter passed.

The flag `useItem` present in each order line (`OrderLineWS`) works the same way when updating an order as it does when creating one. This is something to consider when you want to add order lines to an existing purchase order. You could also use this flag when updating an order line, but is not as common. Keep in mind that when your retrieve an order from the system, all its order lines will be having the flag `useItem` equal to `false`, regardless on how the were created.

Input Parameters

`order` (`OrderWS`): the updated order data. This can be either obtained from a previous call to getOrder() or created directly by your application, although the latter is unadvised (if you do not fill a field, its previous content is lost, so the advised procedure is to first retrieve the data to update, make the changes needed and resubmit that same data back).

Return Value

None. If the order information provided is invalid, a `JbillingAPIException` is generated.

getOrder

Returns the order data for a specific order.

Input Parameters

`orderId` (Integer): Unique identifier for the order.

Return Value

`OrderWS`: The order information, or an exception if the supplied order ID is invalid or if the order ID belongs to a deleted order. The object will have all the related order lines.

rateOrder

Performs pricing calculations on an order as if it was inserted, but does not actually create the order. Useful if you need to have immediate feedback on the pricing applied to a specific customer or item combinations, specially if you use rule-driven pricing (see the “Extensions Guide” for more information on rule-based pricing).

Input Parameters

order (OrderWS): holds the data for the order being rated.

Return value

OrderWS: The data for the order, as generated by the pricing engine. Mostly, it holds the data you passed as input, except for the price and amount fields, that should contain proper values calculated for the order according to pricing rules.

rateOrders

Performs pricing calculations on a set of orders as if they were inserted, but does not actually create the orders. Useful if you need to have immediate feedback on the pricing applied to a specific customer or item combinations, specially if you use rule-driven pricing (see the “Extensions Guide” for more information on rule-based pricing).

Input Parameters

orders (OrderWS[]): holds the data for the orders you wish to rate.

Return value

OrderWS[]: The data for each of the orders passed in the input, as generated by the pricing engine. Mostly, it holds the data you passed as input, except for the price and amount fields, that should contain proper values calculated for the orders according to pricing rules.

getOrderByPeriod

Returns a list of all orders of a specific periodicity for a given user. The method only returns the order's IDs, so subsequent calls to '`getOrder`' are necessary if you need the objects.

Input Parameters

`userId` (`Integer`): the user for which the extraction is desired.

`periodId` (`Integer`): Identifier for the period type. This value can be obtained from the **j**Billing User Interface under “Orders -> Periods” or from your billing administrator.

Return Value

`Integer[]`: Array containing the identifiers for the orders that respond to the input parameters.

getOrderLine

Retrieves a specific order line by supplying its identifier.

Input Parameters

`orderLineId` (`Integer`): unique identifier of the desired order line. You would know about this ID by first getting a complete order.

Return Value

`OrderLineWS`: the order line's information.

updateOrderLine

Updates the order line with the supplied information. This can be used to also delete an order line. If the 'quantity' field of the order line is set to '0', the order line is not updated, it is instead removed.

Input Parameters

`line` (`OrderLineWS`): the updated order line data to be stored in **j**Billing.

Return Value

None. If the provided order line data is invalid or does not correspond to an existing order line, the method generates a `JbillingAPIException`.

getLatestOrder

Retrieves the latest order created for a given user account. This is true unless the order has been deleted. This method will filter out any deleted order.

Input Parameters

`userId` (`Integer`): the user account's unique identifier.

Return Value

`OrderWS`: The data for the latest order inserted for the user, or `null` if the user has not yet been assigned an order or all the orders have been deleted.

getLastOrders

The ids of the last *n* purchase orders belonging to the user id given as a first parameter will be returned as an array. The first element of the array will be the latest purchase order. The next element will be the previous purchase order and so on. Subsequent calls to `getOrder` are necessary to retrieve the related `OrderWS` objects. The caller should check if the purchase order is not deleted by verifying that that `deleted == 0`. If the customer does not have any purchase orders, an empty array is returned.

Input Parameters

`userId` (`Integer`): the user account for which the extraction is to be performed.

`number` (`Integer`): the maximum number of orders that should be extracted.

Return Value

`Integer[]`: Array containing the order identifiers for the most recent orders created for this user. This method does not differentiate between deleted and non-deleted orders.

deleteOrder

Calling this method will mark a purchase order record as deleted, along with all its order lines.

Input Parameters

`orderId` (`Integer`): the unique identifier of the order that is to be deleted.

Return Value

None. If the order identifier provided as input is null or does not correspond to an existing order, the method throws a `JbillingAPIException`.

getCurrentOrder

Returns the mediation current one-time order for this user for the given date. See the “Telecom Guide” for more information about the mediation module and current one-time orders.

Input Parameters

`userId` (`Integer`): The identifier of the user whose current one-time order is being returned.

`date` (`Date`): The date that determines which current one-time order should be returned. The date will fall within the order's active period.

Return Value

`OrderWS`: The current one-time order for the user for the given date. If the user has no mediation main-subscription order, `null` is returned.

updateCurrentOrder

Allows the current one-time order to be updated with an event for this user for the given date. This method is used for real-time mediation of events. See the “Telecom Guide” for more information about the mediation module, mediation events and current one-time orders.

You can pass an array of order lines or an array of pricing fields. If you pass the order lines, then the mediation process will not be called. There is not need for it, since the items, quantities and prices are already in the lines begin passed. The system will then proceed to update the customer's current order with the charges coming from the lines.

If pricing fields are passed instead of order lines, then the system will call the mediation process to resolve the item, quantity and prices from the data coming as fields and create the order lines. With the line (or lines) resolved, the system then can update the current order.

The event description is saved in the mediation event record. Returns the updated order.

Input Parameters

`userId` (`Integer`): The identifier of the user whose current one-time order is being updated and returned.

`lines` (`OrderLineWS[]`): The order lines to be added to, or used to update, the current one-time order, depending on whether the items are already in the order.

`fields` (`PricingField[]`): Array of PricingField structures specifying optional pricing parameters to be passed to the rules engine for evaluation. Pricing fields are descriptors that provide further information to the pricing engine and aid in forming the price for the order itself. This parameter is optional and used in conjunction with the pricing rules engine.

See section “A word on pricing” for a more detailed explanation of the use of pricing fields and the purpose of rule-based pricing.

Note to SOAP based integration implementors: the pricingFields parameter of this call is implemented as an array of PricingField structures only in the API. Direct calls to the SOAP layer require you to encode and decode this array of values into a serialized string (so, for SOAP calls, the third parameter is actually a String). For more details on how to serialize these structures into strings, see section “A word on pricing”.

`date` (`Date`): The date that determines which current one-time order should be updated and returned. The date will fall within the order's active period.

`eventDescription` (`String`): The description to be used for the mediation event record.

Returns

`OrderWS`: The current one-time order for the user for the given date. If the user has no mediation main-subscription order, a `JbillingAPIException` is thrown.

getLatestOrderByItemType

Returns the latest order for the given user that contains item/s of the given item type.

Input Parameters

`userId` (`Integer`): The identifier of the user whose order is to be returned.

`itemTypeId` (`Integer`): The identifier of the type of item/s that the order must contain.

Returns

`OrderWS`: The latest order for the given user that contains item/s of the given item type. Returns `null` if no such order exists. If the `userId` or `itemTypeId` are `null`, a `JbillingAPIException` is thrown.

getLastOrdersByItemType

Returns the ids of the last orders for the given user that contain item/s of the given item type. The `number` parameter limits the maximum amount of order ids returned.

Input Parameters

`userId` (`Integer`): The identifier of the user whose order ids are to be returned.

`itemTypeId` (`Integer`): The identifier of the type of item/s that the orders must contain.

`number` (`Integer`): The maximum number of order ids to be returned.

Returns

`Integer[]`: An array of the order ids for the given user that contain item/s of the given type. Returns `null` if the `userId` or `number` parameters are `null`.

Item Management Calls

Items are the building blocks of purchase orders. Items are usually managed from the web-based application since they don't have the level of activity of orders or payments. Use this service for integration with other applications (for example, an inventory system) or when handling volumes that make the web-based application impractical.

An item can have a simple price or a percentage price. Items with a simple price will simply add that price to the total of a purchase order. A percentage price will impact that total by a percentage. Examples of items with percentage prices are taxes and interests.

Data Structures

ItemDTOEx

This data structure contains the description and properties for an item.

Property Name	Type	Description
currencyId	Integer	Identifier for the currency in which the item's price is expressed. See Appendix A for a list of acceptable values.
deleted	Integer	A flag that indicates if this record is logically deleted in the database. This allows for ‘undo’ of deletions. Valid values are: 0 – the record is not deleted 1 – the record is considered deleted.
description	String	The description of this item
entityId	Integer	Identifier for the entity to which this item belongs.
hasDecimals	Integer	An internal flag indicating whether the item accepts decimal quantities. Can have the following values: 0 – No decimals, quantities are expressed as an integer, 1 – Decimals allowed in quantity values.
id	Integer	A unique number that identifies this record.
number	String	This can be used to identify this item following an external coding system. For example, books can be identified by their ISBN codes.
orderLineTypeId	Integer	The order type that this item will generate, such a 'taxes', or 'items'.
percentage	BigDecimal	If this is a percentage item, its rate is specified in this field.

price	BigDecimal	The price of this item or null if this is a percentage item.
priceManual	Integer	A flag that indicates if this item will allow manual edition of the price when a purchase order is being placed with the web-based application. A value of 1 will allow manual editing, while a value of 0 will display the price as a read-only field.
prices	List<ItemPriceDTOEx>	An item can have many prices, one per currency. This is an array of ItemPriceDTOEx with all the available prices for this item.
promoCode	String	If this item is related to a promotion, this is the code that identifies the promotion.
types	Integer[]	A list of type identifiers that indicates to which types (categories) this item belongs. An item has to belong to at least one type. Item types are created from the web-based GUI by the billing administrator.

ItemPriceDTOEx

This structure contains information regarding an item's price.

Property Name	Type	Description
currencyId	Integer	Identifier for the currency in which the item's price is expressed. See Appendix A for a list of acceptable values.
id	Integer	The identifier of this price element.
name	String	The price's description.
price	BigDecimal	The amount of this price.
priceForm	String	Used for internal purposes only.

ItemTypeWS

This structure contains information regarding an item type.

Property Name	Type	Description
`description`	`String`	Description for this item type.
`id`	`Integer`	Identifier of this item type.
`orderLineTypeId`	`Integer`	Type of order line for this item. See “Order Line Type Codes” in “Appendix A” for valid values for this field.

ValidatePurchaseWS

Property Name	Type	Description
`authorized`	`Boolean`	“true” if the validation has been authorized, “false” otherwise.
`message`	`String[]`	An array of messages detailing the result of the validation operation.
`quantity`	`String`	Quantity of the item that can be applied without exceeding the user's remaining credit limit or prepaid balance.
`success`	`Boolean`	“true” if the validation was successful, “false” otherwise.

Methods

createItem

This method creates a new item. A new record with this item information will be inserted in the database. This method is frequently used to automate the uploading of a large number of items during the initial setup.

Input Parameters

`dto` (`ItemDTOEx`): Information for the item that is being created.

Return Value

`Integer`: The newly created item's identifier. If required fields in the input data structure are missing, a `JbillingAPIException` is thrown to signal the error.

getItem

This method returns the item description for an item. If business rules exist that affect the item being retrieved, they are applied and their effects are reflected in the returned data structure.

Input Parameters

`itemId` (`Integer`): Identifier of the item that is being retrieved. This is the only required parameter, if your intention is to simply retrieve the item data.

`userId` (`Integer`): Identifier of the user for which this item is being retrieved (useful to determine if any specific pricing rules apply to this customer). This parameter is optional and used in conjunction with the pricing rules engine.

`pricingFields` (`PricingField[]`): Array of PricingField structures specifying optional pricing parameters to be passed to the rules engine for evaluation. Pricing fields are descriptors that provide further information to the pricing engine and aid in forming the price for the order itself. This parameter is optional and used in conjunction with the pricing rules engine.

See section “A word on pricing” for a more detailed explanation of the use of pricing fields and the purpose of rule-based pricing.

Note to SOAP based integration implementors: the pricingFields parameter of this call is implemented as an array of PricingField structures only in the API. Direct calls to the SOAP layer require you to encode and decode this array of values into a serialized string (so, for SOAP calls, the third parameter is actually a String). For more details on how to serialize these structures into strings, see section “A word on pricing”.

Return Value

`ItemDTOEx`: The item description structure for the item being retrieved.

getAllItems

This method returns a list of all items registered for a company in **j**Billing at the moment the call is placed. It is useful to keep the data in other application synchronized with **j**Billing.

Input Parameters

None.

Return Value

`ItemDTOEx[]`: Array of item description data structures containing the information for all items currently registered in **j**Billing.

getAllItemCategories

This method returns the list of available item categories currently in the system. Item categories help organize item lists. Categories can be created and populated by means of **j**Billing's web interface.

Input Parameters

None. All currently available item categories will be returned.

Return Value

`ItemTypeWS[]`: An array of ItemTypeWS structures containing all item categories present in the system.

getItemByCategory

This method returns the list of available items currently in the system that belong to a specific item category. Item categories help organize item lists.

Input Parameters

`ItemTypeId (Integer)`: The identifier of the category whose items we want to obtain. Category identifiers can be obtained via the `getAllItemCategories()` API method, which returns the list of available categories.

Return Value

`ItemDTOEx[]`: An array of structures describing each of the items contained in the selected category. A `JbillingAPIException` is thrown if the identifier of the category passed in as a parameter is null or does not correspond to an existing category.

updateItem

Modifies the data associated to a **j**Billing item.

Input Parameters

`item (ItemDTOEx)`: Data structure for the item, containing the parameters being modified (most probably, this structure is retrieved by a previous call to `getItem()`, and changing the desired values from the structure returned).

Return Value

None

getUserItemsByCategory

Returns the ids of items of the given type (category) which are found in recurring (i.e., not one-time) orders for the given user.

Input Parameters

`userId` (`Integer`): The identifier of the user whose orders are searched for items.

`categoryId` (`Integer`): The identifier of the type of item/s that the orders must contain.

Return Value

`Integer[]`: An array of item ids of `categoryId` item type found in recurring orders for this `userId`.

validatePurchase

Returns the quantity available to the user of the given item when priced by the given pricing fields. The customer's dynamic balance is used for this calculating the maximum quantity that can be purchased before the customer's credit limit is reached or prepaid balance is reduced to zero.

Input Parameters

`userId` (`Integer`): The identifier of the user whose available quantity is being calculated.

`itemId` (`Integer`): The identifier of the item to have the quantity returned.

`fields` (`PricingField[]`): Array of PricingField structures specifying optional pricing parameters to be passed to the rules engine for evaluation. Pricing fields are descriptors that provide further information to the pricing engine and aid in forming the price for the order itself. This parameter is optional and used in conjunction with the pricing rules engine.

See section “A word on pricing” for a more detailed explanation of the use of pricing fields and the purpose of rule-based pricing.

Note to SOAP based integration implementors: the pricingFields parameter of this call is implemented as an array of PricingField structures only in the API. Direct calls to the SOAP layer require you to encode and decode this array of values into a serialized string (so, for SOAP calls, the third parameter is actually a String). For more details on how to serialize these structures into strings, see section “A word on pricing”.

Return Value

`ValidatePurchaseWS`: A structure describing the outcome of the operation. Returns `null` if `userId` or `itemId` is `null`.

validateMultiPurchase

Returns the quantity available to the user of the given items when priced by the given pricing fields. The customer's dynamic balance is used for this calculating the maximum quantity that can be purchased before the customer's credit limit is reached or prepaid balance is reduced to zero.

Input Parameters

`userId` (`Integer`): The identifier of the user whose available quantity is being calculated.

`itemIds` (`Integer[]`): The identifier of the items to have the quantity returned.

`fields` (`PricingField[][]`): Multidimensional array of PricingField structures specifying optional pricing parameters to be passed to the rules engine for evaluation (where `fields[0][n]` applies to `itemIds[0]`, `fields[1][n]` applies to `itemIds[1]`, and so on). Pricing fields are descriptors that provide further information to the pricing engine and aid in forming the price for the order itself. This parameter is optional and used in conjunction with the pricing rules engine.

See section “A word on pricing” for a more detailed explanation of the use of pricing fields and the purpose of rule-based pricing.

Note to SOAP based integration implementors: the pricingFields parameter of this call is implemented as an array of PricingField structures only in the API. Direct calls to the SOAP layer require you to encode and decode this array of values into a serialized string (so, for SOAP calls, the third parameter is actually a String). For more details on how to serialize these structures into strings, see section “A word on pricing”.

Return Value

`ValidatePurchaseWS`: A structure that describes the outcome of the validation. Returns `null` if `userId` or any of the `itemIds` passed as parameters are `null`.

Invoice Management Calls

The invoice management calls allow your application to query the system about invoices, and to attempt to pay an invoice through a payment gateway. Invoices in **j**Billing are, for the most part, read-only and are created based on purchase orders. Invoices can be generated, however, for orders that have yet to be invoiced.

Data Structures

InvoiceWS

This data structure contains data regarding an invoice.

Property Name	Type	Description
`balance`	`String`	The amount of this invoice that is yet to be paid. Can also be handled as a Java BigDecimal via the “getBalanceAsDecimal()” and “setBalanceAsDecimal()” methods.
`carriedBalance`	`String`	How much of the total belonging to previous unpaid invoices that have been delegated to this one. It can also be

		handled via the “getCarriedBalanceAsDecimal()” and “setCarriedBalanceAsDecimal()” methods.
createDateTime	Date	This is the invoice date, which is assigned to it by the billing process when it is generated.
createTimeStamp	Date	A time stamp of when this invoice record was created.
currencyId	Integer	Identifier of the currency in which the invoice's amounts are being expressed. See Appendix A for a list of all acceptable values.
customerNotes	String	Notes that are entered in a purchase order can be applied to an invoice. If that is the case, this field will have those user notes.
delegatedInvoiceId	Integer	If this invoice has been included in another invoice (usually for lack of payment), this field will indicate to which invoice it has been delegated.
deleted	Integer	A flag that indicates if this record is logically deleted in the database. This allows for ‘undo’ of deletions. Valid values are: 0 – the record is not deleted 1 – the record is considered deleted.
dueDate	Date	The due date of this invoice. After this date, the invoice should have been paid.
id	Integer	A unique number that identifies this record.
inProcessPayment	Integer	A flag indicating if this invoice will be paid using automated payment (through a payment processor), or if it will be paid externally (for example, with a paper check).
invoiceLines	InvoiceLineDTO[]	A list of objects representing each of this invoice’s lines.

isReview	Integer	This is an internal value that indicates if this invoice is not a 'real' invoice, but one that belongs to a review process.
lastReminder	Date	Date and time of when the latest reminder was issued for this invoice.
number	String	The invoice number, which is assigned to it from a 'preference' field (see the user guide for more information). This is not the ID, which is guaranteed to be unique.
orders	Integer[]	A list of the ids of the purchase orders which have been included in this invoice.
overdueStep	Integer	This marks which step is this invoice in for the penalties (interests) process.
paymentAttempts	Integer	How many payment attempts have been done by the automated payment process to get this invoice paid.
payments	Integer[]	A list of ids of the payments that have been applied to this invoice.
toProcess	Integer	This is '1' if the invoice will be considered by the billing process as unpaid. Otherwise it is '0' and the invoices is either paid or carried over to another invoice.
total	String	The total amount of this invoice. It can also be handled as a Java BigDecimal via the “getTotalAsDecimal()” and “setTotalAsDecimal()” methods.
userId	Integer	The customer to whom this invoice belongs.
statusId	Integer	A flag that indicates the status of this invoice. See section “Invoice Status Codes” on Annex A for valid values for this field.

InvoiceLineDTO

This data structure contains data relative to a single line of an invoice.

Property Name	Type	Description
amount	BigDecimal	The total amount for this line. Usually would follow the formula price * quantity.
deleted	Integer	A flag that indicates if this record is logically deleted in the database. This allows for 'undo' of deletions. Valid values are: 0 – the record is not deleted 1 – the record is considered deleted.
description	String	This description will be displayed in the invoice delivered to the customer.
id	Integer	A unique number that identifies this record.
isPercentage	Integer	Indicates whether the item referenced by this invoice line is a percentage item or not. This is used to aid how the order line is displayed to the customer.
itemId	Integer	Identifier of the item referenced by this invoice line.
price	BigDecimal	The pricing of a single unit of this item.
quantity	BigDecimal	The number of units of the item being invoiced.
sourceUserId	Integer	This field is useful only when many sub-accounts is invoiced together. This field would have the ID of the user that originally purchase an item.

Methods

getInvoiceWS

This method retrieves the invoice information regarding a specific invoice in the system.

Input Parameters

`invoiceId` (`Integer`): Identifier for the invoice that should be retrieved.

Return Value

`InvoiceWS`: contains the information regarding the invoice that was requested. If the identifier supplied as argument is `null` or does not represent a currently existing invoice, a `JbillingAPIException` is thrown.

deleteInvoice

Calling this method will delete the invoice record from the database, along with all its invoice lines. It will also remove any links between the invoice and any related payments, increasing the payments balance accordingly.

Input Parameters

`invoiceId` (`Integer`): the unique identifier of the invoice that is to be deleted.

Return Value

None. If the invoice identifier provided as input is null or does not correspond to an existing invoice, the method throws a `JbillingAPIException`.

getLatestInvoice

Returns the latest invoice that has been issued for a given user. This is particularly important because the latest invoice typically represents the account balance of a customer (this does not have to be the case, but it is a typical **j**Billing configuration).

Input Parameters

`userId` (`Integer`): identifier of the customer whose latest invoice is to be retrieved.

Return Value

`InvoiceWS`: contains the information regarding the latest issued invoice for the user. If the supplied user identifier is `null` or does not represent a currently existing user, a `JbillingAPIException` is thrown to signal the error.

getLastInvoices

Use this method to retrieve several invoices belonging to a customer, starting from the last one. The method will return the ids of the invoices, so you will have to call `getInvoice` to get the complete object related to each ID. The results will be returned in inverse chronological order. This is a handy method to provide your customers with historical data regarding their invoices. For example, `getLastInvoices(1234, 12)` will be returning the last 12 invoices of the customer id 1234, and if you invoice your customers in a monthly basis, that means one year of invoices.

Input Parameters

`userId` (`Integer`): identifier of the user whose latest invoices are to be retrieved.

`number` (`Integer`): the number of invoices that are to be retrieved for this user.

Return Value

`Integer[]`: array of invoice identifiers of the latest *n* invoices for this customer. It is possible that the customer has not generated as many invoices as requested (for example, you ask for the latest 10 invoices, but the system has generated only 8 invoices so far, you'll be returned those 8 invoices, so it is good practice to double check

the size of the returned array). The caller should check if the invoice is not deleted by verifying that that `deleted == 0`. If the customer does not have any invoices, an empty array is returned. If the `userId` provided as parameter is either `null` or does not represent a currently existing user, a `JbillingAPIException` will be thrown, to indicate the error condition.

getInvoicesByDate

Use this method to retrieve all the invoices created in a given period of time. The method will return the ids of the invoices, so you will have to call `getInvoice` to get the complete object related to each ID. The results will be returned in no particular order. This method can help you synchronize **j**Billing with other applications that require an updated list of invoices. For example, to get all the invoices for January 2005, you would call, `getInvoicesByDate("2005-01-01", "2005-01-31")`.

Input Parameters

`since` (`Date`): the starting date for the data extraction.

`until` (`Date`): the ending date for the data extraction.

Return Value

`Integer[]`: This method returns the invoices generated within the period specified by the parameters. Both dates are included in the query. The date used for the query is the actual creation of the invoices (time stamp), regardless of the 'invoice date', that is assigned following the billing rules and configuration parameters. Subsequent calls to `getInvoice` are necessary to retrieve the related `InvoiceWS` objects. If no invoices where generated for the specified period, an empty array is returned. If the parameters do not follow the required format (yyyy-mm-dd), `null` is returned.

getUserInvoicesByDate

Use this method to retrieve all the invoices created for a specific user in a given period of time. This could be useful for synchronizing invoices with an external application. See getInvoicesByDate for further details on the format of the dates passed as parameters.

Input Parameters

`userId (Integer)`: the identifier of the user for which extraction is desired.

`since` (`Date`): the starting date for the data extraction.

`until` (`Date`): the ending date for the data extraction.

Return Value

`Integer[]`: the id of all invoices generated for the given user within the period specified in input. Both dates are included in the query. The date used for the query is the actual creation of the invoices (time stamp), regardless of the 'invoice date', that is assigned following the billing rules and configuration parameters. Subsequent calls to `getInvoice` are necessary to retrieve the related `InvoiceWS` objects. If no invoices where generated for the specified period, an empty array is returned. If the user id

passed as parameter is null or does not refer to an existing user, `null` is returned. If the date parameters do not follow the required format (yyyy-mm-dd), `null` is returned.

createInvoice

Generates invoices for orders not yet invoiced for the given user. Optionally only allows recurring orders to generate invoices. Returns the ids of the newly created invoices.

Calling this method is the equivalent as running the billing process for one single customer. The system will go over all the applicable orders (and overdue invoices) to generate one invoice. This is different than calling 'createOrderAndInvoice'. In that method, the invoice comes from one single order, while in this one (crateInvoice), it comes from potentially many orders.

Input Parameters

`userId` (`Integer`): The identifier of the user that is having the invoices created.

`onlyRecurring` (`boolean`): If true, only recurring orders can generate invoices. If false, both one-time and recurring orders can generate invoices. Please note that if the preference 'ALLOW_INVOICES_WITHOUT_ORDERS' (46) is set to true, this will take precedence to any value coming from this parameter. If you have prefence 46 set to true and you call createInvoice with onlyRecurring equal to false, the system will use a 'true' anyway.

Return Value

`Integer[]`: An array of the ids of the newly created invoices.

getLatestInvoiceByItemType

Returns the latest invoice for the given user that contains item/s of the given item type.

Input Parameters

`userId` (`Integer`): The identifier of the user whose invoice is to be returned.

`itemTypeId` (`Integer`): The identifier of the type of item/s that the invoice must contain.

Returns

`InvoiceWS`: The latest invoice for the given user that contains item/s of the given item type. Returns `null` if no such invoice exists or the `userId` is null. .

getLastInvoicesByItemType

Returns the ids of the last invoices for the given user that contain item/s of the given item type. The `number` parameter limits the maximum amount of invoice ids returned.

Input Parameters

`userId` (`Integer`): The identifier of the user whose invoice ids are to be returned.

`itemTypeId` (`Integer`): The identifier of the type of item/s that the invoices must contain.

`number` (`Integer`): The maximum number of invoice ids to be returned.

Returns

`Integer[]`: An array of the invoice ids for the given user that contain item/s of the given type. Returns `null` if the `userId`, `itemTypeId` or `number` parameters are `null`.

Payment Management Calls

Payments play a crucial role in determining the status of a customer. For the system to properly trace payments, they have to be linked to the invoices they are paying. Payments not related to an invoice should be avoided; they are intended for initial imports from legacy billing systems and exceptional circumstances. Arbitrary credit card payment processing not linked to any invoice is possible, however.

There are two basic types of payments: cheques and credit cards. Different objects are used as fields of `PaymentWS` for each for these types.

There are two different actions related to payments. One is to apply a payment. This means that the payment has already been processed and accepted by the company. Applying the payment will let the system know that a new payment has to be registered. A common example is when the company has received and successfully deposited a cheque. In this case, the system only has to apply the payment to the customer's account.

A more comprehensive action is to process a payment. Processing a payment means the company will submit the payment information to the system, then the system will submit the payment to a payment process for its authorization in real-time, storing the result of this operation. The caller will get the results of this authorization. The most common example for this operation is a credit card payment.

In summary, the system can both request authorization of a payment and apply the payment to the customer's account, or it can do only this last step.

The payment management calls will let you query the system for payments, and also apply and submit a payment to get an invoice payed.

Data Structures

PaymentWS

This data structure contains data pertaining a payment operation.

Property Name	Type	Description
`id`	`Integer`	Unique identifier of the payment record.
`amount`	`String`	The amount of the payment operation. Can also be handled as a

		Java BigDecimal via the “getAmountAsDecimal()” and “setAmountAsDecimal()” methods.
balance	String	Balance of this payment. If greater than 0, this payment could pay part of another invoice. If 0, this payment has already been applied to an invoice, lowering the invoice's balance. It can also be handled as a Java BigDecimal via the “getBalanceAsDecimal()” and “setBalanceAsDecimal()” methods.
createDateTime	Date	Date in which this payment record was created.
updateDateTime	Date	Date in which this payment record was last updated.
paymentDate	Date	Date of the payment.
attempt	Integer	Number of the attempt to process this payment.
deleted	Integer	“1” if this record has been deleted, “0” otherwise.
methodId	Integer	Identifier of the payment method. Refer to Appendix A for a list of acceptable values.
resultId	Integer	Identifier of the result of the payment attempt. Refer to Appendix A for a list of acceptable values.
isRefund	Integer	“1” if this payment constitutes a refund operation, “0” otherwise.
isPreauth	Integer	“1” if this payment is a pre-authorization, “0” otherwise.
currencyId	Integer	Identifier of the currency in which the payment is being made. See Appendix A for a list of acceptable values.

userId	Integer	Identifier of the user this payment record belongs to.
cheque	PaymentInfoChequeDTO	If this payment is done via check, this property contains information about the cheque, otherwise it contains “null”.
creditCard	CreditCardDTO	If this is a credit card payment, this property contains information about the credit card, otherwise it contains “null”.
ach	AchDTO	If this is a payment done with Automatic Clearing House (ACH), this property contains the banking information needed.
method	String	Name of the payment method used.
invoiceIds	Integer[]	Contains the list of invoices this payment is paying.
paymentId	Integer	Refund specific field. When a refund is to be issued, this field holds the identifier of the payment that is to be refunded.
authorizationId	Integer	Refund specific field. Contains the identifier of the authorization field for the refund.

PaymentInfoChequeDTO

This data structure contains information regarding cheques.

Property Name	Type	Description
id	Integer	The unique identifier of this record.
bank	String	The name of the bank this cheque's account belongs to.
number	String	The cheque's number.
date	Date	The cheque's date.

AchDTO

This data structure contains the banking information needed to process an automatic clearing house payment.

Property Name	Type	Description
`abaRouting`	`String`	ABA routing number
`accountName`	`String`	The account name
`accountType`	`Integer`	If this is chequings or a savings account.
`bankAccount`	`String`	The account number
`bankName`	`String`	The bank name
`id`	`Integer`	The unique identifier of this record.

PaymentAuthorizationDTO

This data structure contains information about the payment authorization process.

Property Name	Type	Description
`id`	`Integer`	Unique identifier of the payment authorization.
`processor`	`String`	Name of the payment processor.
`code1`	`String`	Request code number 1.
`code2`	`String`	Request code number 2.
`code3`	`String`	Request code number 3.
`approvalCode`	`String`	Approval code provided by the processor.
`AVS`	`String`	A code with the results of address verification.
`transactionId`	`String`	Identifier of the processor transaction.
`MD5`	`String`	Hash for the transaction.

`cardCode`	`String`	Payment card code.
`createDate`	`Date`	The creation date for this payment authorization record.
`responseMessage`	`String`	The response provided by the processor.

PaymentAuthorizationDTOEx

This data structure contains data about a payment authorization process. It extends the `PaymentAuthorizationDTO` structure (see above). Thus, it shares all the same fields plus the following:

Property Name	Type	Description
`result`	`Boolean`	"true" if the authorization succeeded, "false" otherwise.

Methods

applyPayment

This method enters a payment to the user account. It does not invoke any payment processes, it just signals the payment as "entered". It is useful to signal payments done via external payment processes (a cheque being cashed, for example).

This method can apply a payment of any type. The parameter for the related invoice, although optional, should always be specified to allow the system to properly trace late payments.

Input Parameters

`payment` (`PaymentWS`): data of the payment being applied.

`invoiceId` (`Integer`): optionally identifies an invoice that is being paid by this payment. This parameter can be `null` (indicating this payment does not cover a specific invoice, or covers more than one).

Return Value

`Integer`: identifier of the newly created payment record.

payInvoice

This method executes a payment for a given invoice. The system will call the payment processor plug-in configured in your system to access a payment gateway and submit the payment to it.

Input Parameters

`invoiceId` (`Integer`): identifier of the invoice that is to be paid.

Return Value

`PaymentAuthorizationDTOEx`: contains information about the payment operation, for example, if the payment attempt was successful or not. It will return null if the invoice does not have a balance greater than 0 or if the user does not have a payment method that allows on-line payment processing (usually a credit card).

getPayment

This method returns information about a specific payment.

Input Parameters

`paymentId` (`Integer`): identifier of the payment to be retrieved.

Return Value

`PaymentWS`: information for the specified payment. If the input parameter is either `null` or does not correspond to a currently existing payment record, a `JbillingAPIException` is issued.

getLatestPayment

This method returns the latest payment entered or processed for a specific customer.

Input Parameters

`userId` (`Integer`): identifier of the customer whose payment information is to be retrieved.

Return Value

`PaymentWS`: information for this customer's latest payment. Can be `null` (no payments present for this customer).

getLastPayments

Use this method to retrieve several payments belonging to a customer, starting from the last one. The method will return the ids of the payments, so you will have to call `getPayment` to get the complete object related to each ID.

The results will be returned in inverse chronological order. This is a handy method to provide your customers with historical data regarding their payments. For example, `getLastPayments(1234, 12)` will be returning the last 12 payments of the customer id 1234.

Input Parameters

`userId` (`Integer`): identifier of the customer whose payment information is to be retrieved.

`number` (`Integer`): the number of payments to retrieve.

Return Value

`Integer[]`: array of payment identifiers for the latest `n` payments processed for the user. If the input parameters are missing or incorrect, a `JbillingAPIException` is issued.

processPayment

Allows arbitrary payments to be processed without an invoice. The system will call the payment processor plug-in configured in your system to access a payment gateway and submit the payment to it.

Input Parameters

`payment` (`PaymentWS`): data of the payment being applied.

Return Value

`PaymentAuthorizationDTOEx`: contains information about the payment operation, for example, if the payment attempt was successful or not. If the `payment` object is not for a credit card or ach payment, a `JbillingAPIException` is thrown.

Appendix A

Constants Reference

Language Codes

Language Code	Language Name
1	English
2	Portuguese

Currency Codes

Currency Code	Currency Name	ISO Code	Symbol	Country
1	United States Dollar	USD	US$	USA
2	Canadian Dollar	CAD	C$	Canada
3	Euro	EUR	€	Europe
4	Yen	JPY	¥	Japan
5	Pound Sterling	GBP	£	United Kingdom
6	South Korean Won	KRW		South Korea
7	Swiss Franc	CHF	Sf	Switzerland
8	Swedish Krona	SEK	SeK	Sweden
9	Singapore Dollar	SGD	S$	Singapore
10	Ringgit	MYR	M$	Malaysia
11	Australian Dollar	AUD	$	Australia

Role Codes

Role Code	Meaning
1	Internal
2	Super User
3	Clerk

4	Partner
5	Customer

User Status Codes

User Status Code	Meaning
1	Active
2	Overdue
3	Overdue 2
4	Overdue 3
5	Suspended
6	Suspended 2
7	Suspended 3
8	Deleted

Subscriber Status Codes

Subscriber Status Code	Meaning
1	Active
2	Pending Subscription
3	Unsubscribed
4	Pending Expiration
5	Expired
6	Nonsubscriber

Payment Method Codes

Payment Method Code	Meaning
1	Cheque
2	Visa
3	MasterCard
4	AMEX
5	ACH
6	Discovery
7	Diners
8	PayPal

Order Status Codes

Order Status Code	Meaning
1	Active
2	Finished
3	Suspended
4	Suspended (auto)

Order Billing Type Codes

Order Billing Type Code	Meaning
1	Pre-paid
2	Post-paid

Order Line Type Codes

Order Line Type Code	Meaning
1	Item
2	Tax
3	Penalty

Invoice Status Codes

Invoice Status Code	Meaning
1	Paid
2	Unpaid
3	Unpaid, balance carried to a new invoice.

Period units (units of time)

Unit	Meaning
1	Month
2	Week
3	Day
4	Year

Payment results

Result	Meaning
1	Successful
2	Failed
3	Payment gateway not available

4	Manually entered (payment gateway not involved, such as a cheque).

Dynamic Balance Codes

Dynamic Balance Code	Meaning
1	No dynamic balance
2	Pre paid
3	Credit limit

Provisioning Status Codes

Provisioning Status Code	Meaning
1	Active
2	Inactive
3	Pending Active
4	Pending Inactive
5	Failed
6	Unavailable