

Build a web-based notification tool with XMPP

Write real-time web applications with XMPP, PHP, and JavaScript

Skill Level: Intermediate

[Ben Werdmuller \(ben@benwerd.com\)](mailto:ben@benwerd.com)

Consultant and writer

Freelance

22 Jun 2010

Real-time web applications are networked applications, with web-based user interfaces, that display Internet information as soon as it's published. Examples include social news aggregators and monitoring tools that continually update themselves with data from an external source. In this tutorial, you will create Pingstream, a small notification tool that uses PHP and JavaScript to communicate over the Extensible Messaging and Presence Protocol (XMPP), a set of XML technologies designed to support presence and real-time-communications functionality.

Section 1. Before you start

This tutorial introduces you to the real-time web and takes you through some of the reasons for building real-time web applications. You'll learn techniques that allow you to create responsive, continually updated web applications that conserve server resources while providing a slick user experience.

About this tutorial

Frequently used acronyms

- DOM: Document Object Model
- HTML: HyperText Markup Language
- HTTP: Hypertext Transfer Protocol
- REST: Representational State Transfer
- RSS: Really Simple Syndication
- URL: Uniform Resource Locator
- XML: Extensible Markup Language

Real-time web applications allow users to receive notifications as soon as information is published, without needing to check the original source manually for updates. They have been popularized by social-notification tools like Twitter and Friendfeed, web-based collaboration tools like Google Wave, and web-based chat clients like Meebo.

The Extensible Messaging and Presence Protocol (XMPP) is an XML-based set of technologies for real-time applications, defined as networked applications that continually update in response to new or changed data. It was originally developed as a framework to support instant messaging and presence applications within enterprise environments.

In this tutorial you will build Pingstream, a simple tool that continually updates itself with RSS feed updates as they are published (see [Download](#) for the Pingstream source code). Along the way you will:

- Learn why XMPP is particularly suited to web applications
- Learn about the components of XMPP communication
- Install and configure the Openfire XMPP server
- Connect to the XMPP server using PHP and the XMPPHP library
- Check for new items in an RSS feed and transmit them over XMPP
- Use Strophe and jQuery to connect to an XMPP server over HTTP using Bidirectional-streams Over Synchronous HTTP (BOSH)
- Display XMPP notifications in a web page

Prerequisites

This tutorial assumes that you have some familiarity with developing web applications in PHP, although little advanced programming is involved. You should also have some experience with HTML and JavaScript. Experience with the jQuery

JavaScript framework might be handy. No prior familiarity with XMPP or similar technologies is required.

To follow along with this tutorial, you must have the following server software installed and running:

- PHP 5.2 or higher
- Apache HTTP Server
- MySQL

As you progress through the tutorial, you will also download and install the following software and libraries:

- Openfire
- jQuery
- Strophe
- XMPPHP
- Last RSS

You might also find the MySQL server tool phpMyAdmin useful. If you are using a desktop machine to test your real-time web application software locally, you might find XAMPP useful for managing the installation and running of a test web server infrastructure.

See [Resources](#) for links to all the tool downloads.

Section 2. Introducing the real-time web

In this section, you'll learn what real-time web applications are, why you might want to build one, and how they differ from the typical model for modern web applications.

Taking input and providing feedback

Real time on and off the web

A better term than *real-time web application* is *continually updating web application*. In computer science, both *hard* and *soft* real-time systems must meet operational deadlines. A hard real-time system

fails when a task doesn't complete in the time allotted to it. Real-time web applications are closer to soft real-time systems, in which the lateness of a function doesn't result in a failure of the system (but might degrade performance). But you don't strictly assign task scheduling to a real-time web application. The web is not, currently, an appropriate platform for time-critical applications.

Applications are pieces of specialized software that help users perform tasks. Characteristically, they take input from the user or other sources and provide human-readable output. They might also respond dynamically—in a visual or programmatic way—to changes in input data that's received automatically. For example, a news-monitoring application might notify the user if a news story containing a particular keyword appears in a newswire that the application connects to.

Because of its origins as a document-serving platform, the web is not optimized for applications. HTML is great for representing and hyperlinking text content, but not for creating dynamic interfaces. Web applications can accept and respond to user input, thanks to the combination of server-side scripting languages such as PHP and simple web-input technologies such as forms and JavaScript. But you must overcome some hurdles to create interfaces that update *automatically*. This is a more difficult problem because no front-end web technology has been developed with this functionality in mind. In contrast (taking the news-monitoring example), desktop software doesn't need to refresh its interface for notifications to be delivered to the user; it can continually update itself. The web, on the other hand, is bound to a page-based model.

Nonetheless, real-time web-based applications are possible, and their benefits are clear. Such applications include enterprise chat, networked real-time document collaboration, and search interfaces that reveal new content as soon as it's published.

Web technologies for robust application development

Typically, web applications simulate continually updated interfaces by using Asynchronous JavaScript and XML (Ajax) to poll a server frequently. In this model, the application's web page contains JavaScript, which makes repeated requests to a server callback in the background. Although the responsiveness of Ajax applications is adequate in many situations, this technique has drawbacks.

Ajax does not fare well with unreliable Internet connectivity: a temporarily dropped connection can cause the whole interface to fail. It is also inefficient in terms of server load. Suppose your background Ajax polling function checks the server every 10 seconds. Each time, a new HTTP connection is established, including initialization of the resources needed to service the request, even if there's no new

data to present to the user. The result is an application that uses considerably more processor time and bandwidth than it needs to.

XML-based technologies provide significant advantages to web-scale applications. XML parsers are now a standard part of most environments; no extra software is required to support the reading and writing of data in a suitable format. XML is self-describing; documents that use it do not require an external schema. Finally, just as the web is platform-independent, XML as a technology is interoperable among platforms. As a result, developers can concentrate on the logic specific to their applications.

The web is built on free, interoperable, open standards such as HTML, Cascading Style Sheets (CSS), and JavaScript. If a standard is to emerge for real-time communication on the web, it too should be open, interoperable, and free. XMPP, which is based on XML, meets these criteria. In this tutorial, you will use it to build a client library that accepts input through standard methods (for example, a web hook) and relays the appropriate data to the user in real time.

Section 3. Introducing XMPP

In this section, you'll look at XMPP, its origins, and why it is a suitable protocol for real-time web communications. You will examine the components of an XMPP communication setting, and review some examples of how these can be used.

Web standards and XMPP

XMPP is an XML-based set of technologies for real-time applications. It was originally developed as a framework to support instant messaging and presence applications within enterprise environments. At the time, the existing instant-messaging networks were proprietary and largely inappropriate for enterprise use. AOL Instant Messenger, for example, could not be adapted for secure communications inside a company. Although commercial solutions existed, their fixed feature sets typically could not be adapted to meet an organization's needs. XMPP, then called Jabber, allowed organizations to build their own custom tools to facilitate real-time communication, as well as install ready-made third-party solutions.

XMPP is a decentralized communication network, which means that any XMPP user can message any other XMPP user, network infrastructure permitting. XMPP servers can also talk to one another with a specialized server-to-server protocol, providing interesting potential for decentralized social networks and collaboration

frameworks, although that is beyond the scope of this tutorial.

As its name suggests, XMPP can be used to meet a wide variety of time-sensitive feature requirements. Indeed, Google Wave, a large multiuser collaboration environment, uses XMPP as the basis for its federation protocol. Although XMPP emerged from a requirement for person-to-person instant messaging, it is by no means limited to that task.

The structure of XMPP communication

To facilitate message delivery, each XMPP client user must have a universally unique identifier. For legacy reasons, these are referred to as Jabber IDs, or JIDs. Because of the protocol's distributed nature, it's important that a JID contain all the information required to contact a user: no central repository links users to the servers they connect to. The structure of a JID is similar to an email address (although there is no requirement for a JID to double as a valid email recipient).

Both client and server nodes, which I refer to collectively as *XMPP entities*, have JIDs. John Doe, an employee at SomeCorp, might have the JID `John.Doe@somecorp.com`. Here, `somecorp.com` is the address of the XMPP server for SomeCorp, and `John.Doe` is the username for John Doe.

JIDs can also have a resource attached to them. These allow for further addressing granularity beyond an identifier for an XMPP entity; for example, whereas the preceding example can represent John Doe in totality, `John.Doe@somecorp.com/Work` might be used to send data to his work-related tools.

These resources can take any user-defined name, and an XMPP entity can have any number of resources. As well as being context-dependent, they can be bound to a device, tool, or workstation. For your Pingstream example, each visitor to the website will log into the XMPP server as the same user but have different resources.

Categories of communication

A real-time messaging system using XMPP comprises three broad categories of communication:

- Messaging, wherein data is transferred between parties
- Presence, which allows users to broadcast their online status and availability
- Info/query requests, which allow an XMPP entity to make a request and

receive a reply from another entity

These are complementary. For example, there's often no point sending data to a user, or making an info/query request of an entity, if the user or entity is offline (although in many use cases it is desirable for the server to hold messages for users until they return). Each of them is delivered through a complete XML *stanza* —a discrete item of information expressed in XML.

XMPP stanzas of all three types have the following attributes in common:

- `from`: The JID of the source XMPP entity'
- `to`: The JID of the intended recipient
- `id`: An optional identifier for the conversation
- `type`: Optional subtype of the stanza
- `xml:lang`: If the content is human-readable, description of the message's language

Data transfer over XMPP takes place over XML streams, operating on port 5222 by default. These are effectively two complete XML documents, each corresponding to a direction of communication. Once a session is established, the `stream` element is opened. It will envelop the entire communication document. The stanzas are then injected into the second level of the document. Finally, once communication ends, the `stream` element is closed, forming a complete document.

For example, [Listing 1](#) shows a `stream` element establishing communication from the client to the server:

Listing 1. Stream tag establishing communication from client to server

```
<stream:stream from="[server]" id="[unique ID over conversation]"
xmlns="jabber:client" xmlns:stream="http://etherx.jabber.org/streams" version="1.0">
```

Messages

Once communication is established, the client can send a message to another user with the `message` element, which can contain any of the following child elements:

- `subject`: A human-readable string representing the message topic.
- `body`: A human-readable string representing the message body. Multiple `body` tags can be included, if each one has a different `xml:lang` value. (`xml:lang` is the only possible attribute.)

- `thread`: A unique identifier representing a message thread. Client software can then use this to string together related messages.

However, messages can be as simple as the one in [Listing 2](#):

Listing 2. Sample message

```
<message from="sendinguser@somedomain" to="recipient@somedomain" xml:lang='en'>
  <body>
    Body of message
  </body>
</message>
```

Message stanzas are the most useful stanzas for providing real-time web interfaces. The publish-subscribe model—an alternative to using messages to transmit data in real-time web applications—is described next.

Info/query

Info/query stanzas can have a wide variety of functionality. One example is the publish-subscribe model, wherein a publisher notifies the server of updates to a particular resource, and the server in turn notifies all XMPP users who have opted to subscribe to these notifications and are authorized to do so.

A series of items from the publisher are encoded as stanzas in the Atom XML-based publishing format. Each item is contained within an `item` element, then collectively within a `pubsub` element, and finally as an info/query stanza. In [Listing 3](#) (taken from the XMPP publish-subscribe specification) Shakespeare's Hamlet (with JID `hamlet@denmark.lit/blogbot`) is publishing an update to the `pubsub.shakespeare.lit` pubsub update node with his famous soliloquy:

Listing 3. Update to the `pubsub.shakespeare.lit` pubsub update node

```
<iq type="set"
  from="hamlet@denmark.lit/blogbot"
  to="pubsub.shakespeare.lit"
  id="pub1">
  <pubsub xmlns="http://jabber.org/protocol/pubsub">
    <publish node="princely_musings">
      <item>
        <entry xmlns="http://www.w3.org/2005/Atom">
          <title>Soliloquy</title>
          <summary>
            To be, or not to be: that is the question:
            Whether 'tis nobler in the mind to suffer
            The slings and arrows of outrageous fortune,
            Or to take arms against a sea of troubles,
            And by opposing end them?
          </summary>
          <link rel="alternate" type="text/html"
            href="http://denmark.lit/2003/12/13/atom03"/>
        </entry>
      </item>
    </publish>
  </pubsub>
</iq>
```

```

        <id>tag:denmark.lit,2003:entry-32397</id>
        <published>2003-12-13T18:30:02Z</published>
        <updated>2003-12-13T18:30:02Z</updated>
    </entry>
</item>
</publish>
</pubsub>
</iq>

```

Info/query stanzas are also used to request information about a particular XMPP entity. For example, in the stanza in [Listing 4](#), `boreduser@somewhere` is looking for public items owned by `friendlyuser@somewhereelse`:

Listing 4. User looking for public items owned by `friendlyuser@somewhereelse`

```

<iq type="get"
  from="boreduser@somewhere"
  to="friendlyuser@somewhereelse"
  id="publicStuff">
  <query xmlns="http://jabber.org/protocol/disco#items"/>
</iq>

```

In turn, `friendlyuser@somewhereelse` responds with a list of items that can be subscribed to using publish-subscribe, as in [Listing 5](#):

Listing 5. Response with a list of items

```

<iq type="result"
  from="friendlyuser@somewhereelse"
  to="boreduser@somewhere"
  id="publicStuff">
  <query xmlns="http://jabber.org/protocol/disco#items">
    <item jid="stuff.to.do"
      name="Things to do"/>
    <item jid="stuff.to.not.do"
      name="Things to avoid doing"/>
  </query>
</iq>

```

Each item returned in the result info/query stanza in [Listing 5](#) has a JID that can be subscribed to. Info/query stanzas also allow for a wide variety of server information requests that are outside the scope of this tutorial. Many of them are useful in a web application context for multiserver environments, or as the basis of sophisticated decentralized collaboration frameworks.

Presence

Presence information is contained within a presence stanza. If the `type` attribute isn't present, the XMPP client application assumes that the user is online and available. Otherwise, `type` can be set to `unavailable`, or to the pubsub-specific

values `subscribe`, `subscribed`, `unsubscribe`, and `unsubscribed`. It can also be an error or a probe for the presence information of another user.

A presence stanza can contain the following child elements:

- `show`: A machine-readable value indicating the overall category of online status to display. This can be `away` (temporarily away), `chat` (available and interested in communication), `dnd` (do not disturb), or `xa` (away for an extended time).
- `status`: A human-readable counterpart to `show`. The value is a user-definable string.
- `priority`: A numeric value between -128 and 127 that defines the priority by which messages should be routed to the user. If the value is negative, messages for the user are held.

For example, the `boreduser@somewhere` might use the stanza in [Listing 6](#) to indicate a willingness to chat:

Listing 6. Sample presence notification

```
<presence xml:lang="en">
  <show>chat</show>
  <status>Bored out of my mind</status>
  <priority>1</priority>
</presence>
```

Note the absence of a `from` attribute.

Another user, `friendlyuser@somewhereelse`, might probe the status of `boreduser@somewhere` by sending the stanza in [Listing 7](#):

Listing 7. Probing a user's status

```
<presence type="probe" from="friendlyuser@somewhereelse" to="boreduser@somewhere"/>
Boreduser@somewhere's server would then respond with a tailored presence response:
<presence xml:lang="en" from="boreduser@somewhere" to="friendlyuser@somewhereelse">
  <show>chat</show>
  <status>Bored out of my mind</status>
  <priority>1</priority>
</presence>
```

These presence values derive from person-to-person messaging software. It isn't clear how the value of the `show` element—normally used to determine a status icon to display to other users—is used apart from a literal chat application. Status values might find use in microblogging tools; for example, changes to the status field of a user in Google Talk (an XMPP chat service) can optionally be imported as microblog entries in Google Buzz.

Another possibility is to use status values as carriers for per-user application state data. Although the specification defines status as being human-readable, nothing stops you from storing any string there to suit your purposes. For some applications, it might not be human-readable, or it might carry a data payload in the form of microformats.

You can set presence information independently for each resource owned by an XMPP entity, so only one user account is required to access and receive data for all the tools and contexts connected to a single user within an application. Each resource can be assigned an independent priority level; the XMPP server will try to deliver messages to resources with higher priorities first.

XMPP over HTTP using BOSH

Web applications that wish to communicate over XMPP using JavaScript must meet some special requirements. For security reasons, JavaScript is not allowed to communicate with servers on different domains from the web page's domain. If your web application interface is hosted at `application.mydomain.com`, any XMPP communication must also take place at `application.mydomain.com`.

Firewalls are an additional complication. Ideally, if you use XMPP as the basis of the real-time element of your web interface, you want it to work for users behind a firewall. But corporate firewalls usually leave open only a few ports for a small number of protocols, to allow web data, email, and similar communications to get through. By default, XMPP uses port 5222, which a corporate firewall might well block.

Suppose you know that the firewall your users might sit behind will allow HTTP on port 80 (the default protocol and port for accessing the web). It is optimal if your XMPP communications can tunnel over HTTP on that port. However, HTTP isn't designed for continuous connections. The architecture of the web is different from the communications architecture required for real-time data.

Enter the standard for Bidirectional-streams Over Synchronous HTTP (BOSH). It provides an emulation layer for bidirectional, synchronous data. A long HTTP connection (with a duration of a minute or two) is established with an XMPP server. If new data arrives during that time, the HTTP request returns data and closes; otherwise, it simply expires. Either way, once a request has closed, another one is reestablished. Although the result is a series of repeated connections to a web server, it is an order of magnitude more efficient than Ajax polling, particularly because the connections are made to a specialized server rather than directly to the web application.

XMPP over BOSH allows the web application to continue to communicate with the XMPP server through a native connection. The client connects through a standard

URL over HTTP on port 80. This is then proxied by the web server to an HTTP URL on a different port—often 7070—operated by the XMPP server. As a result, the web application only needs to use resources whenever data is sent to the XMPP server, while the web client can operate from behind a firewall using commonly supported web standards. The overhead of maintaining the long HTTP polls of BOSH is mostly handled by the XMPP server rather than the web server or the web application. Neither the web server nor the XMPP server is subject to the same domain restrictions as JavaScript for their communications, and it is at this point that messages can be sent to other XMPP servers and clients.

Now that you understand how XMPP fits into the real-time web, you're ready to download and set it up so that you can begin to create the Pingstream application.

Section 4. Obtaining and installing an XMPP server

In this section, you'll install the Openfire XMPP server and configure it to support your real-time web application.

Choosing an XMPP server

Two leading open source XMPP servers are available for free download. Both are widely used and available under the GNU Public License version 2, and each has its own strengths and weaknesses:

- **ejabberd:** The *e* in ejabberd refers to Erlang, the soft-real-time programming language. This technical foundation makes ejabberd very fast. It also has a high level of compliance with the XMPP core and related standards. ejabberd can be installed in most environments.
- **Openfire:** Openfire is written in the Java™ language, and is both user friendly and easy to install.

You'll use Openfire for this tutorial.

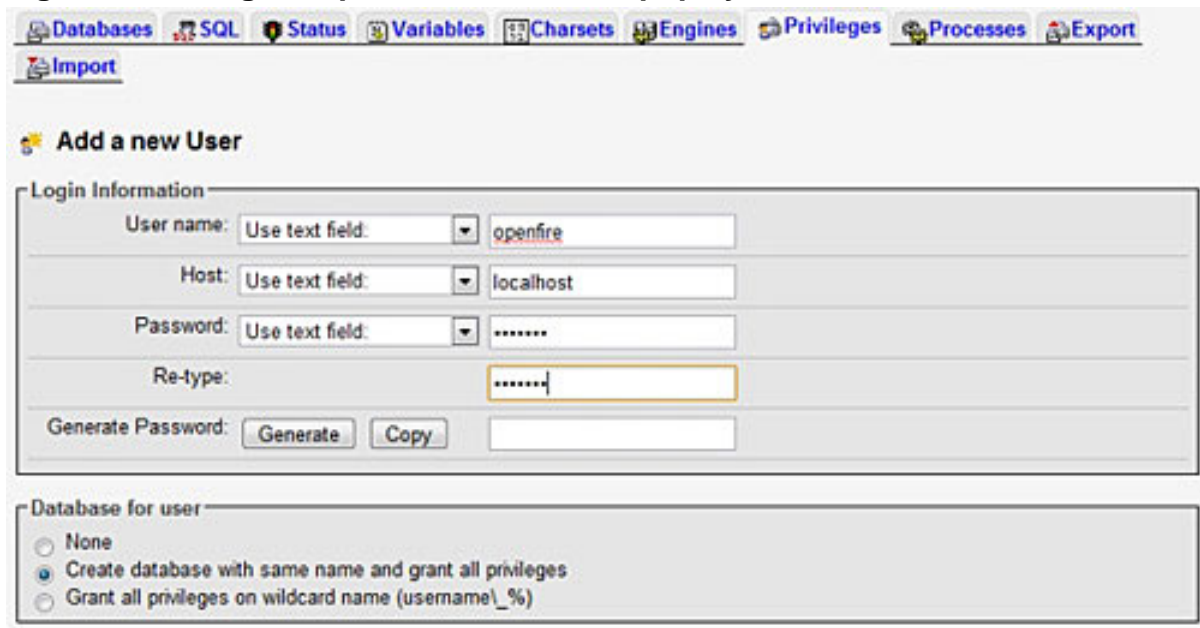
Creating the Openfire database

Create a new MySQL database for your Openfire users and configuration. By using MySQL, you can programmatically add, edit, delete, and query your XMPP server users from your PHP web application, as well as scale your XMPP infrastructure in line with your web infrastructure.

If you have phpMyAdmin installed, for example as part of your XAMPP installation, you can follow these steps to create the database:

1. Select **Privileges** from the home screen.
2. Select **Add a new user**.
3. Add user details (ensuring the host is localhost; for the duration of this tutorial, it will be assumed that you're testing on localhost), and select **Create database with same name and grant all privileges**, as in [Figure 1](#). Don't give your new Openfire database user any global database privileges.

Figure 1. Adding an Openfire database in phpMyAdmin



The screenshot shows the 'Add a new User' form in phpMyAdmin. The 'Login Information' section includes fields for 'User name' (set to 'openfire'), 'Host' (set to 'localhost'), 'Password' (masked with asterisks), and 'Re-type' (also masked). There are 'Generate Password' and 'Copy' buttons. The 'Database for user' section has three radio button options: 'None', 'Create database with same name and grant all privileges' (which is selected), and 'Grant all privileges on wildcard name (username_%)'.

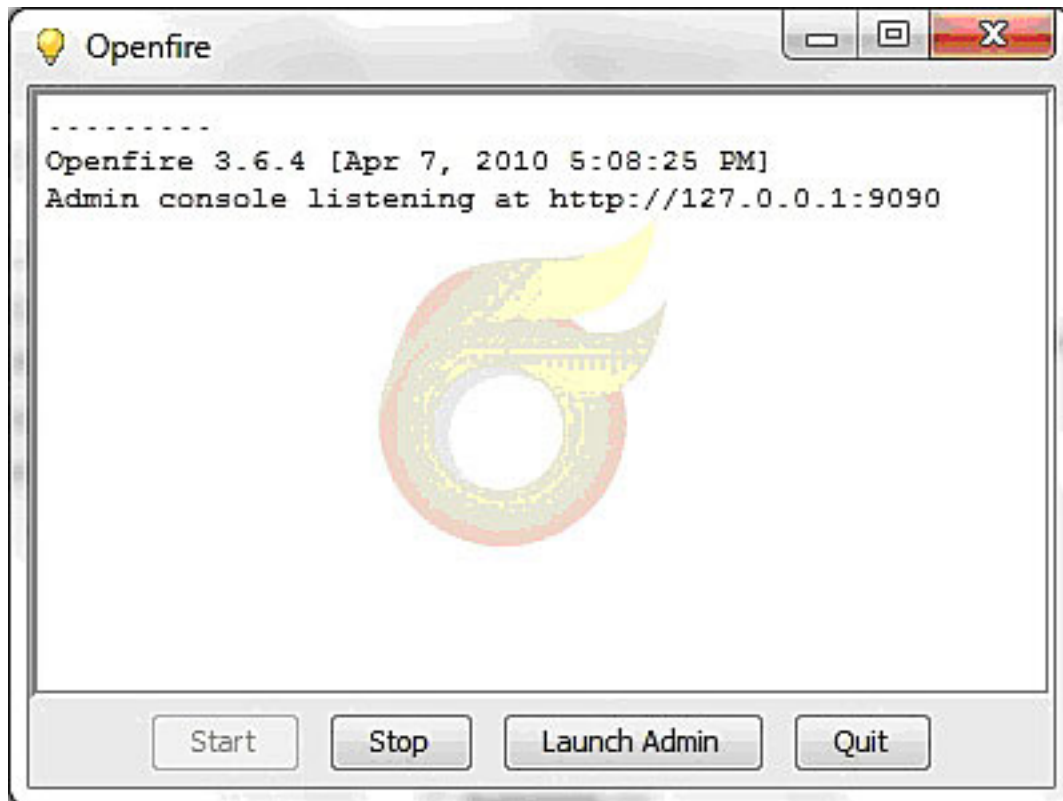
Once you add your user and database, you're ready to install the Openfire server.

Installing Openfire

Download the Openfire installer and run it to install Openfire into the location of your choice (see [Resources](#)). (You can instead choose to check out a recent Openfire version from its Subversion source code repository and build it locally, but that's beyond the scope of this tutorial.) When prompted, tell the Openfire installer to launch the server upon completion.

After the server launches, you should see the server status window in [Figure 2](#):

Figure 2. Status window for Openfire



Click **Launch Admin** to open a web-based wizard, in [Figure 3](#), that takes you through the configuration of your Openfire server:

Figure 3. Configuration wizard

Openfire Setup: Database...

http://127.0.0.1:9090/setup/setup-datasource-standard.jsp

Setup Progress

Language Selection

Server Settings

Database Settings

Profile Settings

Admin Account

Database Settings - Standard Connection

Specify a JDBC driver and connection properties to connect to your database. If you need more information about this process please see the database documentation distributed with Openfire.

Note: Database scripts for most popular databases are included in the server distribution at [Openfire_HOME]/resources/database.

Database Driver Presets: MySQL

JDBC Driver Class: com.mysql.jdbc.Driver

Database URL: jdbc:mysql://localhost:3306/openfire

Username:

Password:

Minimum Connections: 5

Maximum Connections: 25

Connection Timeout: 1.0 Days

Note, it might take between 30-60 seconds to connect to your database.

Continue

The configuration wizard gives you the option to use either an embedded database or a standard database connection. Choose a standard database connection so that you can use your MySQL database.

Select MySQL from the **Database Driver Presets** list. Insert the name of your server and database into the **Database URL** field. For example, for a MySQL database named openfire set up on localhost, you'd enter:

```
jdbc:mysql://localhost:3306/openfire
```

On the next screen of the wizard, opt to store user accounts in the database. Enter the username and password for the database user you created earlier, and continue to the end of the configuration wizard. By this point you should have created a server administrator and set up the domain location for your XMPP server.

Plug-in for per-user notifications

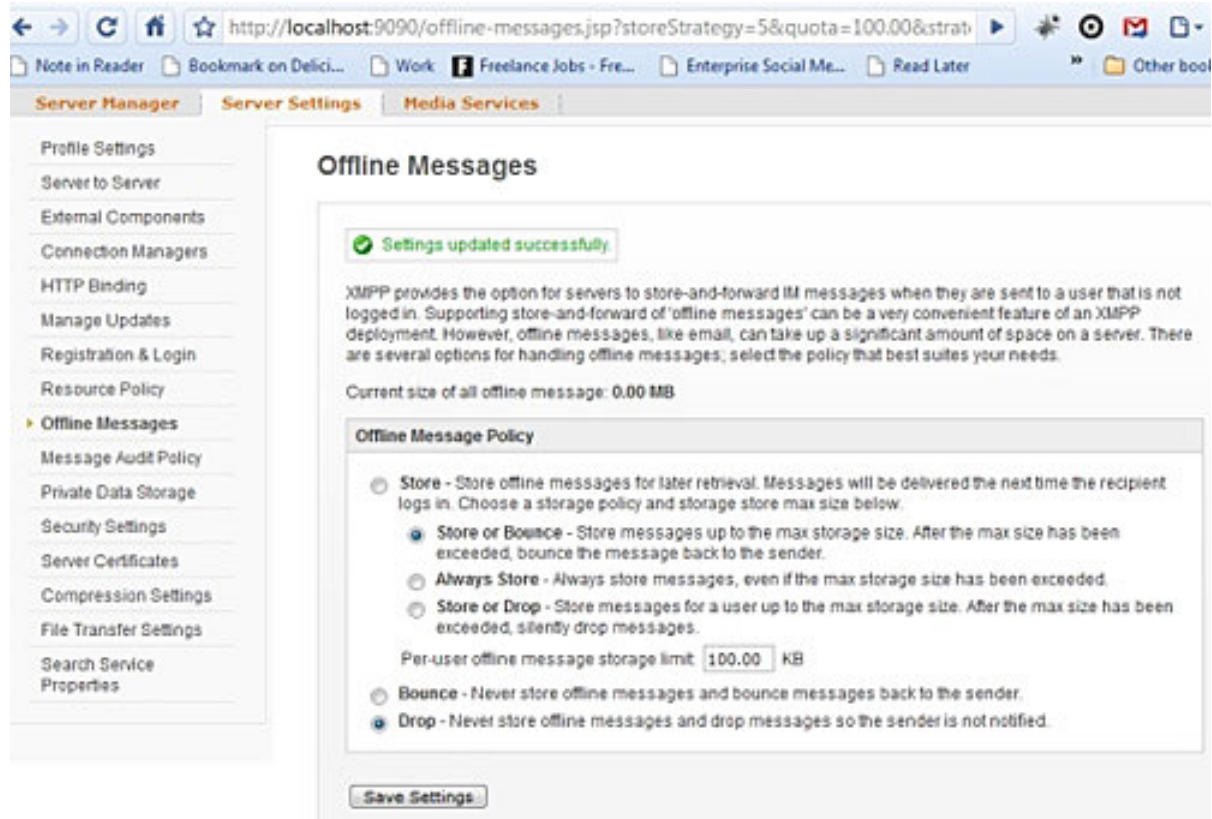
For the purposes of this demo, you'll simply use the two users you create here. However, if you want to support sophisticated per-user notifications in your applications, you need to be able to add and remove users from the PHP portion of your application programmatically. The User Service plug-in for Openfire gives you this capability through a REST interface for XMPP user administration. To install the plug-in, download it from the Openfire plug-ins site (see [Resources](#)). The plug-in itself is a single file: `userservice.jar`. You must place it in the `/plugins` directory of your Openfire installation.

Log into the admin screen using the administrator credentials you established. Click **Edit Properties** (below **Server ports**), and make a note of the listed server name. This will form the domain portion of your JIDs. It is not interchangeable; for example, you cannot use `localhost` in place of `127.0.0.1`, or vice versa.

Click on **Users/Groups** in the top navigation menu and create two new users. These will be your test users during development.

Click on **Server settings** and then **Offline messages**. Because you are using XMPP for interface notifications, set the **Offline Message Policy** to **Drop**, as in [Figure 4](#). You don't want to keep messages that are sent when users aren't logged in; otherwise they might be inundated with thousands of notifications when they return.

Figure 4. Dropping offline messages



In **Server settings**, click on **Server to Server**. For your purposes, you needn't be concerned with external servers, because you will not operate as a connected part of the wider XMPP network. Therefore, set **Service Enabled** to **Disabled**, and **Allowed to Connect** to **White List**. These settings will prevent unauthorized connections from causing havoc.

Configuring Apache to forward XMPP over BOSH

Openfire maintains an HTTP binding URL for access over BOSH at `http://localhost:7070/http-bind`. To use this on port 80, you must configure the Apache HTTP Server to forward a URL to this location. For this, you need to enable the proxy module.

Open your `http.conf` Apache configuration file and find the `LoadModule` entries for `mod_proxy.so` and `mod_proxy_http.so`, which are commented out by default. Uncomment them by removing the leading number sign (`#`) character. The appropriate lines in the Dynamic Shared Object (DSO) Support section of the configuration file (which aren't necessary adjacent to each other) should now resemble those [Listing 8](#):

Listing 8. Enabling proxy support in Apache HTTP Server

```
LoadModule proxy_http_module modules/mod_proxy_http.so
LoadModule proxy_module modules/mod_proxy.so
```

At the bottom of the configuration file, add the lines in [Listing 9](#) (substituting your server IP address for 127.0.0.1 if you're not using localhost as your test server environment):

Listing 9. XMPP proxy rule in httpd.conf

```
# XMPP proxy rule
ProxyRequests Off
ProxyPass /xmpp-httpbind http://127.0.0.1:7070/http-bind/
ProxyPassReverse /xmpp-httpbind http://127.0.0.1:7070/http-bind/
```

Notice in [Listing 9](#) that you use a slightly different URL, `/xmpp-httpbind`, on port 80. This URL is the value that `strophe.js`, the client-side JavaScript framework you'll use later, assigns to a variable that sets the BOSH endpoint.

Restart the server. You're finally ready to begin programming web applications using XMPP.

Section 5. Creating the server application using XMPPHP

In the previous sections, you set up the server and the plug-ins. In this section you will create the server-side portion of your real-time application.

Server-side functionality

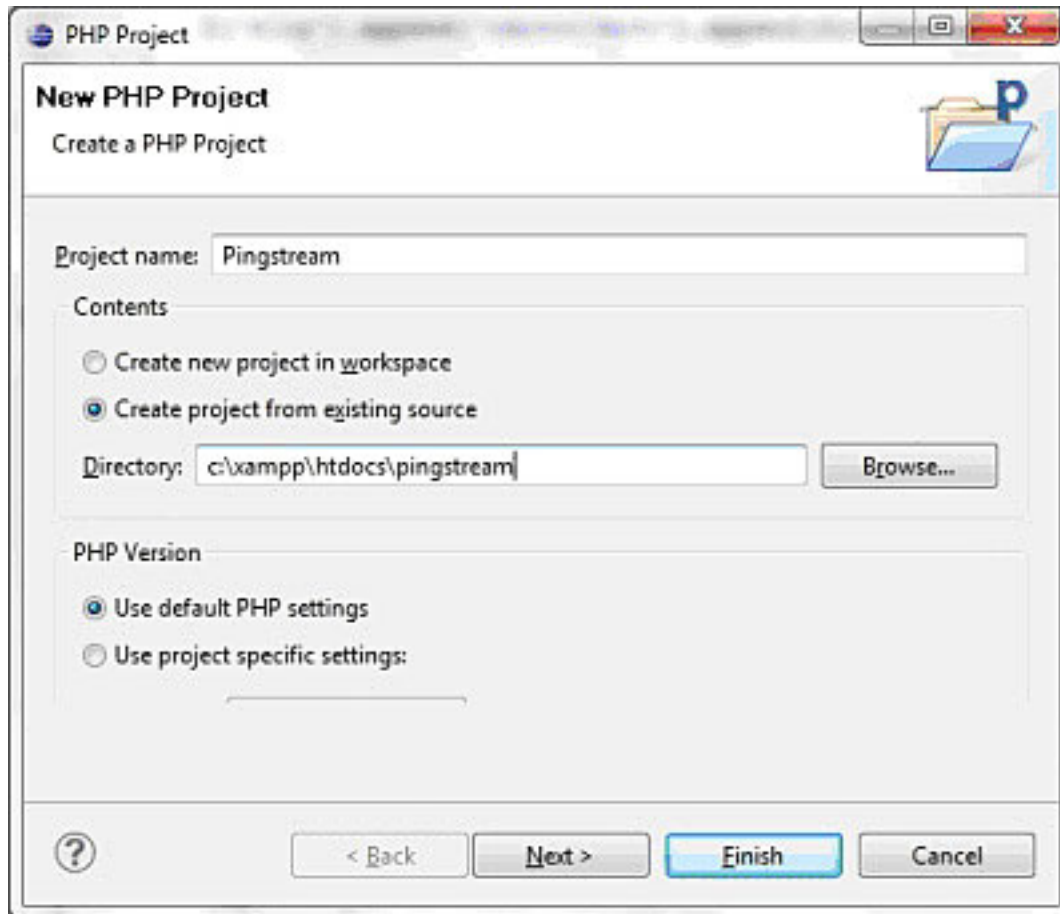
The PHP side of your application will perform two main tasks:

1. A PHP script will fetch an RSS feed and notify you if the first entry has changed since the last time you checked.
2. A front-end script will initialize your JavaScript XMPP client.

Start by creating a new PHP project space called `pingstream` within the web root of your testing server. If you use the PHP Development Tools project extensions for the

Eclipse development environment, take care to create the project within the web root rather than the default workspace for Eclipse (see [Figure 5](#)):

Figure 5. Creating a new Eclipse PHP project



[Figure 5](#) shows the following values for the Pingstream project:

- Project name field: Pingstream
- Contents radio button: Create project from existing source
- Directory field: c:\xampp\htdocs\pingstream
- PHP Version radio button: Use default PHP settings

XMPPHP is the most widely used XMPP library for PHP. Like ejabberd and Openfire, it's free and open source, so it serves as a good starting point for first-time XMPP developers.

Download XMPPHP (see [Resources](#)). Unpack the archive and insert it into the lib/xmpphp subfolder of your new project.

Finally, to save time parsing your RSS feed, download and install the Last RSS PHP parser (see [Resources](#)).

Storing server-side settings

You'll use one of the test users you created earlier as your sending party for notifications. Create a file called `config.inc.php` in the root of your `/pingstream` folder and add the contents of [Listing 10](#):

Listing 10. Server-side configuration settings

```
<?php

// Pingstream configuration file

// Define global $CONFIG array - don't change this!
global $CONFIG;
$CONFIG = array();
$CONFIG['send'] = new stdClass();

// Set account details for sending party
$CONFIG['send']->user = 'testuser'; // User portion of JID
$CONFIG['send']->host = '127.0.0.1'; // Host portion of JID
$CONFIG['send']->resc = 'pingstream'; // Resource portion of JID
$CONFIG['send']->pass = 'mypass'; // Password for user
```

The `config.inc.php` file also must include details about your receiving party, as in [Listing 11](#):

Listing 11. Details about receiving party

```
// Set the receiving account details
$CONFIG['receive'] = 'receivinguser@127.0.0.1';
```

You will cache a small amount of data in a file, so `config.inc.php` needs to include a writeable file path, as in [Listing 12](#):

Listing 12. Setting the location to store temporary files

```
// Full path to your cache directory, with trailing slash
$CONFIG['cachedir'] = '/tmp/';
```

If you're on a Microsoft® Windows® machine, you can set this to be a blank string. Otherwise, make sure the directory that you specify has a full path and a trailing slash, and is set to world-writable.

Defining the data source

Your Pingstream application will check the IBM developerWorks Web development zone feed for updates, so add the contents of [Listing 13](#) to config.inc.php:

Listing 13. Checking a developerWorks feed for updates

```
// Set the RSS feed you're going to check
$CONFIG['rss'] = 'http://www.ibm.com/developerworks/views/web/rss/libraryview.jsp';
```

Now, create another new file, /lib.inc.php. This will be your library file, included in the main controller page and user interface page of your application.

The top of lib.inc.php must reference config.inc.php, the main XMPPHP library, and lastRSS, as in [Listing 14](#):

Listing 14. Loading required libraries

```
<?php
// Load libraries
require_once('XMPPHP/XMPP.php');
require_once('config.inc.php');
require_once('lastRSS.php');
```

Next, you'll create a function to send a message to the client through XMPP. Doing this through XMPPHP is easy. You establish a connection using the credentials you saved in your configuration file, send the message, and close the connection.

In the first part of the function you create a new XMPPHP_XMPP object, as in [Listing 15](#):

Listing 15. Creating a new XMPP connection object

```
// Load configuration
global $CONFIG;

$conn = new XMPPHP_XMPP(
    $CONFIG['connect']->host,
    5222,
    $CONFIG['connect']->user,
    $CONFIG['connect']->pass,
    $CONFIG['connect']->resc);
```

Note that XMPPHP connects to your XMPP server over port 5222, the default port for XMPP communications. Although you need to use BOSH for your client-side communications, there is no requirement for you to do so on the server side.

To connect, you send an initial connection request, wait until you receive notification that your XMPP session has started, and send a presence stanza to announce that you are online (see [Listing 16](#)):

Listing 16. Establishing an XMPP connection

```
$conn->connect();
$conn->processUntil('session_start');
$conn->presence();
```

Next you send the message itself, which you have prefilled in a variable called `$message` (see [Listing 17](#)):

Listing 17. Sending the message

```
$conn->message($CONFIG['receive'], $message);
```

Then you disconnect using `$conn->disconnect();`.

During this process you might encounter errors. [Listing 18](#) puts the code in Listings [15](#), [16](#), and [17](#) into a function to insert into `lib.inc.php`. In the process it wraps the business of sending a message in a `try/catch` statement that writes any exception messages to the error log.

Listing 18. The complete `send_notification` function, including `try/catch` statement for logging errors

```
/**
 * Updates everyone's user interface with a message
 */
function send_notification($message) {

    // Load configuration
    global $CONFIG;

    $conn = new XMPPHP_XMPP(
        $CONFIG['connect']->host,
        5222,
        $CONFIG['connect']->user,
        $CONFIG['connect']->pass,
        $CONFIG['connect']->resc);

    try {
        $conn->connect();
        $conn->processUntil('session_start');
        $conn->presence();
        $conn->message($CONFIG['receive'], $message);
        $conn->disconnect();
    } catch(XMPPHP_Exception $e) {
        error_log($e->getMessage());
    }
}
```

You can use this simple mechanism for any number of public-notification applications. Here, you give all users who visit the site the same notifications, but it is a trivial matter to tailor notifications by the following mechanism:

1. Each user is given a unique session string, or a shared string specific to a particular interest or search.
2. This string is added to the JID of the receiving party as the resource segment
3. Messages are then sent to the `user@domain/session-string` JID

Obtaining external data

Next, you need a function, in [Listing 19](#), to retrieve the last RSS item from your designated feed:

Listing 19. Getting the most recent item in an RSS feed

```
function get_last_feed_item() {
    global $CONFIG;          // Load configuration
    $rss = new lastRSS;      // Initialize lastRSS
    $rss->CDATA = 'content';
    if ($rs = $rss->get($CONFIG['rss'])) {
        if (isset($rs['items'][0]))
            return $rs['items'][0];
        else
            return false;
    }
}
```

This function initializes `lastRSS`, loads the RSS feed that you configured (in your case, the latest article feed for the IBM developerWorks Web development zone), and returns the topmost entry as an array. Once you have the latest feed item, you need to know if it changed since last time you checked. You'll use a small text file for this purpose. For secure applications, it is better to use the database or another method, but for your testing purposes, it's safe to use a small file cache. The cache stores the URL of the latest feed item each time you check; if the URL changed, you have a new item and should notify the user. Another function, `feed_has_changed`, returns `true` or `false`, accordingly. Either way, it also saves the new latest URL in the cache file, ready for the next time it checks. [Listing 20](#) shows the `feed_has_changed` function:

Listing 20. Returning any new feed data

```
function feed_has_changed($url) {

    global $CONFIG;
    $changed = false;

    // Check to see if the file exists, and if it does, if
    // the URL has changed
    if (!file_exists($CONFIG['cachedir'] . 'cache.txt')) {
```

```

        $changed = true;
    } else if (file_get_contents($CONFIG['cachedir'] . 'cache.txt') != $url) {
        $changed = true;
    }

    // If the URL has indeed changed, update the version in the cache
    // and return true
    if ($changed) {
        file_put_contents($CONFIG['cachedir'] . 'cache.txt', $url);
        return true;
    }

    // Otherwise return false
    return false;
}

```

You'll pass a simple HTML-encoded version of the latest item to the client. In a more advanced application, you could encode the item in JavaScript Object Notation (JSON) or XML, including a little more metadata, and allow the client-side JavaScript to format it appropriately depending on the device and browser. For now, however, the version in [Listing 21](#) will do:

Listing 21. Encapsulating a feed item in simple HTML

```

function last_item_html($item) {
    return <<< END
    <div class="item">
        <div class="item_title">
            <h2><a href="{ $item['link'] }"
target="_blank">{ $item['title'] }</a></h2>
        </div>
        <div class="item_body">
            { $item['description'] }
        </div>
    </div>
END;
}

```

Finally, create a new PHP file in your /pingstream directory called backend.php. Use the simple PHP script in [Listing 22](#) as the file's contents. The script retrieves the RSS feed and sends a JSON-encoded version of the most recent feed item through XMPP to your receiving party.

Listing 22. Sending the encapsulated feed item over XMPP

```

<?php

require_once('lib.inc.php');

if ($lastitem = get_last_feed_item()) {
    if (feed_has_changed($lastitem['link'])) {
        send_notification(last_item_html($lastitem));
    }
}

```

This is all you need to send dynamic notifications to public users of the application. Ideally, you should run the backend.php script on a regular cron job. However, for testing purposes you can execute the script manually through a web browser.

Section 6. The browser application: Strophe.js and jQuery

In this section, you will write JavaScript functions in order to receive messages through XMPP over BOSH, and build an HTML user interface to display received notifications.

Creating the user interface

You now need to create the user interface to receive the notifications. Strophe.js is a commonly used JavaScript library for sending and receiving XMPP data over BOSH. For your purposes in Pingstream, you will simply be receiving data, although it's easy to see that two-way communication allows you to build rich collaborative environments quickly.

Although it comes in several flavors, the JavaScript version of Strophe is most useful to you as a browser-based XMPP client. Download the compressed package (see [Resources](#)), and extract it into a strophejs folder in the pingstream directory.

The jQuery JavaScript framework makes both event handling and DOM manipulation considerably simpler. The supplied Strophe.js examples use it extensively, and the two are a natural fit together. Download jQuery (see [Resources](#) for a link) and place the minified version in a jquery directory in pingstream.

Create a new index.html file. Include references to the Strophe and jQuery libraries you just downloaded, as well as a pingstream.js library that you'll define in a moment. In the `body` element, add a `div` element with ID notifications, as in [Listing 23](#):

Listing 23. Client HTML page

```
<!DOCTYPE html>
<html>
  <head>
    <title>Latest content</title>
    <script type="text/javascript"
src="jquery/jquery-1.4.2.min.js"></script>
    <script type="text/javascript"
```

```
src="strophejs/strophe.js"></script>
  <script type="text/javascript"
src="pingstream.js"></script>
</head>
<body>
  <h1>Latest content:</h1>
  <div id="notifications"></div>
</body>
</html>
```

Create the JavaScript file—`pingstream.js`—you just referenced in [Listing 23](#). At the top of `pingstream.js`, define the BOSH proxy endpoint that you configured in Apache earlier, as in [Listing 24](#):

Listing 24. Setting the BOSH endpoint

```
var BOSH_SERVICE = '/xmpp-httpbind';
var connection = null;
```

When the page fully loads, you want to connect automatically to the XMPP server. You can do this using jQuery's `$(document).ready` call; inside, you create a new `strophe.js Strophe.Connection` object and use it to connect to the server, as in [Listing 25](#):

Listing 25. Establishing a connection over BOSH

```
$(document).ready(function () {
  connection = new Strophe.Connection(BOSH_SERVICE);
  connection.connect(
    "sendinguser@127.0.0.1",
    "sendingpass",
    onConnect);
});
```

More-robust options

For the purposes of this tutorial, you are using the sending party you defined earlier. For a more robust application, it might be better to create a new user for each registered application user and subscribe each one to a publish-subscribe interface. Alternatively, Strophe.js can log in anonymously if you leave the username and password blank and the XMPP server is configured to accept this type of connection. In these cases, a JID is created dynamically for each anonymous user; these must be managed. Finally, an extension for an XMPP chat room is also available.

In [Listing 25](#), the `Strophe.Connection.connect` method contains a reference to an `onConnect` function as one of its parameters. `onConnect` is launched once a connection is established. This is your opportunity to add a notification handler for incoming messages; here you register a function called `notifyUser`. Following this, you send a simple presence stanza.

To confirm that you can connect and receive new messages, also send a friendly notification to the user.

Add the code in [Listing 26](#) above the `$(document).ready` call in your JavaScript file:

Listing 26. Handling incoming XMPP messages

```
function onConnect(status)
{
    $('#notifications').html('<p class="welcome">Hello!
Any new posts will appear below.</p>');
    connection.addHandler(notifyUser, null, 'message', null, null, null);
    connection.send($pres().tree());
}
```

Finally, because you registered the notification handler, Strophe.js calls the `notifyUser(msg)` function whenever the XMPP client receives message stanza. The `msg` parameter is a representation of the XML stanza itself, which can be queried as in [Listing 27](#):

Listing 27. Querying the msg parameter

```
var elems = msg.getElementsByTagName('body');
var body = elems[0];
$('#notifications').append(Strophe.getText(body));
```

Ideally, you want to limit messages so they only display if they were sent from your server-side sending user. You can wrap it all in an `if` statement that forms the body of the `notifyUser` function, in [Listing 28](#):

Listing 28. notifyUser function

```
function notifyUser(msg)
{
    if (msg.getAttribute('from') == "testuser@127.0.0.1/pingstream") {
        var elems = msg.getElementsByTagName('body');
        var body = elems[0];
        $('#notifications').append(Strophe.getText(body));
    }
    return true;
}
```

This needs to sit above the `onConnect` function defined in [Listing 26](#).

Putting it all together

Open your `index.html` file in a web browser. You should see a simple heading and a

message saying that updates will appear below (which you might recall was your test notification to yourself that the XMPP connection was working successfully).

Now load backend.php. As if by magic, the latest update from the IBM developerWorks Web development zone is showing up on your page. Other example sources with RSS feeds include Twitter accounts, news agencies, and update feeds from automatic server-monitoring software.

This is a simple start to developing with a powerful platform. Strophe.js can facilitate two-way communication with the application, although it might be simpler to use standard jQuery HTTP callbacks to get user input into the system, to obviate the necessity for programming an XMPP monitoring daemon for your application. More excitingly, when you use the web server as a BOSH proxy, two or more web clients can speak to one another over XMPP without much input from the server-side web application at all. The implications for everything from office collaboration software to games are huge.

Section 7. Conclusion

This tutorial discussed the need for real-time web applications and how XMPP overcomes the shortcomings of existing technologies. To demonstrate the effectiveness of this approach, you developed a simple RSS update notification application using XMPPHP, Last RSS, Strophe.js, Openfire, and PHP.

Although it requires an added server layer and some new JavaScript techniques, XMPP is a much better fit for real-time web applications than the traditional Ajax polling model. It's faster, requires much less overhead in terms of both development and system infrastructure, and uses a powerful emerging standard for web development.

Downloads

Description	Name	Size	Download method
Pingstream source code	pingstream.zip	238KB	HTTP

[Information about download methods](#)

Resources

Learn

- [XMPP Standards Foundation](#): Visit the official XMPP site.
- [Meet the Extensible Messaging and Presence Protocol \(XMPP\)](#) (M. Tim Jones, developerWorks, September 2009): Discover the ins and outs of XMPP, and learn how to use it for simple messaging.
- [Create an alerts system using XMPP, SMS, pureXML, and PHP](#) (Joe Lennon, developerWorks, November 2009): Try using XMPP to send notifications to Google Talk.
- [Jabber](#) (Gerhard Poul, developerWorks, May 2002): Check out this early look at XMPP.
- [Ajax and XML: Ajax for chat](#) (Jack D Herrington, developerWorks, December 2007): See how to implement a real-time web application using Ajax polling.
- [JavaScript Tutorial](#): Learn how to use the scripting language of the web.
- [Ejabberd](#): Learn more about this XMPP server, written in Erlang.
- [My developerWorks](#): Personalize your developerWorks experience.
- [IBM XML certification](#): Find out how you can become an IBM-Certified Developer in XML and related technologies.
- [XML technical library](#): See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- [developerWorks technical events and webcasts](#): Stay current with technology in these sessions.
- [developerWorks on Twitter](#): Join today to follow developerWorks' tweets.
- [developerWorks podcasts](#): Listen to interesting interviews and discussions for software developers.

Get products and technologies

- [Openfire](#): Download Openfire, a cross-platform real-time collaboration server based on the XMPP (Jabber) protocol.
- [PHP](#): Visit the PHP site and get this widely-used scripting language that is well-suited for Web development and can be embedded into HTML. This tutorial uses PHP 5.2 or higher.
- [Apache HTTP Server](#): Download the Apache web server.
- [MySQL](#): Download an open source transactional database.

- [DB2 Express-C](#): Get a free version of the IBM DB2 database server, an excellent foundation for application development for small and medium business.
- [Openfire plug-in](#): Get the User Service plug-in for Openfire.
- [XMPPHP](#): Download the PHP XMPP Library from the project's Google Code site.
- [Last RSS](#): Get Vojtech Semecky's RSS parser for PHP (available under the GNU Public License version 2).
- [jQuery](#): Download the jQuery JavaScript library, available under either the MIT or GNU Public License.
- [Strophe](#): Download Strophe.js, a family of libraries for writing XMPP clients. Its license allows it to be used, modified, and shared freely.
- [phpMyAdmin](#): Get this free software tool that lets you administer MySQL over the web.
- [IBM product evaluation versions](#): Download or [explore the online trials in the IBM SOA Sandbox](#) and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- [XMPP Discussion](#): Subscribe to XMPP discussions for developers, system administrators, and users.
- [XML zone discussion forums](#): Participate in any of several XML-related discussions.
- [developerWorks blogs](#): Check out these blogs and get involved.

About the author

Ben Werdmuller

Ben Werdmuller is a web strategist and developer who has specialized in open source platforms for more than 10 years. He co-founded and was the technical lead for Elgg, an open source social-networking framework. Ben blogs regularly at <http://benwerd.com/>.

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. See the current list of [IBM trademarks](#).

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.