14

Writing Strophe Plug-ins

WHAT'S IN THIS CHAPTER?

- Using Strophe plug-ins
- Creating plug-ins
- Adding namespaces to Strophe
- Building a simple roster plug-in

The applications in this book were all written using Strophe's primitive functions to build and send XMPP stanzas. Developing the code this way provided valuable visibility into how the various extensions of XMPP work. In real applications, however, it is nice to develop some abstractions to reduce some of the grunt work. Strophe allows users to build and load plug-ins that extend its functionality so that such abstractions can be built by developers and used by applications.

As an example, recall that in Chapter 9 you built SketchCast, which broadcast drawing events to a pubsub node for subscribers to watch. The application's code created subscription stanzas, node creation stanzas, and configuration stanzas to accomplish its tasks. Imagine if there was a pubsub plug-in for Strophe that encapsulated this logic for you. Code using this plug-in might appear as follows:

```
SketchCast.connection.pubsub.create(
    SketchCast.node, SketchCast.created, SketchCast.create_error);

SketchCast.connection.pubsub.configure(
    SketchCast.node, {"max_items": 20},
    SketchCast.configured, SketchCast.configure_error);
```

This version of the code is no less correct, but much shorter and more clear. The details of how to build the various stanzas required for the operations are hidden behind the plug-in's interface, leaving you to think about your own application's logic rather than protocol details.

Several Strophe plug-ins are already available on the Strophe web site, and more are created by its users all the time. Before you start your next XMPP project, you might want to take a look at the available plug-ins to see if there is one that meets your application's needs. You should also feel free to contribute plug-ins to the community as well!

Plug-ins are a great way to create running code faster, by leveraging the work of others. They also enable you to modularize your own code, by separating application logic from the underlying protocol semantics. In this chapter, you see exactly how to use existing plug-ins and create ones of your own.

USING PLUG-INS

Using Strophe plug-ins is as easy as loading the plug-in code and then accessing the new functionality through a `Strophe.Connection` instance. Each plug-in automatically installs itself into the Strophe library when loaded and then becomes available as an attribute of the connection whenever a new connection is created.

Loading Plug-ins

Strophe plug-ins must be loaded after the Strophe library. The following HTML code loads the Strophe pubsub plug-in, included with the Strophe library, after the main Strophe code:

```
<script src='strophe.js'></script>
<script src='strophe.pubsub.js'></script>
```

By convention, plug-ins are named `strophe.plugin-name.js`. The plug-in injects itself into the Strophe library automatically once loaded (you see how this is done in the "Creating Plug-ins" section). If you are familiar with jQuery plug-ins, this should look very similar.

Accessing Plug-in Functionality

Once plug-ins are loaded, they become available to all `Strophe.Connection` object instances. If you load a plug-in from the file `strophe.myplugin.js`, the plug-in's interface will be accessible via the `myplugin` attribute on the connection object. If you load the Strophe pubsub as in the previous example, you would access it like this:

```
var connection = new Strophe.Connection(BOSH_URL);
connection.connect(. . .);
. . .
connection.pubsub.subscribe("pubsub.pemberly.lit", "latest_books",
                            function (iq) { . . . },
                            function (iq) { . . . });
```

Other pubsub plug-in functionality would be accessed the same way.

In addition to providing new functions, plug-ins can also add new namespaces to Strophe. These namespaces are accessible alongside the built-in ones found in the `Strophe.NS` object. For example, the Strophe pubsub plug-in adds many pubsub-related namespaces to Strophe, a few of which are:

- `Strophe.NS.PUBSUB`, which expands to `http://jabber.org/protocol/pubsub`
- `Strophe.NS.PUBSUB_OWNER`, which expands to `http://jabber.org/protocol/pubsub#owner`
- `Strophe.NS.PUBSUB_EVENT`, which expands to `http://jabber.org/protocol/pubsub#event`

These namespaces make it very convenient to build pubsub stanzas yourself if for some reason the plug-in doesn't provide the functionality you want directly. They will also save you some typing and some searching through the specifications.

Each plug-in provides different functions, but these functions are always accessible in the same way. If you want to learn more about a particular plug-in and the operations it provides, please see its documentation. Even if the plug-in is sparsely documented or not documented at all, the tools you have learned throughout this book should make it easy to understand all but the most complicated plug-ins. After all, most plug-ins are just convenience wrappers around common protocol operations.

Now that you know how to load and use plug-ins, you should read on to learn how to build your own!

BUILDING PLUG-INS

Plug-ins are created by making a prototype object that defines the plug-in's functionality and registering that prototype with the Strophe library. When a new Strophe connection is created, it creates a plug-in object from the plug-in prototype and calls the object's `init()` function to initialize the plug-in. After initialization, the plug-in is ready to be used by your application code.

The `init()` function also serves another purpose. Most plug-ins will need to send data over the XMPP connection and set up stanza handlers, and for these tasks, the plug-in needs access to a `Strophe.Connection` object. Strophe will pass the connection object as the first parameter to `init()`, and the plug-in can then save this reference for later use.

Another function common to most plug-ins is `statusChanged()`. Strophe calls each plug-in's `statusChanged()` function whenever the connection status changes. This is exactly the same as the connection callback you've seen in many of the previous chapters. This allows the plug-in to react to events like `CONNECTED` and `DISCONNECTED`. This function is optional, because some plug-ins only need to respond to direct invocation.

The example plug-in in Listing 14-1 is one of the smallest Strophe plug-ins you can make. Aside from the `init()` function discussed earlier, it also provides `online()` and `offline()` functions that change your presence.

LISTING 14-1: strophe.simple.js

```
Strophe.addConnectionPlugin('simple', {
    init: function (connection) {
        this.connection = connection;
    },

    online: function () {
        this.connection.send($pres());
    },

    offline: function () {
        this.connection.send($pres({type: "unavailable"}));
    }
});
```

Registering new Strophe plug-ins is accomplished via the `addConnectionPlugin()` function. This function takes the name of the plug-in and the plug-in prototype as arguments. These plug-ins are called connection plug-ins because they augment the `Strophe.Connection` object. In the future, Strophe may support other plug-in types that augment other parts of the library.

Finally, Strophe plug-ins can augment the namespaces available in the `Strophe.NS` object by using `.addNamespace()`. The following code adds service discovery namespaces to Strophe. The simple example plug-in has no new namespaces, but more complex plug-ins will often need to add to the namespaces for the convenience of the plug-in's users.

```
Strophe.addNamespace('DISCO_INFO', 'http://jabber.org/protocol/disco#info');
Strophe.addNamespace('DISCO_ITEMS', 'http://jabber.org/protocol/disco#items');
```

Users of the preceding example plug-in can write `connection.simple.online()` to send available presence and `connection.simple.offline()` to send unavailable presence instead of building the stanzas by hand. In this particular case, this doesn't save much effort, but as you soon see, more sophisticated plug-ins can make your XMPP programming tasks far simpler.

CREATING A ROSTER PLUG-IN

To really get a feel for what plug-ins can do, you will have to build one that has some non-trivial functionality. In this section, you develop a roster management plug-in, which abstracts away common roster operations and makes roster data available in a JavaScript-friendly data structure.

Managing the roster involves a few basic operations. First, you'll need to query the roster and store it. Then you'll need ways to add, edit, and delete roster items. Because other connection resources might also be making roster changes, you'll need to listen for those and update the roster accordingly. Finally, you will want to keep the status of your roster fresh as presence information comes in from your contacts.

Storing Contacts

Before starting on roster mutation operations, you should begin with how to store the roster's state in the plug-in. This will be the main interaction point between various pieces of the code.

Imagine that you've created an application that shows a user's roster in the user interface. To do this, your code could query the roster and add handlers for roster updates. When another piece of your application needs the roster data, you can either query for the roster again, which is inefficient, or have one part of the UI communicate with the other, which tightly couples the functionality. Instead of doing either of those things, it is better to put knowledge of the roster state in one place where the various parts of your application can interact with it.

Following is a sample roster expressed as a JavaScript literal; this same structure will be the basis of the roster state in your plug-in.

```
contacts = {
    "darcy@pemberley.lit": {
        name: "Darcy",
        resources: {
            "library":, {
                show: "away",
                status: "reading"
            }
        },
        subscription: "both",
        ask: "",
        groups: ["Family"]
    },
    "bingley@netherfield.lit": {
        name: "Charles",
        resources: {},
        subscription: "both",
        ask: "",
        groups: ["Friends"]
    }
};
```

The sample roster contains two contacts, Darcy and Bingley, who are online and offline, respectively. The `resources` attribute tells you which resources of the contacts are currently online along with their appropriate metadata; because Darcy has at least one resource, he is considered online, and because Bingley has none, he is considered offline.

The `name`, `subscription`, and `ask` properties of the contact and the `show` and `status` properties of the resource are named for the protocol attributes they represent. These attributes were covered in detail back in Chapter 6. The `show` and `status` attributes come from contacts' `<presence>` stanzas, and `name`, `subscription`, and `ask` are part of the user's roster state for a given contact.

We've simplified things slightly, however, with respect to the `show` and `status` attributes. When a contact's presence has no `<show>` element, they are considered online and available, so the `show` attribute is set to "available." Similarly, if the contact's presence has no `<status>` element, the `status` attribute is set to the empty string.

Finally, the `groups` attribute specifies which roster groups a contact belongs in.

It would be quite tedious for users of your plug-in to have to write code to enumerate the `resources` property to determine if a contact was online. You can make these programmers' lives easier by providing a utility method that computes this property based on the `resources` attribute's value:


```
online: function () {
    var result = false;
    for (var k in this.resources) {
        result = true;
        break;
    }
    return result;
}
```

code snippet strophe.roster.js

Now you have enough pieces to make a first outline of the plug-in code. The following version isn't very useful yet, but you will be extending it over the next sections. Add the following code to a new file called `strophe.roster.js`:


```
// Contact object
function Contact() {
    this.name = "";
    this.resources = {};
    this.subscription = "none";
    this.ask = "";
    this.groups = [];
}

Contact.prototype = {
    // compute whether user is online from their
    // list of resources
    online: function () {
        var result = false;
        for (var k in this.resources) {
            result = true;
            break;
        }
        return result;
    }
};

Strophe.addConnectionPlugin('roster', {
    init: function (connection) {
        this.connection = connection;
        this.contacts = {};

        Strophe.addNamespace('ROSTER', 'jabber:iq:roster');
    }
});
```

code snippet strophe.roster.js

Getting and Maintaining the Roster

You saw how to retrieve and deal with the roster in Chapter 6, and now you will put those skills to use again to populate the roster state in the plug-in. First, you must get an initial copy of the roster from the server once the connection is established. After that, each contact's information will need to be updated as their presence changes. The plug-in must also trigger an event so that the user's code gets notified when new roster changes have been made.

In Chapter 6, you learned how to retrieve rosters from the server. The plug-in must do this every time the connection is established. After all, while a user is disconnected in one client, another client could have made changes. Also, when the plug-in is notified about disconnection, it needs to put the roster representation into an appropriate state.

The following code implements the `statusChanged()` function to handle these tasks for the `CONNECTED` and `DISCONNECTED` statuses. It also notifies any handlers for the `roster_changed` event that the roster has been updated. You should insert this into the plug-in's prototype after `init()`.


```
statusChanged: function (status) {
    if (status === Strophe.Status.CONNECTED) {
        this.contacts = {};

        // build and send initial roster query
        var roster_iq = $iq({type: "get"})
            .c('query', {xmlns: Strophe.NS.ROSTER});

        var that = this;
        this.connection.sendIQ(roster_iq, function (iq) {
            $(iq).find("item").each(function () {
                // build a new contact and add it to the roster
                var contact = new Contact();
                contact.name = $(this).attr('name') || "";
                contact.subscription = $(this).attr('subscription') ||
                    "none";
                contact.ask = $(this).attr('ask') || "";
                $(this).find("group").each(function () {
                    contact.groups.push(this.text());
                });
                that.contacts[$(this).attr('jid')] = contact;
            });

            // let user code know something happened
            $(document).trigger('roster_changed', that);
        });
    } else if (status === Strophe.Status.DISCONNECTED) {
        // set all users offline
        for (var contact in this.contacts) {
            this.contacts[contact].resources = {};
        }

        // notify user code
        $(document).trigger('roster_changed', this);
    }
}
```

code snippet strophe.roster.js

The plug-in will now keep track of the basic roster information, but it doesn't keep this information up to date as the roster changes or as contacts change their presence. You can solve the first issue by setting up stanza event handlers for IQ-set stanzas that contain roster updates from the server. The second issue can be solved by adding presence stanza handlers.

In Chapter 6, you discovered how to modify rosters by adding, updating, and deleting contacts. The server also notifies other connected resources about roster changes so that every client's state remains consistent. For example, if Darcy removes Wickham from his roster while connected on the resource `library`, his other resource, `drawing_room`, will be notified of the change. First, Darcy deletes Wickham from his roster:

```
<iq from='darcy@pemberley.lit/library'
    type='set'
    id='delete1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='wickham@militia.lit' subscription='remove'/>
  </query>
</iq>
```

The server deletes Wickham and notifies all of Darcy's connected resources about the roster change. Darcy's `drawing_room` resource will receive:

```
<iq to='darcy@pemberley.lit/drawing_room'
    type='set'
    id='deleted1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='wickham@militia.lit' subscription='remove'/>
  </query>
</iq>
```

Darcy's `library` resource will also receive the same update, even though it requested the change. The server will always notify all resources of roster state changes.

```
<iq to='darcy@pemberley.lit/library'
    type='set'
    id='deleted1'>
  <query xmlns='jabber:iq:roster'>
    <item jid='wickham@militia.lit' subscription='remove'/>
  </query>
</iq>
```

Both clients must acknowledge the IQ-set stanza with an IQ-result. At this point, each client will have the same view of the roster state.

You can extend `statusChanged()` to set up handlers for roster changes and add a new function to the plug-in prototype to handle these changes. A modified version of `statusChanged()` is shown in the following code, with the new lines highlighted, followed by the new `rosterChanged()` function. Because the server will always send a single change at a time, `rosterChanged()` only needs to handle one `<item/>` child.


```
statusChanged: function (status) {
    if (status === Strophe.Status.CONNECTED) {
        this.contacts = {};

        this.connection.addHandler(this.rosterChanged.bind(this),
                                   Strophe.NS.ROSTER, "iq", "set");

        // build and send initial roster query
        var roster_iq = $iq({type: "get"})
            .c('query', {xmlns: Strophe.NS.ROSTER});

        var that = this;
        this.connection.sendIQ(roster_iq, function (iq) {
            $(iq).find("item").each(function () {
                // build a new contact and add it to the roster
                var contact = new Contact();
                contact.name = $(this).attr('name') || "";
                contact.subscription = $(this).attr('subscription') ||
                    "none";
                contact.ask = $(this).attr('ask') || "";
                $(this).find("group").each(function () {
                    contact.groups.push(this.text());
                });
                that.contacts[$(this).attr('jid')] = contact;
            });

            // let user code know something happened
            $(document).trigger('roster_changed', that);
        });
    } else if (status === Strophe.Status.DISCONNECTED) {
        // set all users offline
        for (var contact in this.contacts) {
            this.contacts[contact].resources = {};
        }

        // notify user code
        $(document).trigger('roster_changed', this);
    }
},

rosterChanged: function (iq) {
    var item = $(iq).find('item');
    var jid = item.attr('jid');
    var subscription = item.attr('subscription') || "";

    if (subscription === "remove") {
        // removing contact from roster
        delete this.contacts[jid];
    } else if (subscription === "none") {
        // adding contact to roster
        var contact = new Contact();
        contact.name = item.attr('name') || "";
```

```
            item.find("group").each(function () {
                contact.groups.push(this.text());
            });
            this.contacts[jid] = contact;
        } else {
            // modifying contact on roster
            var contact = this.contacts[jid];
            contact.name = item.attr('name') || contact.name;
            contact.subscription = subscription || contact.subscription;
            contact.ask = item.attr('ask') || contact.ask;
            contact.groups = [];
            item.find("group").each(function () {
                contact.groups.push($(this).text());
            });
        }

        // acknowledge receipt
        this.connection.send($iq({type: "result", id: $(iq).attr('id')}));

        // notify user code of roster changes
        $(document).trigger("roster_changed", this);

        return true;
    }
```

code snippet strophe.roster.js

With the roster updates handled, you can now move on to handling presence updates. You'll need to add a presence stanza handler to `statusChanged()` as well as a new function in the plug-in's prototype, `presenceChanged()`. The `presenceChanged()` function just needs to add, modify, or delete the appropriate resource in the `resources` attribute for a contact based on the presence stanza information.

Add the following line just after the `addHandler()` line from the preceding version of `statusChanged()`:


```
this.connection.addHandler(this.presenceChanged.bind(this),
                           null, "presence");
```

code snippet strophe.roster.js

Now add the implementation of `presenceChanged()` to the prototype:


```
presenceChanged: function (presence) {
    var from = $(presence).attr("from");
    var jid = Strophe.getBareJidFromJid(from);
    var resource = Strophe.getResourceFromJid(from);
    var ptype = $(presence).attr("type") || "available";

    if (!this.contacts[jid] || ptype === "error") {
        // ignore presence updates from things not on the roster
        // as well as error presence
```

```
            return true;
        }

        if (ptype === "unavailable") {
            // remove resource, contact went offline
            delete this.contacts[jid].resources[resource];
        } else {
            // contact came online or changed status
            this.contacts[jid].resources[resource] = {
                show: $(presence).find("show").text() || "online",
                status: $(presence).find("status").text()
            };
        }

        // notify user code of roster changes
        $(document).trigger("roster_changed", this);
    }
```

code snippet strophe.roster.js

The plug-in will now maintain roster state over the lifetime of the connection. Plug-in users can access this information at any time with `connection.roster.contacts`. All that remains is to assist the developer with making roster changes.

Manipulating the Roster

In the previous section you handled notifications for the three roster manipulations: add, modify, and delete a contact. Here you implement the other side of this functionality by allowing the plug-in's users to change the roster with simple calls to `addContact()`, `deleteContact()`, and `modifyContact()`. There are also two other times when the roster changes; contacts are added whenever a presence subscription is requested, and contacts are usually deleted whenever a user unsubscribes from a contact's presence. You'll make subscribing and unsubscribing just as easy as the other roster modifications with `subscribe()` and `unsubscribe()`.

The direct roster manipulation helpers are very simple. Because the plug-in already handles the change notifications, there is no need for your code to modify the `contacts` attribute at all. Once the server has processed the roster modification, it generates a change notification, and that notification will trigger the handlers you wrote that already do the appropriate modifications of the `contacts` attribute. You only need to send the appropriate IQ-sets to the server to initiate the chain of events. On top of that, roster item modification is exactly the same as addition, so you can reuse the same code!

All three functions are implemented in the following code. The code is very similar to the code you wrote in Chapter 6 to accomplish the same operations.


```
    addContact: function (jid, name, groups) {
        var iq = $iq({type: "set"})
            .c("query", {xmlns: Strophe.NS.ROSTER})
            .c("item", {name: name || "", jid: jid});
        if (groups && groups.length > 0) {
```

```
            $.each(groups, function () {
                iq.c("group").t(this).up();
            });
        }
        this.connection.sendIQ(iq);
    },

    deleteContact: function (jid) {
        var iq = $iq({type: "set"})
            .c("query", {xmlns: Strophe.NS.ROSTER})
            .c("item", {jid: jid, subscription: "remove"});
        this.connection.sendIQ(iq);
    },

    modifyContact: function (jid, name, groups) {
        this.addContact(jid, name, groups);
    }
```

code snippet strophe.roster.js

Recall that subscribing and unsubscribing to someone's presence is a two-step process. For subscriptions, the client first adds a new contact, and then sends a presence subscription request. For unsubscribe requests, the client first sends the correct presence stanza, and then deletes the contact from the roster. The following code makes use of the roster manipulation functions you just wrote, and should be added to the plug-in prototype:


```
    subscribe: function (jid, name, groups) {
        this.addContact(jid, name, groups);

        var presence = $pres({to: jid, "type": "subscribe"});
        this.connection.send(presence);
    },

    unsubscribe: function (jid) {
        var presence = $pres({to: jid, "type": "unsubscribe"});
        this.connection.send(presence);

        this.deleteContact(jid);
    }
```

code snippet strophe.roster.js

Your shiny, new roster plug-in is now complete. It's time to see what it can do!

TAKING THE PLUG-IN FOR A SPIN

Strophe plug-ins are intended to make developers' lives easier than dealing with the messy details directly. Even the simple plug-in you developed in the previous section is surprisingly useful. To

demonstrate this, you build a small mini-application that shows your current roster state and updates even as other resources are manipulating your contacts.

The HTML code and CSS styles for the RosterWatch application are shown in Listing 14-2 and Listing 14-3. There is nothing here that you haven't seen before. The main thing to notice in the HTML file is that the plug-in file `strophe.roster.js` is loaded as well as the Strophe library.

Available for download on Wrox.com

LISTING 14-2: rosterwatch.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
          "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html;charset=UTF-8">
    <title>RosterWatch - Chapter 14</title>

    <link rel='stylesheet' href='http://ajax.googleapis.com/ajax/libs/jqueryu
i/1.7.2/themes/cupertino/jquery-ui.css'>
    <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js'>
    </script>
    <script src='http://ajax.googleapis.com/ajax/libs/jqueryui/1.7.2/jquery-u
i.js'></script>
    <script src='scripts/strophe.js'></script>
    <script src='scripts/flXHR.js'></script>
    <script src='scripts/strophe.flxhr.js'></script>

    <script src='strophe.roster.js'></script>

    <link rel='stylesheet' type='text/css' href='rosterwatch.css'>
    <script src='rosterwatch.js'></script>
  </head>
  <body>
    <h1>RosterWatch</h1>

    <div class='toolbar'>
      <input id='disconnect' type='button' value='disconnect'
             disabled='disabled'>
    </div>

    <div id='roster'>
    </div>

    <!-- login dialog -->
    <div id='login_dialog' class='hidden'>
      <label>JID:</label><input type='text' id='jid'>
      <label>Password:</label><input type='password' id='password'>
    </div>
  </body>
</html>
```

LISTING 14-3: rosterwatch.css

```
body {
    font-family: Helvetica;
}

h1 {
    text-align: center;
}

.toolbar {
    text-align: center;
}

#roster {
    width: 500px;
    margin: auto;
    border: solid 1px black;
}

.hidden {
    display: none;
}

.contact {
    padding: 10px;
}

.name {
    font-size: 150%;
    font-weight: bold;
}

.jid {
    font-size: 80%;
    font-style: italic;
}

.online {
    background-color: #7f7;
}

.away {
    background-color: #f77;
}

.offline {
    background-color: #777;
}
```

The actual JavaScript code for this application is quite simple and appears in Listing 14-4. Because all of the heavy lifting dealing with roster updates is handled by the plug-in, the code deals very little with XMPP except to make the connection and send initial presence. The `roster_changed` event handler just enumerates the roster contacts and writes out HTML, just like any modern, dynamic web application.

LISTING 14-4: rosterwatch.js

```
RosterWatch = {
    connection: null
};

$(document).ready(function () {
    $('#login_dialog').dialog({
        autoOpen: true,
        draggable: false,
        model: true,
        title: 'Connect to XMPP',
        buttons: {
            "Connect": function () {
                $(document).trigger('connect', {
                    jid: $('#jid').val(),
                    password: $('#password').val()
                });

                $('#password').val('');
                $(this).dialog('close');
            }
        }
    });

    $('#disconnect').click(function () {
        $('#disconnect').attr('disabled', 'disabled');
        RosterWatch.connection.disconnect();
    });
});

$(document).bind('connect', function (ev, data) {
    var conn = new Strophe.Connection(
        'http://bosh.metajack.im:5280/xmpp-httpbind');

    conn.connect(data.jid, data.password, function (status) {
        if (status === Strophe.Status.CONNECTED) {
            $(document).trigger('connected');
        } else if (status === Strophe.Status.DISCONNECTED) {
            $(document).trigger('disconnected');
        }
    });

    RosterWatch.connection = conn;
});

$(document).bind('connected', function () {
    $('#disconnect').removeAttr('disabled');

    RosterWatch.connection.send($pres());
});

$(document).bind('disconnected', function () {
```

continues

LISTING 14-4 *(continued)*

```
    RosterWatch.connection = null;

    $('#roster').empty();
    $('#login_dialog').dialog('open');
});

$(document).bind('roster_changed', function (ev, roster) {
    $('#roster').empty();

    var empty = true;
    $.each(roster.contacts, function (jid) {
        empty = false;

        var status = "offline";
        if (this.online()) {
            var away = true;
            for (var k in this.resources) {
                if (this.resources[k].show === "online") {
                    away = false;
                }
            }
            status = away ? "away": "online";
        }

        var html = [];
        html.push("<div class='contact " + status + "'>");

        html.push("<div class='name'>");
        html.push(this.name || jid);
        html.push("</div>");

        html.push("<div class='jid'>");
        html.push(jid);
        html.push("</div>");

        html.push("</div>");

        $('#roster').append(html.join(''));
    });

    if (empty) {
        $('#roster').append("<i>No contacts:(</i>");
    }
});
```

The roster plug-in you created does an excellent job abstracting away the details of roster management, which is exactly what was intended.

Improving the Roster Plug-in

The roster plug-in you created is pretty simple, although it is still quite useful. Here are some ideas for improvements you can try:

- Send more detailed update information in the `roster_changed` event so that the user's code doesn't have to refresh the entire roster every time.
- Extend the plug-in to trigger more events for things like incoming presence subscriptions and contacts going on and offline.
- Add support for Personal Eventing Protocol (XEP-0163) and Entity Capabilities (XEP-0115) to get extended information, such as what music contacts are listening to.

SUMMARY

Plug-ins make developers' lives easier by abstracting away protocol details and letting developers reuse and build easily upon the work of others. The Strophe library provides a simple, yet highly effective plug-in system that allows its users to add advanced functionality. In this chapter you:

- Learned how to load and use Strophe plug-ins
- Discovered how to create your own plug-ins
- Created a roster management plug-in that made dealing with rosters, roster updates, and presence changes really easy
- Saw how easy application logic becomes when plug-ins are available to do the heavily lifting

It's been a long journey, but I hope an interesting and enjoyable one. You built nine applications, including several really sophisticated ones, and learned how to scale XMPP applications. You should now be well versed in the XMPP protocol and the Strophe library as well, and perhaps you've learned a few new JavaScript or jQuery tricks, too. Hopefully your head is now filled with wonderful ideas for great XMPP-powered applications. Good luck and happy hacking!