升。另外，Node.js的一大棘手之处就是它还一直处于开发和改进中，任何事情都很可能发生改变。本书使用2.x系列，因为它比之前的版本健壮许多。

9.2 创建数据库模式

使用MySQL工作时，需要为应用创建数据库模式，它保存在schema.sql文件中。第一部分是创建一个以UTF-8为默认字符集和排序规则的数据库，代码如下所示：

```
DROP DATABASE IF EXISTS PhotoAlbums;

CREATE DATABASE PhotoAlbums
    DEFAULT CHARACTER SET utf8
    DEFAULT COLLATE utf8_general_ci;

USE PhotoAlbums;
```

接着为User表创建表结构，用来存放相册应用的注册用户信息。我们仅仅需要用户的email地址、显示名和密码，并存储一些额外的信息，包括账户的创建时间、上次修改时间以及是否标记为删除状态。User表如下所示：

```
CREATE TABLE Users
(
  user_uuid VARCHAR(50) UNIQUE PRIMARY KEY,
  email_address VARCHAR(150) UNIQUE,

  display_name VARCHAR(100) NOT NULL,
  password VARCHAR(100),

  first_seen_date BIGINT,
  last_modified_date BIGINT,

   deleted BOOL DEFAULT false,

   INDEX(email_address),
   INDEX(user_uuid)
 )
 ENGINE = InnoDB;
```

在MySQL中运行这些命令（mysql-u user-p secret<schema.sql）来建立开始编写代码所需的合适的数据库和表

结构。如果照着GitHub仓库上的代码编写，我们会发现用于相册和照片的表已经添加完成。

9.3 基本数据库操作

MySQL的大部分工作都是创建到MySQL服务器的连接并执行查询和语句。这能让Web应用更简单地运行并挖掘数据库服务器的能力。

9.3.1 连接数据库

为了连接远程服务器，需要通过mysql模块创建一个连接，然后调用connect函数，如下所示：

```
dbclient = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "PhotoAlbums"
});

dbclient.connect(function (err, results) { });
```

如果连接失败，则会接收到一个错误。而如果成功，则可以使用客户端对象来执行查询。当完成数据库相关的工作之后，应该通过调用end方法来关闭连接：

```
dbclient.end();
```

9.3.2 添加查询

连接上数据库服务器之后，我们可以向query方法传入需要执行的SQL语句来执行查询：

```
dbclient.query("SELECT * FROM Albums ORDER BY date", function (err, results, fields)
{});
```

如果查询成功，results参数会传给回调函数，它包含了请求的相应数据，另外，还有第三个参数——fields，如果有额外的信息需要指定则可以使用它（如果没有额外的信息，它通常为空）。对于SELECT语句，回调函数的第二个参数是查询出来的所有行的数组。

```
dbclient.query("SELECT * FROM Albums", function (err, rows) {
    for (var i = 0; i < rows.length; i++) {
        console.log(" -> Album: " + rows[i].name
                    + " (" + rows[i].date + ")");
    }
});
```

如果使用INSERT、UPDATE或者DELETE，则query方法返回的结果与以下所示结果非常相似：

```
{ fieldCount: 0,
  affectedRows: 0,
  insertId: 0,
  serverStatus: 2,
  warningCount: 0,
  message: '',
  changedRows: 0 }
```

我们可以使用这些信息，通过查看affectedRows属性来确保受影响的行数和预期的一样，或通过insertId属性获取最后插入的行，即autogenerated ID。

Node.js中还有许多MySQL的使用方法。现在就让我们开始深入学习吧！

9.4 添加应用身份验证

在上一章中，我们展示了如何将应用的相册和照片存储转换成使用数据库存储，但存在一个问题，即任何人都可以匿名访问添加相册和照片页面。因此，需要对应用做如下改造：

- 为应用提供新用户注册功能。
- 添加相册或者照片时要求用户已经登录和注册。

9.4.1 更新API以支持用户

为了支持新的用户子系统，需要为这些API添加两个新的路由：

```
app.put('/v1/users.json', user_hdlr.register);
app.post('/service/login', user_hdlr.login);
```

第一个路由是个CRUD方法（请参见7.3.1节），通过这个API创建用户。第二个路由用来支持Web浏览器版本应用的登录和验证。我会在后续章节展示这两个方法的实现。

9.4.2 检测核心用户数据操作

要获得数据库的连接，需要为应用程序创建一个data/文件夹，并将一个叫db.js的文件保存到文件夹中（另外还需要一个包含简单错误处理的帮助文件backend_helpers.js）。该文件如代码清单9.1所示，文件提供一个函数，通过使用local.config.js文件提供的连接信息，能够高效地连接MySQL数据库，正如在第8章看到的一样：

代码清单9.1　db.js

```
var mysql = require('mysql'),
    local = require("../local.config.js");

exports.db = function (callback) {

    conn_props = local.config.db_config;
    var c = mysql.createConnection({
        host:     conn_props.host,
        user:     conn_props.user,
        password: conn_props.password,
        database: conn_props.database
    });
    callback(null, c);
};
```

当连接使用结束之后，我们有义务通过调用end方法来关闭连接，以释放它所消耗的资源。有了这个文件，可以开始在data/user.js文件中为新API编写后端代码了。

创建用户

在数据库中注册一个新用户所需的代码如下所示：

```
exports.register = function (email, display_name, password, callback) {
    var dbc;
    var userid;
    async.waterfall([
        // validate ze params
        function (cb) {                                          // 1.
            if (!email || email.indexOf("@") == -1)
                cb(backend.missing_data("email"));
            else if (!display_name)
                cb(backend.missing_data("display_name"));
            else if (!password)
                cb(backend.missing_data("password"));
            else
                cb(null);
        },

        function (cb) {
            db.db(cb);
        },

        function (dbclient, cb) {
            dbc = dbclient;
            bcrypt.hash(password, 10, cb);                       // 3.
        },
        function (hash, cb) {
            userid = uuid();                                     // 4.
            dbc.query(                                           // 5a.
                "INSERT INTO Users VALUES (?, ?, ?, ?, UNIX_TIMESTAMP(), NULL, 0)",
```

```
                [ userid, email, display_name, hash ],
                cb);
        },

        function (results, fields, cb) {                        // 5b.
            exports.user_by_uuid(userid, cb);
        }
    ],
    function (err, user_data) {
        if (dbc) dbc.end();
        if (err) {
            if (err.code
                && (err.code == 'ER_DUP_KEYNAME'
                    || err.code == 'ER_EXISTS'
                    || err.code == 'ER_DUP_ENTRY'))
                callback(backhelp.user_already_registered());
            else
                callback (err);
        } else {
            callback(null, user_data);
        }
    });
};
```

代码做了以下几件事：

1）验证传入的参数，尤其要确保email地址是合法的。如果发送一个验证链接到该email地址以激活账户，则更严格。

2）获取数据库连接。

3）使用bcrypt模块哈希密码。bcrypt是一个生成密码的方法，这使得暴力破解密码变得极其困难。

4）为用户生成一个UUID，稍后会在API中使用它来识别用户。这些ID比简单的整数的用户ID好得多，因为它们难以猜测，也没有明显的顺序。

5）在数据库中执行查询来注册用户，并让数据库返回刚刚创建的用户给调用者。

上面代码中使用了两个新模块：bcrypt（用来密码加密）和node-uuid（生成GUID，使用它来识别用户）。因此需要更新package.json文件的依赖如下所示：

```
  "dependencies": {
    "express": "3.x",
    "async": "0.1.x",
    "mysql": "2.x",
    "bcrypt": "0.x",
    "node-uuid": "1.x"
  }
}
```

注意，这里使用了BIGINT而不是通常的DATATIME字段来存储账户创建日期。因为JavaScript在任何地方都使用时间戳来表示日期和时间，在数据库里同样使用BIGINT类型，在存储和操作这些字段时会更加简单。幸运的是，MySQL新提供了一个函数来协助处理这些日期。

获取用户（通过Email地址或UUID）

现在，已经能够保存用户数据到数据库，我们可以开始编写提取用户的函数。首先，编写一个通用函数，根据数据库中的特殊字段来查询用户：

```
function user_by_field (field, value, callback) {
    var dbc;
    async.waterfall([
        function (cb) {
            db.db(cb);
        },
        function (dbclient, cb) {
            dbc = dbclient;
            dbc.query(
                "SELECT * FROM Users WHERE " + field
                    + " = ? AND deleted = false",
                [ value ],
                cb);
        },
        function (rows, fields, cb) {
            if (!rows || rows.length == 0)
                cb(backhelp.no_such_user());
            else
                cb(null, rows[0]);
        }
    ],
    function (err, user_data) {
        if (dbc) dbc.end();
        callback(err, user_data);
    });
}
```

现在，编写需要输出的函数来获取用户：

```
exports.user_by_uuid = function (uuid, callback) {
    if (!uuid)
        cb(backend.missing_data("uuid"));
    else
        user_by_field("user_uuid", uuid, callback);
};

exports.user_by_email = function (email, callback) {
    if (!email)
        cb(backend.missing_data("email"));
    else
        user_by_field("email_address", email, callback);
};
```

这些就是用户管理中数据部分的全部代码。其他事项都在前端的处理程序上进行处理。

9.4.3 更新express应用

express服务器用来记录用户是否登录的主要方法是session cookie。浏览器每次请求时都会发送cookie，因此可以使用session数据（请参见7.4.5节）来记住用户的登录状态。要设置该功能，需要添加下面的代码到server.js顶部：

```
app.use(express.cookieParser("secret text"));
app.use(express.cookieSession({
    secret: "FLUFFY BUNNIES",
    maxAge: 86400000,
    store: new express.session.MemoryStore()
}));
```

如果用户已经登录，则设置req.session.logged_in=true并将req.session.logged_in_email_address设置为一个合法值。如果用户没有登录，则这些值都设置为undefined。

为了确保用户在使用管理页面之前已经登录，可以引入一个之前没有见过的express功能：能将自己的中间件函数注入到路由声明中。

通常情况下，可以为某个URL指定路由函数，如下所示：

```
app.get(URL, HANDLER_FUNCTION);
```

不仅如此，在URL和处理函数之间可以插入任何数量的中间件函数。这些中间件会按照顺序依次被调用。它们提供三个参数：req、res和next。如果它们想处理请求，则处理请求并发送一个响应（并调用res.end）。如果它们不想处理，则只需简单地调用next()函数。

因此，要求访问管理页面的用户需要处于登录状态，必须修改对应页面的URL路由，如下所示：

```
app.get('/pages/:page_name', requirePageLogin, page_hdlr.generate);
app.get('/pages/:page_name/:sub_page', requirePageLogin, page_hdlr.generate);
```

requirePageLogin函数检查用户请求的页面是否需要认证。如果需要，则检查用户是否已登录。如果已经登录，则允许继续并调用next()；如果还未登录，则拦截URL并重定向到登录页面：

```
function requirePageLogin(req, res, next) {
    if (req.params && req.params.page_name == 'admin') {
        if (req.session && req.session.logged_in) {
            next();
        } else {
            res.redirect("/pages/login");
        }
    } else
        next();
}
```

9.4.4 创建用户处理程序

为支持账户管理，可以在/users.js文件中创建一个新的用户处理程序。与前文中对相册和照片的做法一样，创建一个新的User类以帮助封装用户并实现一个response_obj方法来过滤不需要返回的数据：

```
function User (user_data) {
    this.uuid = user_data["user_uuid"];
    this.email_address = user_data["email_address"];
    this.display_name = user_data["display_name"];
    this.password = user_data["password"];
    this.first_seen_date = user_data["first_seen_date"];
    this.last_modified_date = user_data["last_modified_date"];
    this.deleted = user_data["deleted"];
}

User.prototype.uuid = null;
User.prototype.email_address = null;
User.prototype.display_name = null;
User.prototype.password = null;
User.prototype.first_seen_date = null;
User.prototype.last_modified_date = null;
User.prototype.deleted = false;
User.prototype.check_password = function (pw, callback) {
    bcrypt.compare(pw, this.password, callback);
};
User.prototype.response_obj = function () {
    return {
        uuid: this.uuid,
        email_address: this.email_address,
        display_name: this.display_name,
        first_seen_date: this.first_seen_date,
        last_modified_date: this.last_modified_date
    };
};
```

创建新用户

接下来，实现创建用户的函数。基本流程如下：

1）检查传入的数据，确保数据合法。

2）在后端创建一个用户账户并返回原始数据。

3）标识用户已经登录。

4）返回最新创建的用户对象给调用者。

该函数的代码如下所示：

```
exports.register = function (req, res) {
    async.waterfall([
        function (cb) {                                          // 1.
            var em = req.body.email_address;

            if (!em || em.indexOf("@") == -1)
                cb(helpers.invalid_email_address());
            else if (!req.body.display_name)
                cb(helpers.missing_data("display_name"));
            else if (!req.body.password)
                cb(helpers.missing_data("password"));
            else
                cb(null);
        },
        function (cb) {                                          // 2.
            user_data.register(
                req.body.email_address,
                req.body.display_name,
                req.body.password,
                cb);
        },
        function (user_data, cb) {                               // 3.
            req.session.logged_in = true;
            req.session.logged_in_email_address = req.body.email_address;
            req.session.logged_in_date = new Date();
            cb(null, user_data);
        }
    ],
    function (err, user_data) {                                  // 4.
        if (err) {
            helpers.send_failure(res, err);
        } else {
            var u = new User(user_data);
            helpers.send_success(res, {user: u.response_obj() });
        }
    });
};
```

用户登录

用户登录系统，需要进行以下操作：

1）根据提供的email地址匹配用户对象（如果email地址不存在则抛出一个错误）。

2）使用bcrypt模块比较密码。

3）如果密码匹配则设置seesion变量来标识用户已经登录。否则，标识验证失败。

整个流程如下：

```
exports.login = function (req, res) {
    async.waterfall([
        function (cb) {
            var em = req.body.email_address;
            if (!em || em.indexOf('@') == -1)
                cb(helpers.invalid_email_address());
            else if (req.session && req.session.logged_in_email_address
                    == em.toLowerCase())
                cb(helpers.error("already logged in", ""));

           else if (!req.body.password)
               cb(helpers.missing_data("password"));
           else
               user_data.user_by_email(req.body.email_address, cb);        // 1.
       },
       function (user_data, cb) {
           var u = new User(user_data);
           u.check_password(req.body.password, cb);                        // 2.
       },
       function (auth_ok, cb) {
           if (!auth_ok) {
               cb(helpers.auth_failed());
               return;
           }
           req.session.logged_in_email_address = req.body.email_address;   // 3.
           req.session.logged_in_date = new Date();
           cb(null);
       }
   ],
   function (err, results) {
       if (!err || err.message == "already_logged_in") {
           helpers.send_success(res, { logged_in: true });
       } else {
           helpers.send_failure(res, err);
       }
   });
};
```

测试登录状态

要想测试用户是否已经登录，只需查看对应的seesion变量是否已经设置：

```
exports.logged_in = function (req, res) {
    var li = (req.session && req.session.logged_in_email_address);
    helpers.send_success(res, { logged_in: li });
};
```

注销

最后，注销用户只需清除seesion数据，这样系统就不再认为用户处于登录状态：

```
exports.logout = function (req, res) {
    req.session = null;
    helpers.send_success(res, { logged_out: true });
};
```

9.4.5 创建登录和注册页面

对于新的用户子系统，应用中新增两个页面：登录页面和注册页面。两个页面都和之前一样由两个文件组成：一个JavaScript加载器和一个HTML文件。两者的JavaScript加载器也都非常标准：

```
$(function(){
    var tmpl,   // Main template HTML
    tdata = {};  // JSON data object that feeds the template

    // Initialize page
    var initPage = function() {

        // Load the HTML template
        $.get("/templates/login OR register.html", function(d){
            tmpl = d;
        });

        // When AJAX calls are complete parse the template
        // replacing mustache tags with vars
        $(document).ajaxStop(function () {
            var renderedPage = Mustache.to_html( tmpl, tdata );
            $("body").html( renderedPage );
        });
    }();
});
```

注册页面的HTML如代码清单9.2所示。除了显示注册表单的HTML，还有一些JavaScript以确保用户已经填入所有的字段，验证两个密码是否一致，然后提交数据到后端服务器。如果登录成功，则重定向到应用首页；否则，会显示错误并让用户再试一次。

代码清单9.2　注册页面的Mustache模板（register.html）

```
<div style="float: right"><a href="/pages/login">Login</a></div>
<form name="register" id="register">
  <div id="error" class="error"></div>
  <dl>
    <dt>Email address:</dt>
    <dd><input type="text" size="30" id="email_address"
               name="email_address"/></dd>
    <dt>Display Name:</dt>
    <dd><input type="text" size="30" id="display_name" name="display_name"/></dd>
    <dt>Password:</dt>
    <dd><input type="password" size="30" id="password" name="password"/></dd>
    <dt>Password (confirm):</dt>
    <dd><input type="password" size="30" id="password2" name="password2"/></dd>
    <dd><input type="submit" value="Register"/>
  </dl>
</form>

<script type="text/javascript">
$(document).ready(function () {
    if (window.location.href.match(/(fail)/) != null) {
        $("#error").html("Failure creating account.");
    }
});

$("form#register").submit(function (e) {
  if (!$("input#email_address").val()
      || !$("input#display_name").val()
      || !$("input#password").val()
      || !$("input#password2").val()) {
      $("#error").html("You need to enter an email and password.");
  } else if ($("input#password2").val() != $("input#password").val()) {
      $("#error").html("Passwords don't match.");
  } else {
      var info = { email_address: $("input#email_address").val(),
                   display_name: $("input#display_name").val(),
                   password: $("input#password").val() };
      $.ajax({
          type: "PUT",
          url: "/v1/users.json",
          data: JSON.stringify(info),
          contentType: "application/json; charset=utf-8",
          dataType: "json",
          success: function (data) {
              window.location = "/pages/admin/home";
          },
          error: function () {
              var ext = window.location.href.match(/(fail)/) ? "" : "?fail";
              window.location = window.location + ext;
              return false;
          }
      });
  }
  return false;
});
</script>
```

最后，登录页面的代码如代码清单9.3所示。它和注册页面非常相似：显示表单的HTML并包含一些JavaScript代码用来处理数据并提交到服务器。

代码清单9.3　登录页面的Mustache模板（login.html）

```
<div style='float: right'><a href='/pages/register'>Register</a></div>
<form name="login" id="login">
  <div id="error" class="error"></div>
  <dl>
    <dt>Email address:</dt>
    <dd><input type="text" size="30" id="email_address"
               name="email_address"/></dd>
    <dt>Password:</dt>
    <dd><input type="password" size="30" id="password" name="password"/></dd>
    <dd><input type="submit" value="Login"/>
  </dl>
</form>

<script type="text/javascript">
```

```
$(document).ready(function () {
    if (window.location.href.match(/(fail)/) != null) {
        $("#error").html("Invalid login credentials.");
    }
});
$("form#login").submit(function (e) {
  if (!$("input#email_address").val() || !$("input#password").val()) {
      $("#error").html("You need to enter an email and password.");
  } else {
      var info = { email_address: $("input#email_address").val(),
                   password: $("input#password").val() };
      $.ajax({
          type: "POST",
          url: "/service/login",
          data: JSON.stringify(info),
          contentType: "application/json; charset=utf-8",
          dataType: "json",
          success: function (data) {
              window.location = "/pages/admin/home";
          },
          error: function () {
              var ext = window.location.href.match(/(fail)/) ? "" : "?fail";
              window.location = window.location + ext;
              return false;
          }
      });
  }
  return false;
});
</script>
```

当添加完这些文件（data/user.js和handlers/user.js、login.js和login.html以及register.js和register.html）之后，我们就拥有了一个完整的Web浏览器端的登录系统。

9.5 资源池

在上一章中，我们了解到Node.js中的MongoDB驱动会管理自己的连接集合，或连接“池”。但mysql模块没有类似的功能，因此需要我们自己处理连接池机制。幸运的是，这一点非常容易做到，不必担心模块没有实现这个功能。

在npm中，有个叫做generic-pool的模块允许池化（pooling）任何东西。要使用它，需要创建一个池，指定要创建的项的上限，并提供一个函数，用于在池中创建对象的新实例。然后，要想从池中获取一个项，我们可以调用acquire函数，这个函数会一直等待，直到有可用的池对象出现。当任务完成之后，调用release函数将对象返回给池。

接下来，我们一起看看具体是如何实现的。

9.5.1 入门

添加generic-pool到package.json文件的依赖中：

```
"dependencies": {
  "express": "3.x",
  "async": "0.1.x",
  "generic-pool": "2.x",
  "mysql": "2.x",
  "bcrypt": "0.x",
  "node-uuid": "1.x"
}
```

要在文件中引入这个模块，需要require它，如下所示：

```
var pool = require('generic-pool'),
```

如果想创建一个池，需要创建含有配置信息的池类的新实例，即指定需要的池中项的数量，以及创建新对象的函数：

```
conn_props = local.config.db_config;
mysql_pool = pool.Pool({
    name     : 'mysql',
    create   : function (callback) {
        var c = mysql.createConnection({
            host:     conn_props.host,
            user:     conn_props.user,
            password: conn_props.password,
            database: conn_props.database
        });
        callback(null, c);
    },
    destroy           : function (client) { client.end();
    max               : conn_props.pooled_connections,
    idleTimeoutMillis : conn_props.idle_timeout_millis,
    log               : false
});
```

9.5.2 处理连接

现在，在代码中，无论何时想要一个MySQL服务器的连接，都无需创建一个新连接，而是让池管理这一切。acquire函数会阻塞住，直到有可用的连接出现。db.js中的db函数变成如下所示：

```
exports.db = function (callback) {
    mysql_pool.acquire(callback);
};
```

当连接使用完毕，只需要将连接释放回连接池，代码如下所示：

```
conn.release();
```

而其他等待连接的人就可以立刻获得该连接。

9.6 验证API

现在，你可能会闲下心来并自鸣得意，因为应用会在检测到登录之后，才允许创建相册或者添加照片到相册中，但仍然有一个严重的安全问题：API本身并不足够安全。仍然可以运行以下代码而不需要任何验证：

```
curl -X PUT -H "Content-Type: application/json" \
     -d '{ "name": "hk2012", "title" : "moo", "description": "cow", "date":
"2012-12-28" }'\
     http://localhost:8080/v1/albums.json
```

如果系统只有某些部分是安全的，而其他部分则完全没有受到保护，那么这样的系统不会令人印象深刻，我们需要立即解决这个问题。

可以通过以下方式创建系统：

- Web浏览器的前端部分继续使用登录和注册页面，使用登录服务来管理对服务器的访问。

- API用户在发送请求时会使用HTTP基本身份验证传入用户名和密码，从而验证受限制API的访问权。

HTTP基本身份验证：安全吗？

正如本章中解释的一样，HTTP基本身份验证是一个相对简单的安全形式：每次给服务器发送请求，客户端都会将用户名和经过base64编码的密码传入请求头，每次处理前，都经过服务器的认证。因为base64编码并不是一种加密形式，所以它相当于在互联网上通过纯文本传递用户名和密码。因此，我们会有这样的疑问：本质上，它是不安全的吗？

答案是，很大程度上它是安全的。大多数需要安全性的网站都会使用SSL/HTTPS来加密和传输数据。在客户端和服务器之间进行加密数据传输的方法是相当安全的，而在服务器上绝对也会做一些处理。

部分人会认为Web浏览器会记住基本身份验证的用户名和密码，而这是不安全的。这种看法是合理的，这也是为什么只针对API服务器使用基本认证。对于基于浏览器的应用，应该坚持使用常规的登录页面和session cookie。

因此，虽然HTTP基本身份验证可能并不安全，但加上一点点的准备和预防措施之后，我们的REST API服务器就能成为可信赖的产品级别的安全系统。

客户端不能记住来自API用户的seesion，因为它们没有传入cookie，因此，没有登录状态会被存储下来。这些用户每次独立的请求都需要验证。要达到这个目的，它们传入以下格式的头部信息：

```
Authorization: Basic username:password base64 encoded
```

例如，要发送含有密码kittycatonkeyboard的marcwan@example.org的认证，需要发送

```
Authorization: Basic bWFyY3dhbkBleGFtcGxlLm9yZzpraXR0eWNhdG9ua2V5Ym9hcmQ=
```

幸运的是，我们不必自己完成这些头信息，因为有程序（例如curl）会为我们完成这样的工作。要通过curl请求传递HTTP基本身份验证信息，可以添加下面的代码到请求中：

```
curl -u username:password ...
```

因此，我们可以对系统稍作修改，给这些API添加些安全措施：修改这两个创建相册和添加照片的PUT API。要做到这一点，需要编写一段叫做requireAPILogin的中间件：

```
app.put('/v1/albums.json', requireAPILogin, album_hdlr.create_album);
app.put('/v1/albums/:album_name/photos.json',
        requireAPILogin, album_hdlr.add_photo_to_album);
```

requireAPILogin函数的代码如下所示：

```
function requireAPILogin(req, res, next) {
    if (req.session && req.session.logged_in) {                    // 1.
        next();
        return;
    }
    var rha = req.headers.authorization;                           // 2.
    if (rha && rha.search('Basic ') === 0) {
        var creds = new Buffer(rha.split(' ')[1], 'base64').toString();
        var parts = creds.split(":");                              // 3.
        user_hdlr.authenticate_API(                                // 4.
            parts[0],
            parts[1],
            function (err, resp) {
                if (!err && resp) {
                    next();
                } else
                    need_auth(req, res);
            }
        );
    } else
        need_auth(req, res);
}
```

代码的工作原理如下：

1）为了保持与Web应用的兼容性，首先会检查用户是否已经在浏览器中使用session登录。如果是，表明用户很清楚需要使用什么API。因为命令行或简单客户端的纯API调用不会传入cookie（但浏览器里的Ajax调用API则会），因此没有session，必须提供额外的认证信息。

2）查看请求头Authentication:。

3）解码并分离头部信息中的用户名和密码。

4）在用户处理程序中把这些信息传递给authenticate_API函数。

用户处理程序中的authenticate_API函数如下所示——通过给定的email地址获取用户，使用bcrypt npm模块验证密码，最后返回成功或失败：

```
exports.authenticate_API = function (un, pw, callback) {
    async.waterfall([
        function (cb) {
            user_data.user_by_email(un, cb);
        },

        function (user_data, cb) {
            var u = new User(user_data);
            u.check_password(pw, cb);
        }
    ],
    function (err, results) {
        if (!err) {
            callback(null, un);
        } else {
            callback(new Error("bogus credentials"));
        }
    });
};
```

最后，如果确定用户不符合对应请求的验证，则需要做一些工作，告诉用户不允许使用请求的资源：

```
function need_auth(req, res) {
    res.header('WWW-Authenticate',
               'Basic realm="Authorization required"');
    if (req.headers.authorization) {
        // no more than 1 failure / 5s
        setTimeout(function () {
            res.send('Authentication required\n', 401);
        }, 3000);
    } else {
        res.send('Authentication required\n', 401);
    }
}
```

该函数完成了API服务器在HTTP基本身份验证方面的工作。如果验证失败，它返回响应头WWW-Authenticate。为了额外的安全性，如果看到用户发送错误的用户名/密码组合，则可以暂停几秒钟以阻止进一步的外部攻击。这之后，相册应用程序算是有了一个相对安全的Web和API服务器。

而要真正的安全，则应该确保任何请求都要求一个密码，这个密码是通过HTTPS安全连接发送的，因此没有人能够使用数据包嗅探器在线路上查看密码。我们会在第10章详细讨论HTTPS。

9.7 小结

本章介绍了两个新知识点：在Node应用中使用MySQL数据库和添加用户验证到Web应用以及API服务器。同时，我也在GitHub代码树上更新了含有用户验证子系统的MongoDB版本应用，这样我们就知道它具体是如何实现的。通过创建两个平行的验证系统，我们为浏览器应用创造了良好的用户体验，而让API用户仍然得到最为简单而强大的JSON API。

在结束本书第三部分时，我们学习到如何添加功能强大的新技术到相册应用，包括express和数据库，并创建了功能完整（可能只是基本的）且可以构建的项目。在本书的最后一部分，会覆盖本书之前一直忽略的一些细节。首先，我们会在第10章中学习如何部署应用。

第四部分　进阶篇

第10章　部署和开发

在充分掌握构建Node.js应用的能力之后，我们现在可以将注意力转到部署和开发应用等主题上来。在本章中，我们首先了解人们在产品服务器上部署和运行Node应用的各种方式，包括UNIX/Mac和Windows平台。接着会学习如何利用拥有多核处理器的机器，虽然事实上Node.js是一个单线程平台。

接下来我们会关注如何在服务器上添加对虚拟主机的支持，同时添加SSH/HTTPS来确保应用安全。最后，会快速浏览在Windows和UNIX/Mac机器上跨平台开发Node应用时带来的问题。