

Bigtable、Blobstore、あるいは Google Storage を使用した GAE ストレージ

GAE のデータ・ストレージの 3 つの選択肢に対処する方法を学んでください

John Wheeler

Applications manager
Xerox

2012年 1月 06日

Google App Engine ではリレーショナル・データベースの使用を控え、非リレーショナルデータストアを採用しています。具体的には、Bigtable、Blobstore、そして最も新しい非リレーショナル・データストアである Google Storage for Developers です。この記事では著者の John Wheeler が、それぞれのデータストアのセットアップ方法と使用方法に慣れ親しむためのアプリケーション・シナリオを説明するなかで、この GAE のビッグ・データ・ストレージの 3 つの選択肢の利点と欠点を詳しく探ります。

ディスク・ドライブとそこにマウントされたファイルシステムは、アプリケーションやデータを格納するために常に存在してきたため、当然あるものと思ってしまうがちです。こうした環境でファイルを書き込む場合、考慮しなければならないのは、ファイルの場所、アクセス権、必要なスペースぐらいなもので、後は `java.io.File` を構成してファイルの操作に取り掛かるだけです。`java.io.File` は、ユーザーがデスクトップ・コンピュータで作業していようが、Web サーバー、あるいはモバイル機器で作業していようが、同じように機能します。しかし、GAE (Google App Engine) を使い始めてみると、ファイルシステムが見えないところでいかに重要な役割を果たしていたか (あるいはファイルシステムが無いことの影響) がたちまち明らかになってきます。GAE ではファイルをディスクに書き込むことはできません。それは、使用可能なファイルシステムがないからです。実のところ、`java.io.FileInputStream` はブラックリストによって GAE SDK から除外されているため、このクラスを宣言するだけでも、コンパイル・エラーがスローされます。

幸い、人生には選択肢が付き物です。しかも、GAE ではとりわけ強力なストレージの選択肢を提供しています。初めからスケーラビリティを念頭に設計されている GAE には、2 つのキー・バリュー型ストアがあります。一方は、通常データベースに入れるような標準的なデータを格納するデータストア (Bigtable という名前が付けられています)、もう一方は大規模なバイナリー BLOB データを格納する Blobstore です。両方ともアクセス時間の長さに一定の制限を設けているという

点、そして、これまで皆さんが扱ったことのあるファイルシステムとはまったく異なるという点では共通しています。

この2つのデータストアに加え、GAEの選択肢に新しく加わったのが Google Storage for Developers です。Google Storage for Developers は Amazon S3 のように機能しますが、S3 も従来のファイルシステムとは顕著に異なります。この記事では、GAE ストレージの3つの選択肢のそれぞれを実装するサンプル・アプリケーションを順に作成していきます。Bigtable、Blobstore、Google Storage for Developers を使用した実際の作業から、各実装が持つ利点と欠点を理解してください。

前提条件

この記事の例に従うには、GAE アカウントと、いくつかの無料のオープンソース・ツールが必要です。開発環境には JDK 5 または JDK 6、そして [Eclipse IDE for Java Developers](#) の他、以下のものも必要となります。

- Google Plug-in for Eclipse
- [Apache Commons FileUpload](#)
- [Objectify-Appengine](#)

Google Storage for Developers はこの記事が執筆された 2010年の時点では、米国内の限定された数の開発者使のみが使用することができます。Google Storage にすぐにアクセスできないとしても、Bigtable と Blobstore の例には従うことができるので、Google Storage が機能する仕組みについても十分感触がつかめるはずです。

事前に必要なセットアップ: サンプル・アプリケーション

GAE ストレージ・システムについて詳しく探る前に、サンプル・アプリケーションに必要な以下の3つのクラスを作成してください。

- 写真を表す Bean クラス。Photo Bean クラスには、タイトルとキャプションなどのフィールドの他、バイナリー画像データを保存するための2、3のフィールドを含めます。
- Photo オブジェクトをデータストアに永続化する DAO クラス。DAO クラスには Photo オブジェクトを挿入するメソッドと、ID を基準に Photo オブジェクトを抽出するメソッドを含めます。この DAO クラスは、Objectify-Appengine というオープンソース・ライブラリーを使用して永続化に対応します。
- Template Method パターンを使用して3つのステップからなるワークフローをカプセル化するサーブレット・クラス。GAE ストレージの各選択肢について探る際には共通してこのワークフローを使用します。

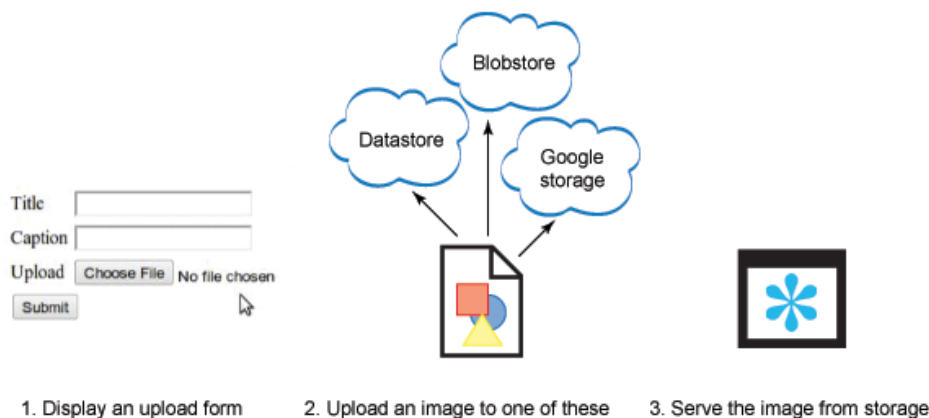
アプリケーションのワークフロー

この記事では GAE データ・ストレージの3つの選択肢について学ぶために、共通の手順に従います。それによって、それぞれの技術に焦点を絞れるとともに、各ストレージ手法の利点と欠点を比較することができるからです。サンプル・アプリケーションは、一貫して以下のワークフローに従います。

1. アップロード・フォームを表示する。
2. 画像をストレージにアップロードし、レコードをデータストアに保存する。
3. 画像を取得する。

図 1 に、アプリケーションのワークフローを示します。

図 1. ストレージの各選択肢のデモに使用する 3 ステップのワークフロー



このワークフローに従うことによってもたらされるメリットとして、このサンプル・アプリケーションは、ストレージに対してバイナリーを書き込み、それを読み出すという、あらゆる GAE プロジェクトにとって重要なタスクの練習にもなります。それでは早速、3 つのクラスの作成に取り掛かりましょう。

GAE を対象とした単純なアプリケーション

[Eclipse](#) をまだ入手していない場合は、ダウンロードしてください。その上で、Google Plug-in for Eclipse をインストールして、GWT を使用しない新規 Google Web アプリケーション・プロジェクトを作成します。プロジェクト・ファイルの構造については、この記事に付属の[サンプル・コード](#)を参照してください。Google Web アプリケーションをセットアップしたら、アプリケーションの最初のクラスとして、リスト 1 に記載する `Photo` クラスを追加します (ゲッターおよびセッターは省略されています)。

リスト 1. Photo クラス

```
import javax.persistence.Id;

public class Photo {

    @Id
    private Long id;
    private String title;
    private String caption;
    private String contentType;
    private byte[] photoData;
    private String photoPath;

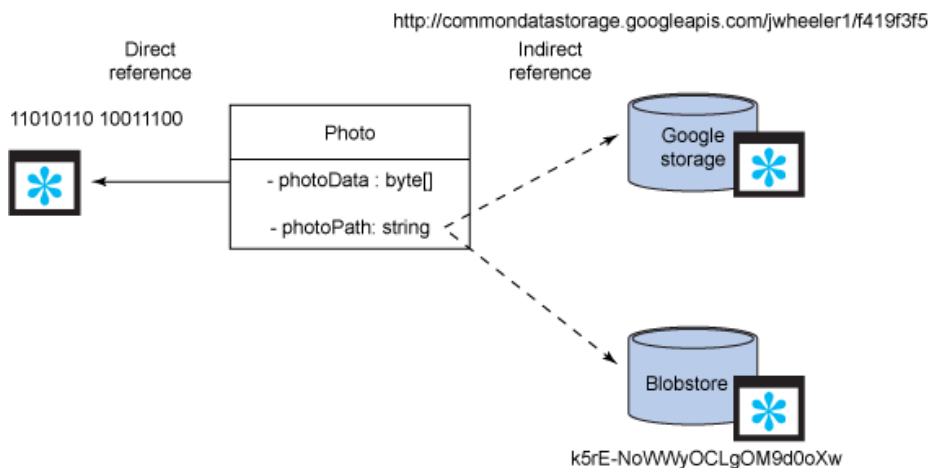
    public Photo() {
    }

    public Photo(String title, String caption) {
        this.title = title;
        this.caption = caption;
    }

    // getters and setters omitted
}
```

@Id アノテーションは、主キーとするフィールドを指定します。このことは、Objectify を扱う際に重要になってきます。主キーは、データストアに保存される 1 つひとつのレコード (エンティティとも呼ばれます) に必要です。画像をアップロードする 1 つの方法としては、バイト配列である `photoData` に直接画像を格納する方法があります。`photoData` は、Blob プロパティとして `Photo` クラスの残りのフィールドと併せてデータストアに書き込まれます。つまり、画像は Bean と一緒に保存および取得されるということです。画像を Blobstore または Google Storage にアップロードする場合は、データはシステム外部に保存され、`photoPath` がその場所を示します。いずれにしても、`photoData` または `photoPath` のどちらかが使用されます。図 2 では、それぞれの機能を明らかにしています。

図 2. `photoData` および `photoPath` の機能



次に、Bean のパーシスタンスに対処します。

オブジェクト・ベースの永続化

前述のとおり、`Photo` Bean 用の DAO クラスを作成するには、Objectify を使用します。永続 API としては JDO と JPA の方がより一般的で広く使われているかもしれませんが、JDO と JPA を習得するには時間がかかります。また別の選択肢として、下位レベルの GAE データストア API を使用することもできますが、その場合 Bean とデータストア・エンティティの間でマーシャリングを行わなければならない、うんざりするような作業をする羽目になります。Objectify は私たちに代わって、そうした面倒な部分を Java リフレクションという手段で解決します (Objectify-Appengine を始めとし、GAE における永続化の手段について詳しく学ぶには、「[参考文献](#)」を参照してください)。

まず、`PhotoDao` という名前のクラスを作成するところから始めます。、このクラスをリスト 2 のようにコーディングしてください。

リスト 2. PhotoDao クラス

```
import com.googlecode.objectify.*;
import com.googlecode.objectify.helper.DAOWBase;

public class PhotoDao extends DAOWBase {

    static {
        ObjectifyService.register(Photo.class);
    }

    public Photo save(Photo photo) {
        ofy().put(photo);
        return photo;
    }

    public Photo findById(Long id) {
        Key<Photo> key = new Key<Photo>(Photo.class, id);
        return ofy().get(key);
    }
}
```

PhotoDao が継承する DAOWBase は、Objectify インスタンスの遅延ロードを行うコンビニエンス・クラスです。このサンプル・アプリケーションでは、Objectify が API への主要なインターフェースであり、そのインターフェースとしては ofy メソッドが使われます。ただし、ofy を使用するには、[リスト 2](#) の Photo オブジェクトのような静的イニシャライザーに永続クラスをあらかじめ登録しておく必要があります。

DAO には Photo オブジェクトを挿入するメソッドと検索するメソッドの 2 つがあります。それぞれのメソッドで Objectify を扱うのは、ハッシュ・テーブルを扱うのと同じくらい簡単です。findById の中で Key を使って Photo オブジェクトを取得していることにお気付きかもしれませんが、これについて深く考える必要はありません。この場合の Key は、単なる id フィールドのラッパーであると考えてください。

これで、Photo Bean クラス、そして永続ストアを管理するための PhotoDao クラスは作成できたので、次は、アプリケーションのワークフローを具体化します。

Template Method パターンによるアプリケーションのワークフロー

Mad Libs (訳注: 米国で子供向けに古くから出版されている言葉遊びのゲームの本) で遊んだことがあるとしたら、Template Method パターンを理解できるはずです。Mad Libs の 1 つひとつの物語には空白の部分がたくさんあり、読者がこれらの空白を埋めていきます。読者の入力によって(空白の部分をどのような単語で埋めるかによって) 物語は大幅に変わってくるという仕組みです。同じように、Template Method パターンを使用するクラスには一連のステップがあり、その一部が空白になっています。

これから Template Method パターンを使用して、サンプル・アプリケーションのワークフローを実行するサブレットを作成します。まず、抽象サブレットのスタブを作成し、AbstractUploadServlet という名前を付けてください。参考として、リスト 3 のコードを使用することができます。

リスト 3. AbstractUploadServlet

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public abstract class AbstractUploadServlet extends HttpServlet {

}
```

次に、リスト 4 に示す 3 つの抽象メソッドを追加します。それぞれのメソッドが、ワークフローの各ステップを表します。

リスト 4. 3 つの抽象メソッド

```
protected abstract void showForm(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException;

protected abstract void handleSubmit(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException;

protected abstract void showRecord(long id, HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException;
```

ここでは Template Method パターンを使用していることから、[リスト 4](#) のメソッドは空白の部分、リスト 5 のコードはこれらの部分を組み込んで完成させる物語だと考えてください。

リスト 5. ワークフローの登場

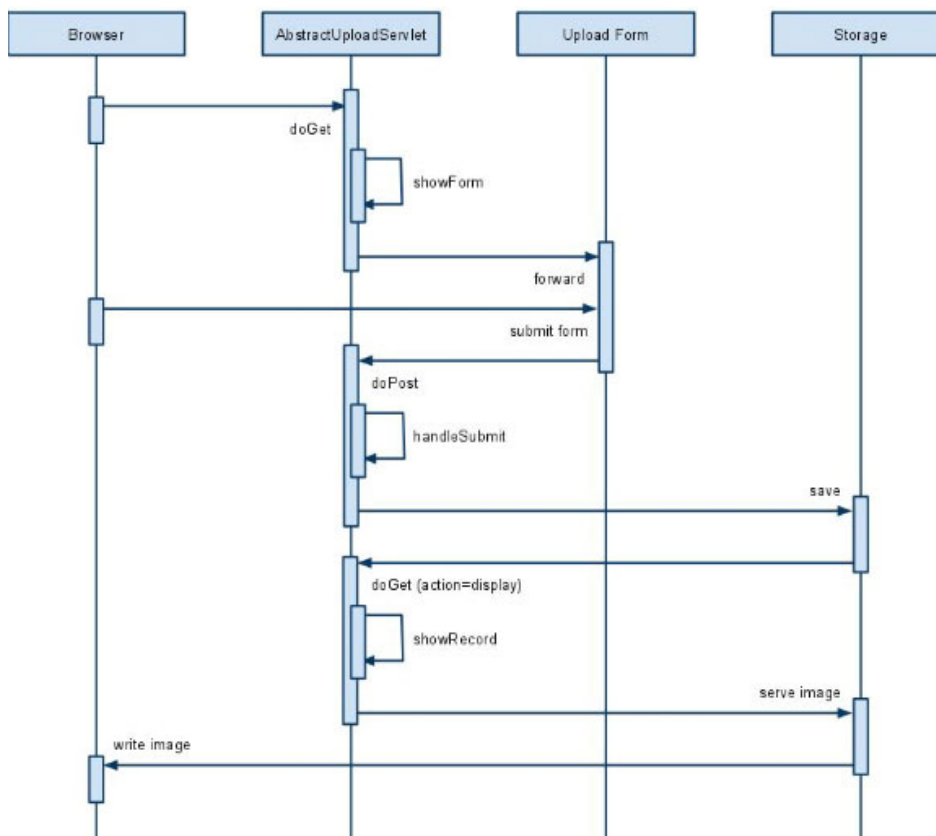
```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String action = req.getParameter("action");
    if ("display".equals(action)) {
        // don't know why GAE appends underscores to the query string
        long id = Long.parseLong(req.getParameter("id").replace("_", ""));
        showRecord(id, req, resp);
    } else {
        showForm(req, resp);
    }
}

@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    handleSubmit(req, resp);
}
```

サーブレットについての復習

昔ながらのサーブレットをしばらく扱っていないという方のために言っておくと、doGet と doPost は、それぞれ HTTP の GET と POST を処理するための標準的なメソッドです。通常は、GET を使用して Web リソースを取得し、POST を使用してデータを送信します。この考えに従うと、サンプル・アプリケーションでの doGet の実装はアップロード・フォームまたはストレージからの写真を表示し、doPost はアップロード・フォームの送信を処理することになります。それぞれの振る舞いを具体的に定義するのは、AbstractUploadServlet を継承するクラスです。図 3 に、発生するイベントのシーケンスを示します。ワークフローの内容を正確に理解するには、多少時間がかかるかもしれません。

図 3. ワークフローのシーケンス図



ここまでで必要な3つのクラスが作成されたので、サンプル・アプリケーションはいつでも実行できる状態にあります。ここからは、それぞれの GAE ストレージの選択肢がこのアプリケーションのワークフローとどのような形で関わってくるかの説明に焦点を絞ることができます。最初に取り上げるのは、Bigtableです。

GAE ストレージの選択肢 1: Bigtable

Google による GAE のドキュメントでは、Bigtable のことをシャードイング手法で分割されてソートされた配列として説明していますが、私はそれよりも、膨大な数のサーバーに分割された巨大なハッシュ・テーブルとして考えたほうが理解しやすいと思います。Bigtable には、リレーショナル・データベースと同じようにデータ型があります。実際、Bigtable とリレーショナル・データベースはどちらも BLOB 型を使用してバイナリーを保存します。

BLOB 型を Blobstore と混同しないでください。Blobstore は、GAE で提供している、もう1つのキー・バリュー型データストアです。Blobstore については、次のセクションで詳しく説明します。

Bigtable で BLOB データを扱う場合、BLOB データは他のフィールドと一緒にロードされるため、そのまますぐに使用できて非常に便利です。一方、重要な注意事項も1つあります。それは、将来的に緩和される可能性があるとは言え、BLOB データのサイズが最大 1MB に制限されることです。最近のデジタル・カメラでは、1枚の写真のデータ・サイズが 1MB を下回るようなものはほとんどないため、画像が関係するような場合に(このサンプル・アプリケーションもこれに該当します) Bigtable を使用すると、このようなデータ・サイズの問題が出てくる可能性があります。と

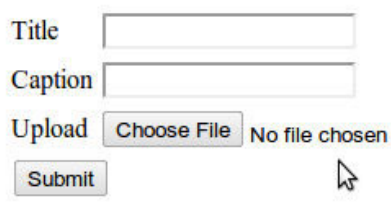
りあえずは 1MB の制限があっても問題がない場合、あるいは画像よりもデータ・サイズが小さいものを保存するのであれば、3 つの GAE ストレージの選択肢のうち、最も扱いやすい Bigtable が賢明な選択になります。

Bigtable にデータをアップロードできるように、まずアップロード・フォームを作成します。続いて、Bigtable 用にカスタマイズされた 3 つの抽象メソッドで構成されるサブレットを実装します。そして最後に、ユーザーが簡単に破ってしまいそうな 1MB の制限を超えた場合に備えてエラー処理を実装します。

アップロード・フォームの作成

図 4 に、Bigtable のアップロード・フォームを示します。

図 4. Bigtable のアップロード・フォーム



このフォームを作成するには、datastore.jsp という名前のファイルを作成して、そこにリスト 6 のコード・ブロックを含めてください。

リスト 6. カスタム・アップロード・フォーム

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <form method="POST" enctype="multipart/form-data">
      <table>
        <tr>
          <td>Title</td>
          <td><input type="text" name="title" /></td>
        </tr>
        <tr>
          <td>Caption</td>
          <td><input type="text" name="caption" /></td>
        </tr>
        <tr>
          <td>Upload</td>
          <td><input type="file" name="file" /></td>
        </tr>
        <tr>
          <td colspan="2"><input type="submit" /></td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

フォームの method 属性は POST に設定し、エンクロージャー・タイプは multipart/form-data に設定する必要があります。action 属性は指定されていないため、フォームの送信先はこのフォー

ム自体です。POST を実行すると、AbstractUploadServlet の doPost メソッドが実行され、それによって handleSubmit メソッドが呼び出されます。

フォームは完成したので、次は、このフォームをつかさどるサーブレットの実装に取り掛かります。

Bigtable への画像のアップロードとアップロードした画像の取得

これからサーブレットに、3つのメソッドを順番に実装していきます。まず、前のセクションで作成したフォームを表示するメソッド、次に画像のアップロード処理を行うメソッドを実装します。最後に実装するメソッドは、アップロードした画像がどのように取得されるかを確認するために、アップロードした画像の取得を実行するメソッドです。

このサーブレットでは、[Apache Commons の FileUpload ライブラリー](#)を使用します。このライブラリーとその依存関係をダウンロードしてプロジェクトに含めてください。それが終わったら、リスト7のスタブを作成します。

リスト 7. DatastoreUploadServlet

```
import info.johnwheeler.gaestorage.core.*;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import org.apache.commons.fileupload.*;
import org.apache.commons.fileupload.servlet.ServletFileUpload;
import org.apache.commons.fileupload.util.Streams;

@SuppressWarnings("serial")
public class DatastoreUploadServlet extends AbstractUploadServlet {
    private PhotoDao dao = new PhotoDao();
}
```

ここではまだ、興味深いことは何も行われていません。必要なクラスをインポートして、後で使用する PhotoDao オブジェクトを構成しているだけです。DatastoreUploadServlet をコンパイルするのは、抽象メソッドを実装してからです。では早速、各メソッドを実装していきましょう。最初の実装するのは、リスト8の showForm メソッドです。

リスト 8. showForm メソッド

```
@Override
protected void showForm(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    req.getRequestDispatcher("datastore.jsp").forward(req, resp);
}
```

見てのとおり、showForm メソッドはアップロード・フォームに処理を委譲するだけに過ぎません。handleSubmit メソッドの実装は、これよりも複雑になってきます(リスト9を参照)。

リスト 9. handleSubmit メソッド

```
@Override
protected void handleSubmit(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException {
    ServletFileUpload upload = new ServletFileUpload();
}
```

```
try {
    FileItemIterator it = upload.getItemIterator(req);

    Photo photo = new Photo();

    while (it.hasNext()) {
        FileItemStream item = it.next();
        String fieldName = item.getFieldName();
        InputStream fieldValue = item.openStream();

        if ("title".equals(fieldName)) {
            photo.setTitle(Streams.asString(fieldValue));
            continue;
        }

        if ("caption".equals(fieldName)) {
            photo.setCaption(Streams.asString(fieldValue));
            continue;
        }

        if ("file".equals(fieldName)) {
            photo.setContentType(item.getContentType());
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            Streams.copy(fieldValue, out, true);
            photo.setPhotoData(out.toByteArray());
            continue;
        }
    }

    dao.save(photo);
    resp.sendRedirect("datastore?action=display&id=" + photo.getId());
} catch (FileUploadException e) {
    throw new ServletException(e);
}
```

上記のコードは長くなっていますが、行っている内容は単純なものです。handleSubmit メソッドは、アップロード・フォームのリクエスト本体のストリームを取得し、フォームを構成するフィールドの1つをリクエストから抽出して FileItemStream に格納し、FileItemStream に格納された値を使用して Photo オブジェクトの構成要素の1つをセットアップするという処理を、フォームの各フィールドに対して順番に行っていきます。フィールドの1つひとつに対して、それがどのフィールドでどんな値が設定されているのかを調べていくのは、ちょっと不格好ですが、これが、ストリーミング・データとストリーミング API を処理する方法です。

コードに話を戻すと、file フィールドに達した時点で ByteArrayOutputStream の助けを借りて、アップロードされたデータが photoData に保存されます。最後に、PhotoDao によって Photo オブジェクトを保存してリダイレクトを送信すると、今度はリスト 10 に記載する最後の抽象メソッド、showRecord の出番となります。

リスト 10. showRecord メソッド

```
@Override
protected void showRecord(long id, HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException {
    Photo photo = dao.findById(id);

    resp.setContentType(photo.getContentType());
    resp.getOutputStream().write(photo.getPhotoData());
    resp.flushBuffer();
}
```

showRecord メソッドは Photo オブジェクトを検索し、content-type ヘッダーを設定してから photoData バイト配列を HTTP レスポンスに直接書き込みます。すると、flushBuffer メソッドが残りのコンテンツをブラウザに出力します。

最後に必要な作業は、1MB を超える大きさのアップロードに対するエラー処理コードを追加することです。

エラー・メッセージの表示

前述したように、画像に関わるほとんどの場合で、Bigtable が課す 1MB の制約を破らないようにするのは容易なことではありません。私たちにできることは、ユーザーに画像のサイズを変更して再試行するように伝えるのが、せいぜいのところです。リスト 11 にデモとして、GAE 例外がスローされると、単純に例外メッセージを表示するコードを記載します (これは、標準的なサーブレット固有のエラー処理であり、GAE 特有のエラー処理ではありません)。

リスト 11. エラーの発生

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class ErrorServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        String message = (String)
            req.getAttribute("javax.servlet.error.message");

        PrintWriter out = res.getWriter();
        out.write("<html>");
        out.write("<body>");
        out.write("<h1>An error has occurred</h1>");
        out.write("<br />" + message);
        out.write("</body>");
        out.write("</html>");
    }
}
```

web.xml には ErrorServlet と、この記事でこれから作成する他のサーブレットも併せて忘れずに登録してください。ErrorServlet を指すエラー・ページを登録するコードは、リスト 12 のとおりです。

リスト 12. エラーの登録

```
<servlet>
  <servlet-name>errorServlet</servlet-name>
  <servlet-class>
    info.johnwheeler.gaestorage.servlet.ErrorServlet
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>errorServlet</servlet-name>
  <url-pattern>/error</url-pattern>
</servlet-mapping>

<error-page>
  <error-code>500</error-code>
  <location>/error</location>
</error-page>
```

以上で、GAE データストアとしても知られる Bigtable の簡単な紹介は終わりです。Bigtable は GAE ストレージの選択肢のなかで最も直観的なデータストアですが、ファイルのサイズが難点となります。1 ファイルあたり 1MB に制限されるとあっては、サムネールより大きなデータはおろか、サムネールにさえ使用したいとは思わないでしょう。次に紹介する Blobstore は、最大 2GB のファイルの保存および取得に対応できる、もう 1 つのキー・バリュー型ストレージの選択肢です。

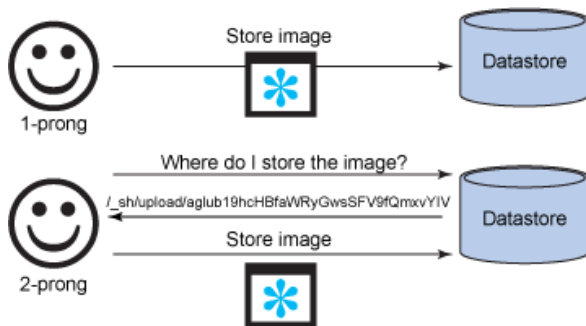
GAE ストレージの選択肢 2: Blobstore

Blobstore は、扱うことのできるデータ・サイズに関しては Bigtable よりも優れていますが、問題がないわけではありません。具体的には、Blobstore では 1 回限りのアップロード URL を使用しなければならないことです。そのため、Blobstore を中心に Web サービスを作成するのは簡単ではありません。以下に、アップロード URL の一例を記載します。

```
/_ah/upload/ag1ub19hcHBfaWRyGwsSFV9fQmxvY1VwbG9hZFN1c3Npb25fXxh9DA
```

Web サービスのクライアントが POST 送信するには、まず送信先の URL を要求しなければならないため、余分な呼び出しがネットワーク上に送信されることになります。これは、多くのアプリケーションでは大きな問題にならないかもしれませんが、完全に洗練されているとは言えません。さらに、クライアントが GAE 上で実行されていて CPU 時間で課金される場合には、法外な費用がかかることも考えられます。このような問題があっても、URL フェッチを使ってアップロードを 1 回限りの URL に転送するサーブレットを作成すれば対処できると考えているならば、考え直してください。URL フェッチには 1MB の転送量制限があります。したがって、その方向で考えているとしたら、Bigtable を使用することと同じことです。枠組みとして、図 5 に Web サービスの呼び出しが 1 回で済む場合と 2 回必要な場合の違いを示します。

図 5. Web サービスの呼び出しが 1 回で済む場合と 2 回必要な場合の違い



Blobstore に利点と欠点があることは、次のセクションでさらにわかってくるはずです。これから再びアップロード・フォームを作成し、`AbstractUploadServlet` が提供する 3 つの抽象メソッドを実装します。ただし今回は、Blobstore 用にコードを調整します。

Blobstore に対するアップロード・フォーム

Blobstore に対するアップロード・フォームを新たにゼロから作り直す必要はないので、datastore.jsp を blobstore.jsp という名前のファイルにコピーして、リスト 13 に太字で示す部分のコードの変更を反映してください。

リスト 13. blobstore.jsp

```
<% String uploadUrl = (String) request.getAttribute("uploadUrl"); %><html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <form method="POST" action="<%= uploadUrl %>"
      enctype="multipart/form-data">
    <!-- labels and fields omitted -->
    </form>
  </body>
</html>
```

1 回限りのアップロード URL は、サブレット (この後コーディングします) で生成されます。サブレットで生成された URL は、リクエストから構文解析されてフォームの action 属性に挿入されます。アップロード先の Blobstore サーバーについては、制御する方法がありませんが、そうすると、他のフォームの値はどのようにして取得するのでしょうか。その答えは、Blobstore API のコールバック・メカニズムにあります。1 回限りの URL が生成されるときにコールバック URL を API に渡すと、アップロードの完了後に Blobstore はそのコールバックを呼び出し、元のリクエストおよびアップロードされたすべての BLOB データを渡します。この動作の一部始終は、これから `AbstractUploadServlet` を実装するなかで理解できるはずです。

Blobstore へのアップロード

まずはリスト 14 を参考に、`AbstractUploadServlet` を継承する `BlobstoreUploadServlet` という名前のスタブ・クラスを作成してください。

リスト 14. BlobstoreUploadServlet

```
import info.johnwheeler.gaestorage.core.*;
import java.io.IOException;
import java.util.Map;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import com.google.appengine.api.blobstore.*;

@SuppressWarnings("serial")
public class BlobstoreUploadServlet extends AbstractUploadServlet {
    private BlobstoreService blobService =
        BlobstoreServiceFactory.getBlobstoreService();
    private PhotoDao dao = new PhotoDao();
}
```

最初のクラス定義は `DatastoreUploadServlet` と同様ですが、今回は `BlobstoreService` 変数が追加されています。この変数が、リスト 15 の `showForm` メソッドで 1 回限りの URL を生成します。

リスト 15. Blobstore の showForm メソッド

```
@Override
protected void showForm(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String uploadUrl = blobService.createUploadUrl("/blobstore");
    req.setAttribute("uploadUrl", uploadUrl);
    req.getRequestDispatcher("blobstore.jsp").forward(req, resp);
}
```

リスト 15 のコードでは、アップロード URL を作成して、それをリクエストの属性に設定します。そして、このアップロード URL を必要としている、リスト 13 で作成したフォームに処理を委譲します。コールバック URL は、`web.xml` に定義されているようにサーブレットのコンテキストに設定されます。したがって、Blobstore のアップロード・フォームから POST が返されると、リスト 16 に示す `handleSubmit` メソッドに制御が移ります。

リスト 16. Blobstore の handleSubmit メソッド

```
@Override
protected void handleSubmit(HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException {
    Map<String, BlobKey> blobs = blobService.getUploadedBlobs(req);
    BlobKey blobKey = blobs.get(blobs.keySet().iterator().next());

    String photoPath = blobKey.getKeyString();
    String title = req.getParameter("title");
    String caption = req.getParameter("caption");

    Photo photo = new Photo(title, caption);
    photo.setPhotoPath(photoPath);
    dao.save(photo);

    resp.sendRedirect("blobstore?action=display&id=" + photo.getId());
}
```

`getUploadedBlobs` は `BlobKeys` の Map を返します。アップロード・フォームがサポートしているのは、単一のアップロードであるため、必要とされる唯一の `BlobKey` を取得し、そのストリング表現を `photoPath` 変数に格納します。その後、残りのフィールドが構文解析されて変数に入れられ、新規 `Photo` インスタンスに設定されます。このインスタンスがデータストアに保存されたら、リスト 17 の `showRecord` メソッドにリダイレクトされます。

リスト 17. Blobstore の showRecord メソッド

```
@Override
protected void showRecord(long id, HttpServletRequest req,
    HttpServletResponse resp) throws ServletException, IOException {
    Photo photo = dao.findById(id);
    String photoPath = photo.getPhotoPath();

    blobService.serve(new BlobKey(photoPath), resp);
}
```

showRecord メソッドでは、上記の handleSubmit メソッドで保存した Photo オブジェクトを Blobstore からロードします。Bigtable の場合とは異なり、アップロードされた実際のバイトは、Bean には保存されていません。代わりに、photoPath を使用して BlobKey を再作成し、画像をブラウザに提供するためにこの BlobKey を使用します。

Blobstore では、フォーム・ベースのアップロードをごく簡単に扱えるようになっていますが、Web サービス・ベースのアップロードとなると別の話です。次に調べる Google Storage for Developers は、これとはまったく正反対の難問を私たちに突き付けます。つまり、Web サービス・ベースのアップロードは簡単に処理できる一方、フォーム・ベースのアップロードを扱うには多少の工夫が必要になってきます。

GAE ストレージの選択肢 3: Google Storage

Google Storage for Developers は、GAE ストレージの 3 つの選択肢のなかで最も強力であり、いくつかの問題を片付けさえすれば簡単に使用することができます。Google Storage には Amazon S3 との共通点がたくさんあります。実際、この 2 つは同じプロトコルを使用し、どちらも RESTful なインターフェースを備えているため、S3 で機能するように作成されたライブラリー (例えば、JetS3t) は Google Storage でも機能します。しかし、この記事が執筆している時点では、これらのライブラリーの Google App Engine での動作は確実ではありません。それは、ライブラリーによって、スレッドの生成などの許可されていない処理が実行されるためです。したがって、今のところは RESTful なインターフェースを使って、これらの API に任せられた面倒な作業の一部を処理するしかありません。

けれども Google Storage には、その面倒に見合うだけの価値があります。その主な理由は、アクセス制御リスト (ACL) による強力なアクセス制御がサポートされることです。ACL によって、読み取り専用アクセス権や、読み取り/書き込みアクセス権をオブジェクトに付与することができるため、Facebook や Flickr のように簡単に写真を公開したり、個人用に設定したりすることが可能です。ACL については、この記事では説明しないので、ここでアップロードするあらゆるデータは公開されて、読み取り専用で設定されることを前提とします。ACL の詳細を学ぶには、Google Storage のオンライン・ドキュメント ([参考文献](#)) を参照してください。

Google Storageについて

2010年5月にプレビュー版としてリリースされた Google Storage for Developers は、この記事の執筆時点では米国の限られた数の開発者のみに利用が制限されていて、プレビュー版の利用希望者がその順番を待っている状態です。Google Storageはまだ初期段階にあるため、実装するにはいくつかの問題 (このセクションで対処する問題) を伴います。Google Storage と GAE を統合する明確な手段がないということは、コーディング作業がさらに必要になることを意味しますが、場合によっては (アクセス制御が必要な場合など)、その作業を行うだけの価値はあります。近い将来、統合ライブラリーが登場することを願います。

Blobstore とは異なり、Google Storage はデフォルトで Web サービス・クライアントおよびブラウザー・クライアントでの使用に対応しており、データは RESTful な PUT または POST で送信するようになっています。PUT の場合、リクエストの構成方法とヘッダーの作成方法を Web サービス・クライアントが制御することができます。POST の場合、この記事で取り上げる、ブラウザー・ベースのアップロード用です。アップロード・フォームを処理するには、JavaScript による改変が必要となりますが、この後わかるように、これが事態を複雑にする要因です。

Google Storage アップロード・フォームの改変

Google Storage は、Blobstore とは異なり、POST 送信されたアップロード・データの処理完了後、コールバック URL に転送を行いません。代わりに、ユーザーが指定する URL にリダイレクトします。問題はここにあります。なぜなら、リダイレクトではフォームの値が渡されないためです。この問題に対処するには、同じ 1 つの Web ページに 2 つのフォームを作成するという方法があります。一方のフォームにはタイトルとキャプションのテキスト・フィールドを含め、もう一方のフォームに、ファイル・アップロード・フィールドと必須の Google Storage パラメーターを含めます。そして、最初のフォームを Ajax を使用して送信し、Ajax コールバックが呼び出された時点で 2 番目のアップロード・フォームを送信するという方法です。

このフォームはここまでで説明した他の 2 つのフォームよりも複雑なので、順を追って作成していきます。まず始めに、処理の委譲元であるサーブレット (まだ作成していません) で設定された値を抽出します (リスト 18 を参照)。

リスト 18. フォームの値の抽出

```
<%  
String uploadUrl = (String) request.getAttribute("uploadUrl");  
String key = (String) request.getAttribute("key");  
String successActionRedirect = (String)  
    request.getAttribute("successActionRedirect");  
String accessId = (String) request.getAttribute("GoogleAccessId");  
String policy = (String) request.getAttribute("policy");  
String signature = (String) request.getAttribute("signature");  
String acl = (String) request.getAttribute("acl");  
%>
```

uploadUrl 変数には、Google Storage の REST エンドポイントが格納されます。API で提供しているエンドポイントは、以下に示す 2 つです。どちらも許容されますが、斜体で示されているコンポーネントは、それぞれの場合に応じた適切な内容に置き換える必要があります。

- bucket.comondatastorage.googleapis.com/object
- comondatastorage.googleapis.com/bucket/object

残りの変数は、Google Storage の必須パラメーターです。

- key: Google Storage にアップロードするデータの名前
- success_action_redirect: アップロード完了時のリダイレクト先
- GoogleAccessId: Google によって割り当てられた API キー
- policy: Base 64 方式でエンコードされた JSON スtring。データのアップロード方法を制約します。

- signature: ハッシュ・アルリズムと Base 64 方式でエンコードされた署名付きポリシー。認証に使用されます。
- acl: アクセス制御リスト仕様

2つのフォームと送信ボタン

リスト 19 に示す最初のフォームに含まれるのは、タイトル・フィールドとキャプション・フィールドだけです。フォームをラップする `<html>` タグと `<body>` タグは省略されています。

リスト 19. 最初のアップロード・フォーム

```
<form id="fieldsForm" method="POST">
  <table>
    <tr>
      <td>Title</td>
      <td><input type="text" name="title" /></td>
    </tr>
    <tr>
      <td>Caption</td>
      <td>
        <input type="hidden" name="key" value="%= key %>" />
        <input type="text" name="caption" />
      </td>
    </tr>
  </table>
</form>
```

このフォームに関しては、このフォームが POST 送信する先がこのフォーム自身であることを除けば、特に説明することはありません。そこで、リスト 20 に示されている 2 番目のフォームの説明に移ります。このフォームには 6 つの隠し入力フィールドが含まれているため、最初のフォームよりもコードが長くなっています。

リスト 20. 隠しフィールドが含まれる 2 番目のフォーム

```
<form id="uploadForm" method="POST" action="%= uploadUrl %>"
  enctype="multipart/form-data">
  <table>
    <tr>
      <td>Upload</td>
      <td>
        <input type="hidden" name="key" value="%= key %>" />
        <input type="hidden" name="GoogleAccessId"
          value="%= accessId %>" />
        <input type="hidden" name="policy"
          value="%= policy %>" />
        <input type="hidden" name="acl" value="%= acl %>" />
        <input type="hidden" id="success_action_redirect"
          name="success_action_redirect"
          value="%= successActionRedirect %>" />
        <input type="hidden" name="signature"
          value="%= signature %>" />
        <input type="file" name="file" />
      </td>
    </tr>
    <tr>
      <td colspan="2">
        <input type="button" value="Submit" id="button"/>
      </td>
    </tr>
  </table>
</form>
```

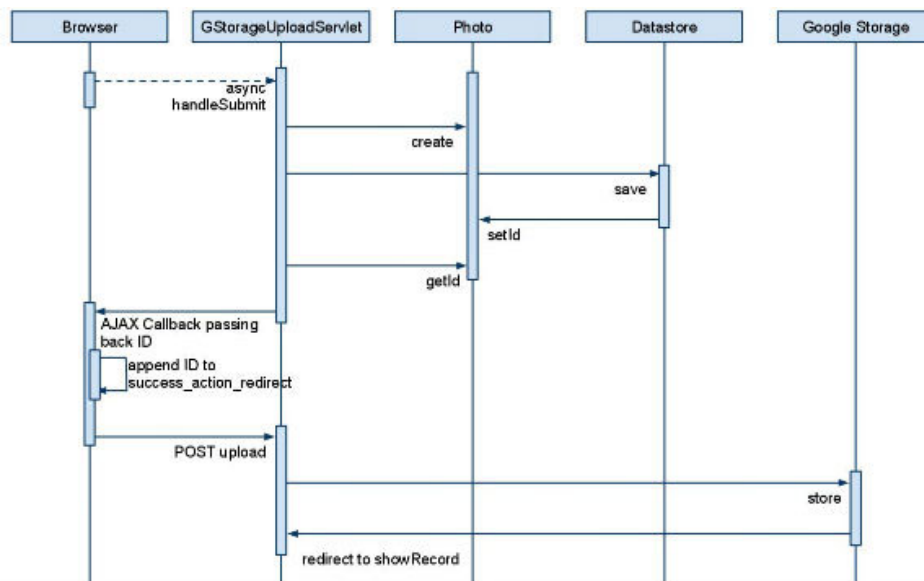
JSP スクリプトレット (リスト 18) で抽出された値は、これらの隠しフィールドに取り込まれます。ファイル入力が一番下にあります。送信ボタンはありきたりなもので、リスト 21 に記載する JavaScript を実装することで、初めてボタンとして機能するようになります。

リスト 21. アップロード・フォームの送信

```
<script type="text/javascript"
src="https://Ajax.googleapis.com/Ajax/libs/jquery/1.4.3/jquery.min.js">
</script>
<script type="text/javascript">
    $(document).ready(function() {
        $('#button').click(function() {
            var formData = $('#fieldsForm').serialize();
            var callback = function(photoId) {
                var redir = $('#success_action_redirect').val() +
                    photoId;
                $('#success_action_redirect').val(redir);
                $('#uploadForm').submit();
            };
            $.post("gstorage", formData, callback);
        });
    });
</script>
```

リスト 21 の JavaScript は jQuery を使用して作成されています。このライブラリーを使ったことがないとしても、上記のコードを理解するのは難しくないはずです。コードが最初に実行するのは、jQuery のインポートです。次に、クリック・リスナーをボタンにインストールして、ボタンがクリックされると最初のフォームが Ajax によって送信されるようにします。そこからは、サーブレットの `handleSubmit` メソッド (この後すぐに作成します) に制御が移り、そこで `Photo` オブジェクトが作成されてデータストアに保存されます。そして最後に、新規 `Photo ID` がコールバックに返され、`success_action_redirect` の URL にその末尾に `Photo ID` が追加されてから、アップロード・フォームが送信されます。これで、リダイレクトによって戻ったときに `Photo` オブジェクトを検索すれば、その画像を表示することができるというわけです。図 6 に、以上のイベント・シーケンスを示します。

図 6. JavaScript 呼び出しパスを示すシーケンス図



フォームの処理は完了したので、今度はポリシー文書を作成して署名を付けるためのユーティリティ・クラスが必要です。ユーティリティ・クラスを作成したら、AbstractUploadServlet のサブクラス化に取り組むことができます。

ポリシー文書の作成と署名の付与

ポリシー文書は、アップロードを制約するために作成します。例えば、アップロードの最大サイズやファイルの許容タイプを指定したり、ファイル名に関する制約を設けたりすることができます。パブリック・バケットにポリシー文書は必要ありませんが、Google Storage のようなプライベート・バケットには必要です。ポリシー文書を機能させるためにはまず、リスト 22 に記載するコードを基に、GSUtils という名前のユーティリティ・クラスのスタブを作成してください。

リスト 22. GSUtils クラス

```

import java.io.UnsupportedEncodingException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;

import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

import com.google.appengine.repackaged.com.google.common.util.Base64;

private class GSUtils {
}
  
```

ユーティリティ・クラスは、静的メソッドだけで構成されるのが通常であるため、デフォルト・コンストラクターをプライベート・コンストラクターにして、インスタンス化できないようにするのが賢明です。ユーティリティ・クラスのスタブを作成したら、今後はポリシー文書の作成に取り掛かります。

このポリシー文書は JSON フォーマットですが、JSON は単純なので、何らかの巧妙なライブラリーに頼る必要はありません。ライブラリーを使用する代わりに、単純な `StringBuilder` を使って手作業で文書の内容を作り上げていくことができます。その場合、最初に行わなければならない作業は、ISO8601 形式の日付を設定し、その日付になるとポリシー文書の有効期限が切れるように設定することです。ポリシー文書の有効期限が切れた後は、アップロードは成功しません。次に、前述の制約をポリシー文書に組み込みます。これらの制約は、ポリシー文書では条件 (condition) と呼ばれます。最後に、ポリシー文書を Base 64 方式でエンコードして、呼び出し側に返します。

`GSUtils` にリスト 23 のメソッドを追加してください。

リスト 23. ポリシー文書の作成

```
public static String createPolicyDocument(String acl) {
    GregorianCalendar gc = new GregorianCalendar();
    gc.add(Calendar.MINUTE, 20);

    DateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss'Z'");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    String expiration = df.format(gc.getTime());

    StringBuilder buf = new StringBuilder();
    buf.append("{\"expiration\": \"");
    buf.append(expiration);
    buf.append("\", \"conditions\": [");
    buf.append(", {\"acl\": \"");
    buf.append(acl);
    buf.append("\", \"");
    buf.append("[\"starts-with\", \"$key\", \"\"]");
    buf.append(", [\"starts-with\", \"$success_action_redirect\", \"\"]");
    buf.append("]}");

    return Base64.encode(buf.toString().replaceAll("\n", "").getBytes());
}
```

上記で使用している `GregorianCalendar` は、有効期限を 20 分後に設定しています。これはその場しのぎのコードなので、このコードをコンソールに出力してコピーし、JSONLint (「[参考文献](#)」を参照) のようなツールで実行することになります。次に、`acl` をポリシー文書に渡します。これは、アクセス制御リストをハード・コーディングするのを避けるためです。すべての変数は、`acl` のように、メソッド引数として渡さなければなりません。最後に、ポリシー文書を呼び出し側に返す前に Base 64 方式でエンコードすれば、ポリシー文書は完成です。ポリシー文書に許容される内容についての詳細は、Google Storage のドキュメントを参照してください。

Google の Secure Data Connector

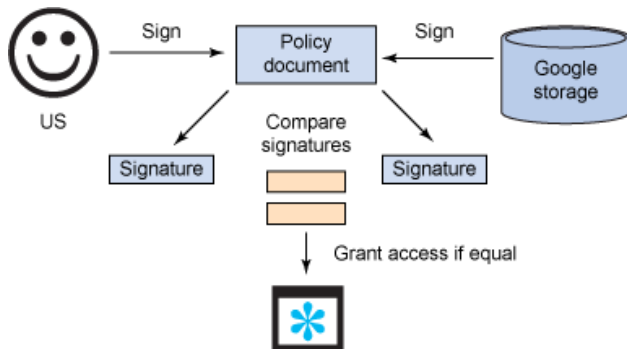
この記事では Google の Secure Data Connector (SDC) を使用しませんが、Google Storage を使用することを計画している場合には SDC について調べておく価値があります。SDC によって、ファイアウォールで保護されているシステム上のデータにも、簡単にアクセスできるようになります。

Google Storage での認証

ポリシー文書が果たす役割は 2 つあります。それは、ポリシー文書はポリシーを施行するだけでなく、アップロードを認証するために生成する署名のベースにもなるからです。Google Storage に登録すると、登録したユーザー本人と Google しか知らないシークレット・キーが提供されま

す。ユーザー側ではこのシークレット・キーを使って文書に署名を付け、Google 側でも同じキーを使って文書に署名を付けます。2つの署名が一致すれば、アップロードが許可されるという仕組みです。図7に、この認証サイクルをわかりやすく示します。

図7. アップロードが Google Storage に認証される仕組み



署名を生成するためには、GSUtils のスタブを作成したときにインポートした `javax.crypto` および `java.security` パッケージを使用します。リスト24に、この2つを使用して署名を生成するメソッドを示します。

リスト24. ポリシー文書の署名

```

public static String signPolicyDocument(String policyDocument,
    String secret) {
    try {
        Mac mac = Mac.getInstance("HmacSHA1");
        byte[] secretBytes = secret.getBytes("UTF8");
        SecretKeySpec signingKey =
            new SecretKeySpec(secretBytes, "HmacSHA1");
        mac.init(signingKey);
        byte[] signedSecretBytes =
            mac.doFinal(policyDocument.getBytes("UTF8"));
        String signature = Base64.encode(signedSecretBytes);
        return signature;
    } catch (InvalidKeyException e) {
        throw new RuntimeException(e);
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(e);
    } catch (UnsupportedEncodingException e) {
        throw new RuntimeException(e);
    }
}

```

Java コードでセキュア・ハッシュの計算をするには煩わしい作業を伴うので、この記事では敢えて省略します。重要なのは、[リスト24](#)に示すように適切に署名を付ける方法と、ハッシュを返す前に Base 64 方式でエンコードしなければならないことです。

これで前提条件が整ったので、おなじみの作業に戻ります。つまり、Google Storage にファイルをアップロードして、そこからファイルを取得するための3つの抽象メソッドを実装する作業です。

Google Storage へのアップロード

リスト25のコードを基に、`GStorageUploadServlet` という名前のスタブ・クラスを作成することから始めます。

リスト 25. GStorageUploadServlet

```
import info.johnwheeler.gaestorage.core.GSUtils;
import info.johnwheeler.gaestorage.core.Photo;
import info.johnwheeler.gaestorage.core.PhotoDao;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.UUID;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class GStorageUploadServlet extends AbstractUploadServlet {
    private PhotoDao dao = new PhotoDao();
}
```

リスト 26 に記載する `showForm` メソッドが、アップロード・フォームを介して Google Storage に渡さなければならないパラメーターを設定します。

リスト 26. Google Storage の showForm メソッド

```
@Override
protected void showForm(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    String acl = "public-read";
    String secret = getServletConfig().getInitParameter("secret");
    String accessKey = getServletConfig().getInitParameter("accessKey");
    String endpoint = getServletConfig().getInitParameter("endpoint");
    String successActionRedirect = getBaseUrl(req) +
        "gstorage?action=display&id=";
    String key = UUID.randomUUID().toString();
    String policy = GSUtils.createPolicyDocument(acl);
    String signature = GSUtils.signPolicyDocument(policy, secret);

    req.setAttribute("uploadUrl", endpoint);
    req.setAttribute("acl", acl);
    req.setAttribute("GoogleAccessId", accessKey);
    req.setAttribute("key", key);
    req.setAttribute("policy", policy);
    req.setAttribute("signature", signature);
    req.setAttribute("successActionRedirect", successActionRedirect);

    req.getRequestDispatcher("gstorage.jsp").forward(req, resp);
}
```

`acl` は `public-read` に設定されていることに注意してください。つまり、アップロードされるすべてのものは、誰でも表示できるということです。次の `secret`、`accessKey`、`endpoint` という 3 つの変数は、Google Storage にアクセスして認証するために使用されます。これらの変数は、`web.xml` で宣言されている初期パラメーター (`init-param`) から抽出されます (詳細は[サンプル・コード](#)を参照)。Blobstore ではアップロード・データの処理完了後、コールバック URL に転送を行い、そこで `showRecord` メソッドに制御が移りますが、それとは異なり、Google Storage はリダイレクトを実行すると説明したことを思い出してください。このリダイレクト URL が格納される場所は、`successActionRedirect` です。`successActionRedirect` は、リスト 27 に記載するヘルパー・メソッドを使用してリダイレクト URL を組み立てます。

リスト 27. getBaseUrl() メソッド

```
private static String getBaseUrl(HttpServletRequest req) {  
    String base = req.getScheme() + "://" + req.getServerName() + ":" +  
        req.getServerPort() + "/";  
    return base;  
}
```

このヘルパー・メソッドは、受信されるリクエストをポーリングして基本 URL を組み立てた後、制御を `showForm` メソッドに戻します。制御が戻ると `showForm` メソッドでは、UUID (Universally Unique Identifier) を使用してキーが作成されます。UUID は、一意であることが保証された `String` です。次に、先ほど作成したユーティリティ・クラスによってポリシーと署名が生成されます。そして最後に、リクエストの各属性をアップロード・フォームである JSP 用に設定してから、そのアップロード・フォームへ処理を委譲します。

リスト 28 に `handleSubmit` メソッドを記載します。

リスト 28. Google Storage の handleSubmit メソッド

```
@Override  
protected void handleSubmit(HttpServletRequest req, HttpServletResponse  
    resp) throws ServletException, IOException {  
    String endpoint = getServletConfig().getInitParameter("endpoint");  
    String title = req.getParameter("title");  
    String caption = req.getParameter("caption");  
    String key = req.getParameter("key");  
  
    Photo photo = new Photo(title, caption);  
    photo.setPhotoPath(endpoint + key);  
    dao.save(photo);  
  
    PrintWriter out = resp.getWriter();  
    out.println(Long.toString(photo.getId()));  
    out.close();  
}
```

最初のフォームが送信された時点で、Ajax による POST によって `handleSubmit` メソッドに制御が移ることを思い出してください。アップロード自体は、このメソッドで処理されるのではなく、別途 Ajax コールバックで処理されます。`handleSubmit` メソッドは最初のフォームを構文解析し、`Photo` オブジェクトを作成してデータストアに保存するだけに過ぎません。その後、`Photo` オブジェクトの ID がレスポンス本体に書き込まれて、Ajax コールバックに返されます。

コールバックでは、アップロード・フォームが Google Storage エンドポイントに送信されます。Google Storage は、アップロード・データの処理を完了すると、リダイレクトを `showRecord` メソッド (リスト 29 を参照) に送信するようにセットアップされます。

リスト 29. Google Storage の showRecord メソッド

```
@Override  
protected void showRecord(long id, HttpServletRequest req,  
    HttpServletResponse resp) throws ServletException, IOException {  
    Photo photo = dao.findById(id);  
    String photoPath = photo.getPhotoPath();  
    resp.sendRedirect(photoPath);  
}
```

`showRecord` メソッドは `Photo` オブジェクトを検索し、その `photoPath` にリダイレクトします。`photoPath` は Google のサーバー上でホストされる画像を指しています。

まとめ

この記事では、Google を中心とした 3 つのストレージの選択肢を検討し、それぞれの利点と欠点を評価しました。Bigtable は簡単に扱うことができるものの、ファイル・サイズが 1MB に制限されます。Blobstore では各 BLOB につき、最大 2GB まで許可されますが、Web サービスで 1 回限りの URL に対処するのに苦労します。最後の選択肢である Google Storage for Developers は、3 つのうちで最も堅牢なオプションです。このストレージは、使用した分だけ料金が発生し、1 つのファイルに保存できるデータの量に制限はありません。その一方、Google Storage のライブラリーは現在 GAE をサポートしていないため、Google Storage は最も扱うのが複雑なソリューションです。また、ブラウザー・ベースのアップロードをサポートするのも、そう簡単ではありません。

Java 開発者の開発プラットフォームとして Google App Engine の人気が高まるなか、そのさまざまなストレージの選択肢を理解しておくことが重要です。この記事では、Bigtable、Blobstore、そして Google Storage for Developers の単純な実装例を紹介しました。特定のストレージを使用することに決めて、それを使用し続けるにしても、あるいは場合によって使用するストレージを使い分けるにしても、GAE に大量のデータを保存するために必要な知識は身に付いたはずですよ。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-gaestorage.zip	12KB

著者について

John Wheeler



John Wheeler は、10 年以上プロとして活躍しているプログラマーです。彼は『Spring in Practice』の共著者であり、アプリケーション・マネージャーとして Xerox に勤務しています。彼の [Web サイト](#) にアクセスして、ソフトウェア開発に関する他の記事を調べてください。

© Copyright IBM Corporation 2012

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)