GOOGLE APP ENGINE

CHAPTER 7

PHP: STORING DATA

Storing Data in PHP

The App Engine environment provides multiple options to store data for your application:

- Google Cloud SQL: A relational SQL database for your App Engine application, based on the familiar MySQL database.
- Google Cloud Storage: A storage service for objects and files up to terabytes in size, and accessible to App Engine apps via the Google Cloud Storage client library.

Name	Structure	Consistency	Cost
Google Cloud SQL	Relational	Strongly consistent	Google offers two billing plans for Google Cloud SQL: Packages and Per Use. More information is available in theCloud SQL price sheet.
Google Cloud Storage (GCS)	Files and their associated metadata	Strongly consistent except when performing list operations that get a list of buckets or objects.	There are no charges associated with making calls to Google Cloud Storage. However, any data stored in GCS is charged the usual GCS data storage fees. Cloud Storage prices are available on the Cloud Storage price sheet.

Google Cloud SQL

Google Cloud SQL is a web service that allows you to create, configure, and use relational databases that live in Google's cloud. It is a fully-managed service offering the capabilities of a MySQL database, allowing you to focus on application development.

For more information, see Google's Cloud SQL Overview.

Google Cloud Storage

Google Cloud Storage is useful for storing and serving large files. Additionally, Cloud Storage offers the use of access control lists (ACLs), the ability to resume upload operations if they're interrupted, and many other features. The GCS client library makes

use of this resume capability automatically for your app, providing you with a robust way to stream data into GCS.

There are no charges associated with making calls to Google Cloud Storage. However, any data stored at GCS is charged the usual GCS data storage fees, which are listed on the Cloud Storage price sheet.

Applications connect to Google Cloud Storage using the Google Cloud Storage Client Library.

Using Google Cloud SQL

This document describes how to use Google Cloud SQL instances with the App Engine PHP SDK.

To learn more about Google Cloud SQL, see the Google Cloud SQLdocumentation. If you haven't already created a Google Cloud SQL instance, the first thing you need to do is create one.

Creating a Cloud SQL Instance

A Cloud SQL instance is equivalent to a server. A single Cloud SQL instance can contain multiple databases. Follow these steps to create a Google Cloud SQL instance:

Sign into the Google Developers Console.

1.Create a new project, or open an existing project.

2.From within a project, select **Cloud SQL** to open the Cloud SQL control panel for that project.

3.Click **New Instance** to create a new Cloud SQL instance in your project, and configure your size, billing and replication options.

- More information on Cloud SQL billing options and instance sizes
- More information on Cloud SQL replication options

You'll notice that the App Engine application associated with your current project is already authorized to access this new instance. For more information on app authorization see the Access Control topic in the Cloud SQL docs.

That's it! You can now connect to your Google Cloud SQL instance from within your app, or any of these other methods.

MySQL case sensitivity

When you are creating or using databases and tables, keep in mind that all identifiers in Google Cloud SQL are case-sensitive. This means that all tables and databases are stored with the same name and case that was specified at creation time. When you try to access your databases and tables, make sure that you are using the exact database or table name.

For example, if you create a database named PersonsDatabase, you will not be able to reference the database using any other variation of that name, such as personsDatabase or personsdatabase. For more information about identifier case sensitivity, see the MySQL documentation.

Build a starter application and database

The easiest way to build an App Engine application that accesses Google Cloud SQL is to create a starter application then modify it. This section leads you through the steps of building an application that displays a web form that lets users read and write entries to a guestbook database. The sample application demonstrates how to read and write to a Google Cloud SQL instance.

Step 1: Create your App Engine sample application

Follow the instructions for the Hello World! chapter of the PHP Getting Started guide to create a simple App Engine application.

Step 2: Grant your App Engine application access to the Google Cloud SQL instance

You can grant individual Google App Engine applications access to a Google Cloud SQL instance. One application can be granted access to multiple instances, and multiple applications can be granted access to a particular instance. To grant access to a Google App Engine application, you need its application ID which can be found at the Google App Engine administration console under the Applications column.

Note: An App Engine application must be in the same region (either EU or US) as a Google Cloud SQL instance to be authorized to access that Google Cloud SQL instance.

To grant an App Engine application access to a Google Cloud SQL instance:

1. Go to the Google Developers Console.

2.Select a project by clicking the project name.

3.In the left sidebar of your project, click Cloud SQL.

4.Click the name of the instance to which you want to grant access.

5.In the instance dashboard, click Edit.

6.In the Instance settings window, enter your Google App Engine application ID in the Authorized App Engine applications section. You can grant access to multiple applications, by entering them one at a time.

Note: In order to improve performance, the Google Cloud SQL instance will be kept as close as possible to the first App Engine application on the list, so this should be the application whose performance is most important.

7.Click Confirm to apply your changes.

After you have added authorized applications to your Google Cloud SQL instance, you can view a list of these applications in the instance dashboard, in the section titled Applications.

Step 3: Create your database and table

For example, you can use MySQL Client to run the following commands:

1.Create a new database called `guestbook` using the following SQL statement:

```
CREATE DATABASE guestbook;
```

2.Inside the `guestbook` database create a table called `entries` with columns for the guest name, the message content, and a random ID, using the following statement:

```
CREATE TABLE guestbook.entries (
  entryID INT NOT NULL AUTO_INCREMENT,
  guestName VARCHAR(255),
  content VARCHAR(255),
  PRIMARY KEY(entryID)
);
```

After you have set up a bare-bones application, you can modify it and deploy it.

Connect to your database

1. Working with different connection interfaces
2. Connect, post, and get from your database
3. Update your configuration file

Working with different connection interfaces

Google Cloud SQL supports connections from PHP using common connection methods, including PDO_MySQL,mysqli, and MySQL API. PDO_MySQL, mysqli, and MySQL API are enabled by default in App Engine, in development (locally) or in your production (deployed App Engine application).

In these connection examples, an App Engine app that belongs to a Google Cloud Platform project called `<your-project-id>` is connecting to a Cloud SQL instance named `<your-instance-name>`. These following examples show how to connect from a deployed App Engine application using a socket or named pipe that specifies the Cloud SQL instance. When you connect from App Engine, you can use the root user and no password (as shown here), or you can use a specific database user and password.

PDO_MySQL (connecting from App Engine)

```
$db = new pdo('mysql:unix_socket=/cloudsql/<your-project-id>:<your-instance-name>;dbname=<database-name>',

  'root',  // username

  ''       // password

  );
```

mysqli (connecting from App Engine)

```
$sql = new mysqli(null,
  'root', // username
  '',     // password
  <database-name>,
  null,
  '/cloudsql/<your-project-id>:<your-instance-name>'
  );
```

MySQL API (connecting from App Engine)

```
$conn = mysql_connect(':/cloudsql/<your-project-id>:<your-instance-name>',
  'root', // username
  ''      // password
  );
mysql_select_db('<database-name'>);
```

These connection examples cannot be used from your development environment. To connect from your development environment to either a local MySQL instance or a Cloud SQL instance, use the following connection code:

PDO_MySQL (connecting using an IP address)

```
$db = new pdo('mysql:host=127.0.0.1:3306;dbname=<database-name>',
  '<username>',
  '<password>'
);
```

mysqli (connecting using an IP address)

```
$sql = new mysqli('127.0.0.1:3306',
  '<username>',
  '<password>',
  <database-name>
  );
```

MySQL API (connecting using an IP address)

```
$conn = mysql_connect('127.0.0.1:3306',
  '<username>',
  '<password>'
  );
mysql_select_db('<database-name>');
```

To connect to a Cloud SQL instance from your development environment, substitute "127.0.0.1" with the instance IP address. You do not use the "/cloudsql/"-based connection string to connect to a Cloud SQL instance if your App Engine app is running locally in the Development Server.

If you want to use the same code locally and deployed, you can use a Special $_SERVER keys variable (`SERVER_SOFTWARE`) to determine where your code is running. This approach is shown below.

Connect, post, and get from your database

Create two files `guestbook.php` and `sign.php`. `guestbook.php` displays the current entries in the guestbook and accepts input to add a new entry. When you click the Sign Guestbook button on the `guestbook.php` page, the `sign.php` page is invoked, which adds the entry and then redirects back to the `guestbook.php` page.

In both files, you access a Cloud SQL instance by using a Unix socket with the prefix `/cloudsql/`. In the connection code in both files, replace `<your-project-id>` and `<your-instance-name>` with the project ID of your Google Cloud Project, and the name of your Cloud SQL instance.

Here is guestbook.php:

```
<html>
 <body>
  <?php
  use google\appengine\api\users\User;
  use google\appengine\api\users\UserService;
  $user = UserService::getCurrentUser();
  if ($user) {
    echo 'Hello, ' . htmlspecialchars($user->getNickname());
  }
  else {
    header('Location: ' .
UserService::createLoginURL($_SERVER['REQUEST_URI']));
  }
  ?>
  <h2>Guestbook Entries</h2>
  <?php
  // Create a connection.
  $db = null;
  if (isset($_SERVER['SERVER_SOFTWARE']) &&
  strpos($_SERVER['SERVER_SOFTWARE'],'Google App Engine') !== false) {
    // Connect from App Engine.
    try{
       $db = new pdo('mysql:unix_socket=/cloudsql/<your-project-id>:<your-
instance-name>;dbname=guestbook', 'root', '');
    }catch(PDOException $ex){
        die(json_encode(
            array('outcome' => false, 'message' => 'Unable to connect.')
            )
        );
    }
  } else {
    // Connect from a development environment.
    try{
       $db = new pdo('mysql:host=127.0.0.1:3306;dbname=guestbook', 'root',
'<password>');
    }catch(PDOException $ex){
        die(json_encode(
            array('outcome' => false, 'message' => 'Unable to connect')
            )
        );
```

```
    }
  }
  try {
    // Show existing guestbook entries.
    foreach($db->query('SELECT * from entries') as $row) {
            echo "<div><strong>" . $row['guestName'] . "</strong> wrote <br> "
. $row['content'] . "</div>";
     }
  } catch (PDOException $ex) {
    echo "An error occurred in reading or writing to guestbook.";
  }
  $db = null;
  ?>
  <h2>Sign the Guestbook</h2>
  <form action="/sign" method="post">
    <div><textarea name="content" rows="3" cols="60"></textarea></div>
    <div><input type="submit" value="Sign Guestbook"></div>
  </form>
  </body>
</html>
```

Here is `sign.php`:

```
<?php
header('Location: '."/guestbook");
use google\appengine\api\users\User;
use google\appengine\api\users\UserService;
$user = UserService::getCurrentUser();
$db = null;
if (isset($_SERVER['SERVER_SOFTWARE']) &&
strpos($_SERVER['SERVER_SOFTWARE'],'Google App Engine') !== false) {
  // Connect from App Engine.
  try{
     $db = new pdo('mysql:unix_socket=/cloudsql/<your-project-id>:<your-
instance-name>;dbname=guestbook', 'root', '');
  }catch(PDOException $ex){
      die(json_encode(
          array('outcome' => false, 'message' => 'Unable to connect.')
          )
      );
  }
} else {
```

```
  // Connect from a development environment.
  try{
     $db = new pdo('mysql:host=127.0.0.1:3306;dbname=guestbook', 'root',
'<password>');
  }catch(PDOException $ex){
      die(json_encode(
          array('outcome' => false, 'message' => 'Unable to connect')
          )
      );
  }
}
try {
  if (array_key_exists('content', $_POST)) {
    $stmt = $db->prepare('INSERT INTO entries (guestName, content) VALUES
(:name, :content)');
    $stmt->execute(array(':name' => htmlspecialchars($user->getNickname()),
':content' => htmlspecialchars($_POST['content'])));
    $affected_rows = $stmt->rowCount();
    // Log $affected_rows.
  }
} catch (PDOException $ex) {
  // Log error.
}
$db = null;
?>
```

The connection code shown above connects to the Google Cloud SQL instance as the `root` user but you can connect to the instance as a specific database user. For more information, see Working with different connection interfaces.

For information about creating MySQL users, see Adding Users in the MySQL documentation.

Managing connections

An App Engine application is made up of one or more modules. Each module consists of source code and configuration files. An instance instantiates the code which is included in an App Engine module, and a particular version of module will have one or more instances running. The number of instances running depends on the number of incoming requests. You can configure App Engine to scale the number of instances automatically in response to processing volume (see Instance scaling and class).

When App Engine instances talk to Google Cloud SQL, each App Engine instance cannot

have more than 12 concurrent connections to a Cloud SQL instance. Always close any established connections before finishing processing a request. Not closing a connection will cause it to leak and may eventually cause new connections to fail. You can exit this state by shutting down the affected App Engine instance.

You should also keep in mind that there is also a maximum number of concurrent connections and queries for each Cloud SQL instance, depending on the tier (see Cloud SQL pricing). For guidance on managing connections, see How should I manage connections? in the "Google Cloud SQL FAQ" document.

Update your configuration file

In your `app.yaml` file, make sure you have script handlers for `guestbook.php` and `sign.php`.

```
application: <your-application-name>

version: 1

runtime: php

api_version: 1

handlers:

- url: /

  script: guestbook.php

- url: /sign

  script: sign.php
```

That's it! Now you can deploy your application and try it out!

Using a local MySQL instance during development

The above example shows how to connect to a Cloud SQL instance when the code runs in App Engine and how to connect to a local MySQL server when the code runs in the Development Server. We encourage this pattern to minimize confusion and also maximize flexibility.

Size and access limits

The following limits apply to Google Cloud SQL:

Instance Connections

- Each tier allows for maximum concurrent connections and queries. For more information, see the pricingpage.
- There is a limit of 100 pending connections independent of tier.
- Establishing a connection takes, on the server side, about 1.25 ms; because of the 100 pending connection limit, this means a maximum of 800 connection per second. If more than 100 clients try to connect simultaneously then some them will fail.

These limits are in place to protect against accidents and abuse. For questions about increasing these values, contact the cloud-sql@google.com team.

Instance Size

The size of all instances is limited to 250GB by default. Note that you only pay for the storage that you use, so you don’t need to reserve this storage in advance. If you require more storage, up to 500GB, then it is possible to increase limits for individual instances for customers with a silver Google Cloud support package.

Google App Engine Limits

Requests from Google App Engine applications to Google Cloud SQL are subject to the following time and connection limits:

- All database requests must finish within the HTTP request timer, around 60 seconds.
- Offline requests like cron tasks have a time limit of 10 minutes.
- Requests from App Engine modules to Google Cloud SQL are subject to the type of module scaling and instance residence time in memory.
- Each App Engine instance cannot have more than 12 concurrent connections to a Google Cloud SQL instance.

Google App Engine applications are also subject to additional Google App Engine quotas and limits as discussed on the Quotas page.

Using Google Cloud Storage

Reading and Writing Files

In App Engine, the local filesystem that your application is deployed to is not writeable. This behavior ensures the security and scalability of your application.

However, if the application needs to write and read files at runtime, App Engine provides a built-in Google Cloud Storage (GCS) stream wrapper that allows you to use many of the standard PHP filesystem functions to read and write files in an App Engine PHP app.

Important: One major difference between writing to a local disk and writing to GCS is that GCS doesn't support modifying or appending to a file after you close it. To get an equivalent behavior in GCS, you must create a new file with the same name, which overwrites the original. Because of this, the *performance*characteristics of reading and writing files to GCS can be quite different from reading and writing to local disk,*especially when frequent modifications need to be made to the same file*.

There are two ways to write files to GCS:

- Write files from your app
  - Simple file write
  - Streamed file write
- Let the user upload files to GCS

Writing files from your app

If you write files from your app, you can write the entire file at once, or you can stream the file write.

Important: For very large files, you may not be able to write directly from your app, because of the requirement for all app requests to complete within 60 seconds. So, for very large files, you may need to upload the filedirectly, not write it from the app.

The GCS stream wrapper is built in to the run time, and is used when you supply a file name starting with `gs://`. The wrapper requires the name of the bucket or file object to be in the form:

```
gs://bucket_name/desired_object_name
```

Simple file write

To write data to Google Cloud Storage from your app, you use `file_put_contents`, using a valid cloud storage URL. For example:

```
<?php

file_put_contents('gs://my_bucket/hello.txt', 'Hello');
```

where `my_bucket` is a properly configured GCS bucket.

Or, if you want to use stream options to supply permissions, caching, and/or metadata options, you could write the file as follows:

```
$options = ['gs' => ['Content-Type' => 'text/plain']];

$ctx = stream_context_create($options);

file_put_contents('gs://my_bucket/hello.txt', 'Hello', 0, $ctx);
```

Streamed file write

Alternatively, you could use `fopen`/`fwrite` to write data in a streaming fashion:

```
<?php

$fp = fopen('gs://my_bucket/some_file.txt', 'w');

fwrite($fp, 'Hello');

fclose($fp);
```

Note that when you use streaming, data will be flushed to GCS in smaller chunks. You do not need to know the total length of the data to be written up front: it will be calculated when the file resource is closed:

Note: You can also use stream options when using streaming file writes to supply permissions, caching, and/or metadata options.

These are the basic ways to write files. For special use cases and more advanced file management, see the topics listed under Where to go next.

Deleting files

If you want to delete the file itself, use the PHP unlink() function.

User uploads

For details about this file write option, see Allowing Users to Upload Files

Reading files

For information about reading files from GCS, see Providing Public Access to Files

Setup and Requirements

If you use the default GCS bucket available for App Engine apps, very little setup is required. See Setup for more details.

Supported PHP filesystem functions

Many of the commonly used PHP file functions are supported, along with file information and directory functions. For a complete list of supported PHP functions, see PHP filesystem functions support.

Extended features provided by Cloud Storage Tools API

The GCS stream wrapper lets you use PHP filesystem calls. However, there are extended features available that you may need for an optimal use of GCS. These extended features are provided in the Cloud Storage Tools API:

This API provides a set of functions that support the serving of files and images, along with other useful utilities. We'll cover several of these in the other topic pages.

Is there any other way to read and write files?

An App Engine PHP app must use the Cloud Storage stream wrapper to write files at runtime. However, if an app needs to *read* files, and these files are static, you can optionally read static files uploaded with your app using PHP filesystem functions such as `file_get_contents`.

For example:

```php
<?php

file_get_contents('config/my_configuration.json');
```

where the path specified must be a path relative to the script accessing them.

You must upload the file or files in an application subdirectory when you deploy your app to App Engine, and must configure the `app.yaml` file so your app can access those files. For complete details, see PHP Application Configuration with `app.yaml`.

In the `app.yaml` configuration, notice that if you use a static file or directory handler (`static_files` or`static_dir`) you must specify `application_readable` set to true or your app won't be able to read the files. However, if the files are served by a `script` handler, this isn't necessary, because these files are readable by script handlers by default.

Where to go next

Read the following topics for details on using the Cloud Storage Tools API:

- Setup, quick setup instructions.
- Providing Public Access to Files, shows how to enable users to download files via browser.
- Allowing Users to Upload Files, shows how to upload files via browser directly, bypassing your app.
- Working with Image Files, shows optimal ways to manage and serve images.
- Advanced File Management, covers the following:
  - Permissions, caching, and metadata stream options.
  - PHP filesystem function support.
  - Using PHP `include` and `require`.
  - Reading and writing custom metadata.
  - Cached file reads.

Setup

Your app requires a properly configured GCS bucket. Fortunately, each app can easily gain access to such a bucket. This bucket, called the *default GCS bucket*, needs no further configuration, permissions, sign-up, or activation. Moreover, the default GCS bucket has a free quota, so you do not need to make your app billable.

Note: If you don't want to use the default GCS bucket, you'll need to configure your own GCS bucket with proper permissions to access the objects. For more information, see Google Cloud Storage Prerequisites. However, note that this requires you to make your app billable and you may incur costs.

For apps created after the App Engine 1.9.0 release, the default GCS bucket is automatically created with your app.

For apps created before the App Engine 1.9.0 release, you can obtain a a default bucket for it by clicking Createwithin the Cloud Integration section in the *Application Settings* page of the App Engine Admin Console.

The default bucket name is typically `<app_id>.appspot.com`, where `<app_id>` is your app ID. You can find the bucket name in the App Engine Admin console Application Settings page, under the label *Google Cloud Storage Bucket*. Alternatively, you can use the CloudStorageTools::getDefaultGoogleStorageBucketName() method to find the name programmatically.

For example, you would write to the default bucket using the normal Cloud Storage stream wrapper:

```
<?php

file_put_contents('gs://<app_id>.appspot.com/hello.txt', 'Hello');
```

or

```
<?php

$fp = fopen('gs://<app_id>.appspot.com/some_file.txt', 'w');

fwrite($fp, 'Hello');

fclose($fp);
```

Providing Public Access to Files

1. Serving files from a script
2. Serving files directly from Google Cloud Storage
3. Serving files uploaded with your app

A common use case is making your files publically accessible via the web. You can do this in App Engine PHP apps in any of these ways:

- Serve files in GCS from a script (your app serves the file).
- Serve files from GCS (GCS serves the file directly).
- Serving files uploaded with your app, (using the static handler in `app.yaml`).

Note that the last methodology does not use GCS.

Serving files from a script

If you want to serve files from your app, you must write a script that serves the file from GCS as follows:

```
<?php

CloudStorageTools::serve('gs://scores/today.txt');
```

Serving files from the app in this way allows the developer to determine user identity and ensure that only authorised users access the file. The downside to this approach is that your application needs to run this code to serve the file, which consumes instance hours and thus incurs cost.

Serving files directly from Google Cloud Storage

There is a faster and more cost-effective way to serve files than serving them from the app, as mentioned above: serving them from GCS directly via HTTP. The files need to be configured to be readable by anonymous users at file write time. (As we'll show in the snippet below, you set the `acl` stream option to `public-read`.)

Once the file is written to GCS as publically readable, you need to get the public URL for the file, using theCloudStorageTools::getPublicUrl() method.

In the following example, we create a public-readable file containing some random numbers, writing it to a GCS bucket, and redirect to that file from GCS.

```
<?php

require_once 'google/appengine/api/cloud_storage/CloudStorageTools.php';

use google\appengine\api\cloud_storage\CloudStorageTools;

$object_url = 'gs://my-bucket/'.time().rand(0,1000).'.txt';

$options = stream_context_create(['gs'=>['acl'=>'public-read']]);

$my_file = fopen($object_url, 'w', false, $options);

for($i=0;$i<900;$i++) {

  fwrite($my_file, 'Number '.$i.' - '.rand(0,1000).'\n');

}

fclose($my_file);

$object_public_url = CloudStorageTools::getPublicUrl($object_url, false);

header('Location:' .$object_public_url);
```

The limitation of this approach is that there is no control over who can access the file, since the file is readable by anyone.

Serving files uploaded with your app

This option is fully described under Is there any other way to read and write files.

Allowing Users to Upload Files

Note: These instructions assume you have a GCS bucket for your app, as described in Setup.

When you upload directly to GCS, you make an HTTP POST to a specific URL, which we'll describe in a moment. App Engine then uses a specific *upload service* to handle the post and write the file to GCS. When the file write is complete, App Engine notifies your app that the upload is complete. Because your app is invoked only upon completion, you can use this method to upload very large files, up to the current maximum of 100 Terabytes.

Important: Direct file uploads to your POST handler, without using the App Engine upload service, are not supported and will fail.

Moreover, user upload of files directly to GCS is faster and more cost-effective than writing to GCS from your App Engine app, because this consumes instance hours and incurs cost.

Note that in the user upload, the file write does not occur within a request to the application. Therefore it is exempt from the 60 second limit that would otherwise apply and allows uploads of very large files.

Implementing file uploads

To implement user file upload:

1.Create the application specific upload URL, using the method CloudStorageTools::createUploadUrl() as follows:

```
require_once 'google/appengine/api/cloud_storage/CloudStorageTools.php';

use google\appengine\api\cloud_storage\CloudStorageTools;

$options = [ 'gs_bucket_name' => 'my_bucket' ];

$upload_url = CloudStorageTools::createUploadUrl('/upload_handler.php',
$options);
```

See `createUploadUrl options` for details about available options. Note that `my_bucket` will be`YOUR_APP_ID.appspot.com` if using the default bucket.

2.Note that you must start uploading to this URL within 10 minutes of its creation. Also, you cannot change the URL in any way - it is signed and the signature is checked before your upload begins.

3.Use this URL as the action for the form you use to accept uploads, for example:

```
<form action="<?php echo $upload_url?>" enctype="multipart/form-data"
method="post">

    Files to upload: <br>

   <input type="file" name="uploaded_files" size="40">

   <input type="submit" value="Send">

</form>
```

After the file(s) upload, a POST is made to the path specified as the first parameter to `createUploadUrl`; in the example above, this is `/upload_handler.php`. The PHP runtime forms the correct `$_FILES` super global, and`tmp_filename` refers to the filename of the newly uploaded file in GCS.

For example, suppose the content of upload_handler.php is the following:

```
<?php

var_dump($_FILES);

?>
```

Uploading a file called `hello.txt` might result in the following output:

```
array(1) {

  ['uploaded_files']=>

  array(5) {

    ['name']=>    string(14) 'hello.txt'

    ['type']=>    string(10) 'text/plain'

    ['tmp_name']=>    string(73)
'gs://my_bucket/L2FwcHMtdXBsb2FkL2Jsb2JzL2IxNUFBVGNJNXNTd0VqR0tFSUtDRGxadGc'

    ['error']=>    int(0)
```

```
    ['size']=>    int(1452)

  }

}
```

After the upload is complete, you can read the uploaded file using the `gs://` stream wrapper. You use`move_uploaded_file` like you normally would for any other uploaded file, for example:

```
$gs_name = $_FILES['uploaded_files']['tmp_name'];

move_uploaded_file($gs_name, 'gs://my_bucket/new_file.txt');
```

You can also then rename the file, for example:

```
// assumes the same $ctx object as shown earlier

if (false == rename('gs://my_bucket/oldname.txt', 'gs://my_bucket/newname.txt',
$ctx)) {

  die('Could not rename.');

}
```

`createUploadUrl` options

Valid `createUploadUrl` options are shown in the following table:

Option	Description
`max_bytes_per_blob`	Integer. Default value: `unlimited`. The value of the largest size allowed for an uploaded blob.
`max_bytes_total`	Integer. Default value: `unlimited`. The total size of all uploaded blobs.
`gs_bucket_name`	String. The name of a Google Cloud Storage bucket that the blobs should be uploaded to. If you don't specify a value, the blob is uploaded to the application's default bucket.

Serving Images

The Cloud Storage Tools API provides convenience methods that you can use to serve image files conveniently:

- `CloudStorageTools.getImageServingUrl()`
- `CloudStorageTools.deleteImageServingUrl()`

One advantage of using this method to serve images rather than simply making the files public using`CloudStorageTools::getPublicUrl`, is that the image method listed above allows dynamic manipulation, such as image resizing, which means you don't have to store different image sizes.

You use getImageServingUrl returns a URL that serves an image. If the image will be displayed within an HTTPS page, set `secure_url` to `True` to avoid mixed-content warnings.

Notice that this URL is publically readable by everyone, but it is not "guessable".

Important: You cannot serve an image from two separate apps; only the first app that calls`getImageServingUrl` can get the URL to serve it because that app has obtained ownership of the image to serve it. Any other app that calls `getImageServingUrl` on the image will therefore be unsuccessful. If a second app needs to serve the image, the app needs to first copy the image and then invoke`getImageServingUrl` on the copy. (The copy is deduped in storage.)

To stop serving the URL, call deleteImageServingUrl.

Note: If you want to delete the file itself, use the PHP `unlink()` function.

The following snippet shows how to use this feature, with the image `myfile.png` made available in a resized and cropped format:

```
require_once 'google/appengine/api/cloud_storage/CloudStorageTools.php';

use google\appengine\api\cloud_storage\CloudStorageTools;

$object_image_file = 'gs://my-bucket/myfile.png';

$object_image_url = CloudStorageTools::getImageServingUrl($object_image_file,

                                          ['size' => 400, 'crop' => true]);

header('Location:' .$object_image_url);
```

Advanced File Management

Permissions, caching and metadata options

You may need to write files using stream options. If so, you can specify the following options when configuring your stream:

Option	Possible Values	Description
`acl`	One of the following values: • `public-read` • `public-read-write` • `authenticated-read` • `bucket-owner-read` • `bucket-owner-full-control`	For descriptions of what these settings do in GCS, see Understanding Default Bucket and Object ACLs. If you do not set `acl`, Cloud Storage sets this parameter as null and uses the default object ACL for that bucket (by default, this is `project-private`).
`Content-Type`	Any valid MIME type	If you do not specify a content type when you upload an object, the Google Cloud Storage (GCS) system defaults to`binary/octet-stream` when it serves the object.
`enable_cache`	true or false (true by default)	Files that are read from GCS are cached into memory (see memcache)for performance improvement. Caching can be turned off using the `enable_cache` directive in the stream context.

Option	Possible Values	Description
`enable_optimistic_cache`	true or false (false by default)	You can enable optimistic caching to read the file object from the cache without checking whether the underlying object was changed in GCS since the last time it was cached. Optimistic caching is ideal for write-once, read-many scenarios.
`metadata`	An associative array, eg.	`['foo' => 'far', 'bar' => 'boo']`See Reading and writing custom metadata.
`read_cache_expiry_seconds`	The number of seconds that an object will remain valid in the cache	You can change the length of time that a cached object remains valid by using the`read_cache_expiry_seconds` directive. This specifies the time after which the object will be re-cached upon the next read attempt. By default this is set to 1 hour (3600).

The following snippet shows how to use stream options:

```
$options = ['gs' => ['Content-Type' => 'text/plain']];

$ctx = stream_context_create($options);

file_put_contents('gs://my_bucket/hello.txt', 'Hello', 0, $ctx);
```

In the snippet above, `$options` is a set of arguments that the stream will use when writing new objects, which can be set as the default options using `stream_context_set_default`.

PHP filesystem functions support on Google Cloud Storage

Many of the native PHP filesystem functions are fully supported, but some are not, and still others have modified support. The following table lists each of these native functions, indicating whether the function is supported or not. If a function is supported but with modifications or limitations, those are described.

Filesystem Function	Supported?	Details
basename — Returns trailing name component of path.	Supported.	
chgrp — Changes file group.	Not supported.	Always returns `false`.
chmod — Changes file mode.	Not supported.	Always returns `false`.
chown — Changes file owner.	Not supported.	Always returns `false`.
clearstatcache — Clears file status cache.	Supported.	
copy — Copies file.	Supported.	
dirname — Returns parent directory's path.	Supported.	Supported but includes the `gs://` prefix.
disk_free_space — Returns available space on filesystem or disk partition.	Not supported.	This is Disabled.
disk_total_space — Returns the total size of a filesystem or disk partition.	Not supported.	This is Disabled.
diskfreespace — Alias of disk_free_space.		
fclose — Closes an open file pointer.	Supported.	
feof — Tests for end-of-file on a file pointer.	Supported.	
fflush — Flushes the output to a file.	Supported.	Has no effect. (Always returns `true`.)
fgetc — Gets character from file pointer.	Supported.	
fgetcsv — Gets line from file pointer and parse for CSV fields.	Supported.	

Filesystem Function	Supported?	Details
fgets — Gets line from file pointer.	Supported.	
fgetss — Gets line from file pointer and strip HTML tags.	Supported.	
file_exists — Checks whether a file or directory exists.	Supported.	
file_get_contents — Reads entire file into a string.	Supported.	
file_put_contents — Write a string to a file.	Supported.	
file — Reads entire file into an array.	Supported.	
fileatime — Gets last access time of file.	Not supported.	Always returns 0.
filectime — Gets inode change time of file.	Not supported.	Always returns 0.
filegroup — Gets file group.	Not supported.	Always returns 0.
fileinode — Gets file inode.	Not supported.	Always returns 0.
filemtime — Gets file modification time.	Supported.	
fileowner — Gets file owner.	Not supported.	Always returns 0.
fileperms — Gets file permissions.	Supported.	The execute bit is always off.
filesize — Gets file size.	Supported.	
filetype — Gets file type.	Supported.	
flock — Portable advisory file locking.	Not Supported.	Always returns `false`.
fopen — Opens file or URL.	Supported.	Only supports these modes: `r`, `rb`, `rt`, `w`,`wb`, `wt`.
fpassthru — Output all remaining data on a file pointer.	Supported.	

Filesystem Function	Supported?	Details
fputcsv — Format line as CSV and write to file pointer.	Supported.	
fputs — Alias of fwrite.		
fread — Binary-safe file read.	Supported.	
fscanf — Parses input from a file according to a format.	Supported.	
fseek — Seeks on a file pointer.	Supported.	Only supports files opened using read mode.
fstat — Gets information about a file using an open file pointer.	Supported.	
ftell — Returns the current position of the file read/write pointer.	Supported.	
ftruncate — Truncates a file to a given length.	Not supported.	Always returns `false`.
fwrite — Binary-safe file write.	Supported.	
glob — Find pathnames matching a pattern.	Not supported.	Always returns `false`.
is_dir — Tells whether the filename is a directory.	Supported.	
is_executable — Tells whether the filename is executable.	Not supported.	Always returns `false`.
is_file — Tells whether the filename is a regular file.	Supported.	
is_link — Tells whether the filename is a symbolic link.	Not supported.	Always returns `false`.
is_readable — Tells whether a file exists and is readable.	Supported.	
is_uploaded_file — Tells whether the file was uploaded via HTTP POST.	Supported.	
is_writable — Tells whether the filename is writable.	Supported.	Value is cached and may not reflect permission changes immediately.
is_writeable — Alias of is_writable.		
lchgrp — Changes group ownership of symlink.	Not supported.	This is Disabled.

Filesystem Function	Supported?	Details
lchown — Changes user ownership of symlink.	Not supported.	This is Disabled.
link — Create a hard link.	Not supported.	This is Disabled.
linkinfo — Gets information about a link.	Not supported.	Always returns `-1`.
lstat — Gives information about a file or symbolic link.	Supported.	
mkdir — Makes directory.	Supported.	
move_uploaded_file — Moves an uploaded file to a new location.	Supported.	
parse_ini_file — Parse a configuration file.	Supported.	
pathinfo — Returns information about a file path.	Supported.	
pclose — Closes process file pointer.	Not supported.	This is Disabled.
popen — Opens process file pointer.	Not supported.	This is Disabled.
readfile — Outputs a file.	Supported.	
readlink — Returns the target of a symbolic link.	Not supported.	Always returns `false`.
realpath — Returns canonicalized absolute pathname.	Not supported.	Always returns `false`.
rename — Renames a file or directory.	Supported.	
rewind — Rewind the position of a file pointer.	Supported.	Supported only for read mode.
rmdir — Removes directory.	Supported.	
set_file_buffer — Alias of stream_set_write_buffer.		
stat — Gives information about a file.	Supported.	
symlink — Creates a symbolic link.	Not	This is Disabled.

Filesystem Function	Supported?	Details
	supported.	
tempnam — Create file with unique file name.	Not supported.	This is Disabled.
tmpfile — Creates a temporary file.	Supported.	Returns a memory-backed file similar to `php://memory`.
touch — Sets access and modification time of file.	Not supported.	Always returns `false`.
umask — Changes the current umask.	Supported.	Does not apply to Cloud Storage files.
unlink — Deletes a file.	Supported.	

In addition, the following PHP directory-reading functions are supported:

Function Name
`opendir`
`readdir`
`rewinddir`
`closedir`

Note: As GCS listing operations are eventually consistent, `readdir` may not reflect the change immediately after an object is added, renamed or removed.

Using PHP include and require

To help maintain the security of your application, the ability to include or require from a GCS file is disabled by default, but you can enable it, as follows:

To include or require PHP code from Google Cloud Storage in your application, you must specify which buckets contain those files using the google_app_engine.allow_include_gs_buckets directive in yourphp.ini file.

Reading and writing custom metadata

To attach custom metadata to a file being written to Google Cloud Storage, add the metadata to $optionsbefore creating the stream context used for the file_put_contents call.

In this example, metadata named foo is given the value far, and another named bar is given the value boo. The example also sets the content type to text/plain. The stream context is created using this information in$options as shown, and the file is written to GCS using file_put_contents(), along with the metadata and content type:

```
// set metadata

$options = [

  'gs' => [

        'Content-Type' => 'text/plain',

        'metadata' => ['foo' => 'far', 'bar' => 'boo'],

  ],

];

$ctx = stream_context_create($options)

file_put_contents('gs://foo/bar', $ctx);
```

To read the custom metadata and content type for a file, call fopen on the file and then useCloudStorageTools::getContentType to get the content type and CloudStorageTools::getMetaDatato get the metadata.

The custom metadata and content type are available once you have the file pointer returned from the fopen call.

```php
// read content type and metadata

$fp = fopen('gs://foo/bar', 'r');

$content_type = CloudStorageTools::getContentType($fp);

var_dump($content_type);

$metadata = CloudStorageTools::getMetaData($fp);

var_dump($metadata);

/* expected output:

text/plain

array(2) {

  ['foo']=>

  string(3) 'far'

  ['bar']=>

  string(3) 'boo'

}

*/
```

Cached file reads

By default, the GCS stream wrapper caches file reads into memcache to improve performance on subsequent reads. Caching can be turned off using the `enable_cache` directive in the stream context.

Note: When caching is enabled, files read from GCS are cached as they are read, and the cache also contains the file information regarding writability. So, be aware that `is_writable` could be querying stale file information. Also, if you change the ACL on a GCS bucket to to either grant or deny write permissions to the application, this change might take some time to be reflected.

For further performance improvement, you can also enable optimistic caching, which will read the file object from the cache without seeing whether the underlying object was changed in Google Cloud Storage since the last time it was cached, using the `enable_optimistic_cache` directive (set to `false` by default). Optimistic caching is ideal for write-once, read-many scenarios.

You can also change the length of time that a cached object remains valid by using the`read_cache_expiry_seconds` directive, after which time the object will be re-cached upon the next read attempt. By default this is set to 1 hour (`3600`).

In this example, optimistic caching is turned on, and file objects are configured to be marked for re-caching from Google Cloud Storage after 5 minutes.

```
$ctx = [
  'gs' => [
    'enable_cache' => true,
    'enable_optimistic_cache' => true,
    'read_cache_expiry_seconds' => 300,
  ]
];
// Use $ctx for the stream context and do read operation...
```