Start Lab 01:00:00

JAVAMS07 Uploading and Storing Files

1 hour Free ★★★★☆ Rate Lab

Overview

In this series of labs, you take a demo microservices Java application built with the Spring framework and modify it to use an external database server. You adopt some of the best practices for tracing, configuration management, and integration with other services using integration patterns.

Google Cloud has a bucket-based object storage solution called Cloud Storage. Cloud Storage is designed to store a large number of files and a large volume of binary data, so that you don't need to manage your own file systems or file sharing services. Cloud Storage can be used directly by many other Google Cloud products. For example, you can store data files on Cloud Storage and process those data filesin a managed Hadoop (Cloud Dataproc) cluster. You can also import structured data stored on Cloud Storage directly into BigQuery for ad hoc data analytics using standard SQL.

Cloud Storage provides fast, low-cost, highly durable, global object storage for developers and enterprises that need to manage unstructured file data. In Cloud Storage, the consistent API, low latency, and speed across storage classes simplify development integration and reduce code complexity.

In this lab, you add the ability to upload an image associated with a message. You store the image in Cloud Storage.

Objectives

In this lab, you learn how to perform the following tasks:

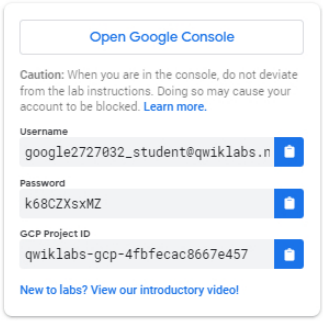
- Add the Spring starter for Cloud Storage
- Create a Cloud Storage bucket
- Modify the HTML template of the frontend application to enable file uploads
- Modify the frontend application to process and store images on Cloud Storage
- Modify the frontend application to display uploaded message images

Task 0. Lab Preparation

Access Qwiklabs

How to start your lab and sign in to the Console

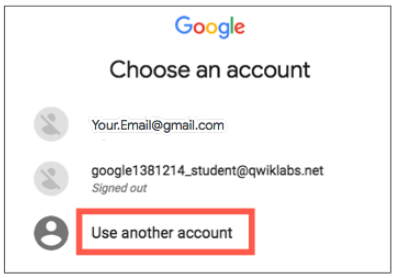
1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.



2. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Choose an account** page.

   Tip: Open the tabs in separate windows, side-by-side.

3. On the Choose an account page, click **Use Another Account**.



4. The Sign in page opens. Paste the username that you copied from the Connection Details panel. Then copy and paste the password.

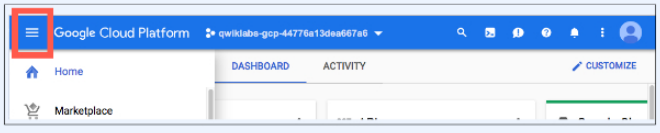
   Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own GCP account, do not use it for this lab (avoids incurring charges).

5. Click through the subsequent pages:

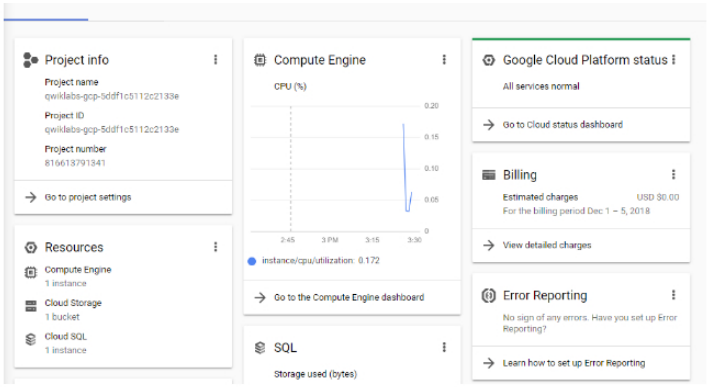
   - Accept the terms and conditions.
   - Do not add recovery options or two-factor authentication (because this is a temporary account).
   - Do not sign up for free trials.

   After a few moments, the GCP console opens in this tab.

Note: You can view the menu with a list of GCP Products and Services by clicking the **Navigation menu** at the top-left, next to "Google Cloud Platform".



After you complete the initial sign-in steps, the project dashboard appears.



Fetch the application source files

To begin the lab, click the **Activate Cloud Shell** button at the top of the Google Cloud Console. To activate the code editor, click the `Open Editor` button on the toolbar of the Cloud Shell window. This sets up the editor in a new tab with continued access to Cloud Shell.

> **Note:** a Cloud Storage bucket that is named using the project ID for this lab is automatically created for you by the lab setup. The source code for your applications is copied into this bucket when the Cloud SQL server is ready. You might have to wait a few minutes for this action to complete.

1. In the Cloud Shell command line, enter the following command to create an environment variable that contains the project ID for this lab:

```
export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
```

2. Verify that the demo application files were created.

```
gsutil ls gs://$PROJECT_ID
```

3. Copy the application folders to Cloud Shell.

```
gsutil -m cp -r gs://$PROJECT_ID/* ~/
```

4. Make the Maven wrapper scripts executable. Now you're ready to go!

```
chmod +x ~/guestbook-frontend/mvnw
chmod +x ~/guestbook-service/mvnw
```

Task 1. Add the Cloud Storage starter

In this task, you add the Cloud Storage starter to the guestbook frontend application.

1. In the Cloud Shell code editor, open `~/guestbook-frontend/pom.xml`.

2. Insert the following new dependency at the end of the `<dependencies>` section, just before the closing `</dependencies>` tag:

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-storage</artifactId>
        </dependency>
```

Task 2. Store the uploaded file

In this task, you update the home page to enable files to be uploaded with a message.

Modify the main.container class to allow file uploads

You edit the `main.container` class input form action to handle multi-part data encoding.

1. In the Cloud Shell code editor, open `~/guestbook-frontend/src/main/resources/templates/index.html`.
2. Change this line:

```
<form action="/post" method="post">
```

To these lines:

```
<!-- Set form encoding type to multipart form data -->
<form action="/post" method="post" enctype="multipart/form-data">
```

3. Insert the following tags before the `<input type="submit" value="Post"/>` tag:

```
<!-- Add a file input -->
<span>File:</span>
<input type="file" name="file" accept=".jpg, image/jpeg"/>
```

Update FrontendController to accept the uploaded file

You update the FrontendController.java file to accept file data if the user chooses to upload an image with a message.

1. In the Cloud Shell code editor, open `~/guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java`.
2. Insert the following `import` directives immediately below the existing `import` directives:

```
import org.springframework.cloud.gcp.core.GcpProjectIdProvider;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.context.ApplicationContext;
import org.springframework.core.io.Resource;
import org.springframework.core.io.WritableResource;
import org.springframework.util.StreamUtils;
import java.io.*;
```

3. Insert the following code near the beginning of the `FrontendController` class definition, immediately before `@GetMapping("/")`.

You need to get `Applicationcontext` in order to create a new resource. The project ID is required in order to access Cloud Storage because this demo application uses the project ID as the Cloud Storage bucket name.

```
    // The ApplicationContext is needed to create a new Resource.
```

`frontend/src/main/java/com/example/frontend/FrontendController.java`.

2. Insert the following `import` directives immediately below the existing `import` directives:

```
import org.springframework.cloud.gcp.core.GcpProjectIdProvider;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.context.ApplicationContext;
import org.springframework.core.io.Resource;
import org.springframework.core.io.WritableResource;
import org.springframework.util.StreamUtils;
import java.io.*;
```

3. Insert the following code near the beginning of the `FrontendController` class definition, immediately before `@GetMapping("/")`.

   You need to get `Applicationcontext` in order to create a new resource. The project ID is required in order to access Cloud Storage because this demo application uses the project ID as the Cloud Storage bucket name.

```
    // The ApplicationContext is needed to create a new Resource.
34      private String greeting;
35
36      @Autowired
37      private OutboundGateway outboundGateway;
38
39      // The ApplicationContext is needed to create a new Resource.
40      @Autowired
41      private ApplicationContext context;
42      // Get the Project ID, as its Cloud Storage bucket name here
43      @Autowired
44      private GcpProjectIdProvider projectIdProvider;
45
46      @GetMapping("/")
47      public String index(Model model) {
48          if (model.containsAttribute("name")) {
49              String name = (String) model.asMap().get("name");
50              model.addAttribute("greeting", String.format("%s %s", greeting, name));
51          }
52          model.addAttribute("messages", client.getMessages().getContent());
53          return "index";
54      }
55
56      @PostMapping("/post")
```

> **Note**
>
> The uploaded files are stored in a Cloud Storage bucket that uses the lab project ID as its name. This bucket has been created automatically during the lab setup so you did not have to create it yourself but if you are replicating this lab in your own environment you will have to create the storage bucket for the application.

Change the definition for the post method

You modify the post method to save uploaded images to Cloud Storage.

1. Near the end of the file, change this line:

```
public String post(@RequestParam String name, @RequestParam String
message, Model model) {
```

   To these lines:

```
    public String post(
       @RequestParam(name="file", required=false) MultipartFile file,
         @RequestParam String name,
         @RequestParam String message, Model model)
       throws IOException {
```

2. Insert the following code immediately after the line `model.addAttribute("name", name);`:

```
        String filename = null;
        if (file != null && !file.isEmpty()
            && file.getContentType().equals("image/jpeg")) {
                // Bucket ID is our Project ID
                String bucket = "gs://" +
                      projectIdProvider.getProjectId();
                // Generate a random file name
                filename = UUID.randomUUID().toString() + ".jpg";
                WritableResource resource = (WritableResource)
                      context.getResource(bucket + "/" + filename);
                // Write the file to Cloud Storage
                try (OutputStream os = resource.getOutputStream()) {
```

```
                    os.write(file.getBytes());
                }
        }
```

3. Add the following code to insert the location of the uploaded file immediately before the `client.add(payload);` line:

```
```

2. Insert the following code immediately after the line `model.addAttribute("name", name);`:

```
        String filename = null;
        if (file != null && !file.isEmpty()
            && file.getContentType().equals("image/jpeg")) {
                // Bucket ID is our Project ID
                String bucket = "gs://" +
                      projectIdProvider.getProjectId();
                // Generate a random file name
                filename = UUID.randomUUID().toString() + ".jpg";
                WritableResource resource = (WritableResource)
                      context.getResource(bucket + "/" + filename);
                // Write the file to Cloud Storage
                try (OutputStream os = resource.getOutputStream()) {
                     os.write(file.getBytes());
                 }
        }
```

3. Add the following code to insert the location of the uploaded file immediately before the `client.add(payload);` line:

```
            log.info("Saving message");

            // Post the message to the backend service
            GuestbookMessage payload = new GuestbookMessage();
            payload.setName(name);
            payload.setMessage(message);

            // Store the generated file name in the database
            payload.setImageUri(filename);
            client.add(payload);

            // Add a log message at the end.
            log.info("Saved message");


            // At the very end, publish the message
            outboundGateway.publishMessage(name + ": " + message);
        }
        return "redirect:/";
    }
}
```

Task 3. Test the application in the Cloud Shell

In this task, you run the application in the Cloud Shell to test the new image upload functionality.

1. In the Cloud Shell change to the `guestbook-service` directory.

```
cd ~/guestbook-service
```

2. Run the backend service application.

```
./mvnw spring-boot:run -Dspring-boot.run.jvmArguments="-
Dspring.profiles.active=cloud"
```

The backend service application launches on port 8081.This takes a minute or two to complete and you should wait until you see that the GuestbookApplication is running.

```
Started GuestbookApplication in 20.399 seconds (JVM running...)
```

3. Open a new Cloud Shell session tab to run the frontend application by clicking the plus (+) icon to the right of the title tab for the initial Cloud Shell session.

4. Change to the `guestbook-frontend` directory.

```
cd ~/guestbook-frontend
```

5. Start the frontend application with the `cloud` profile.

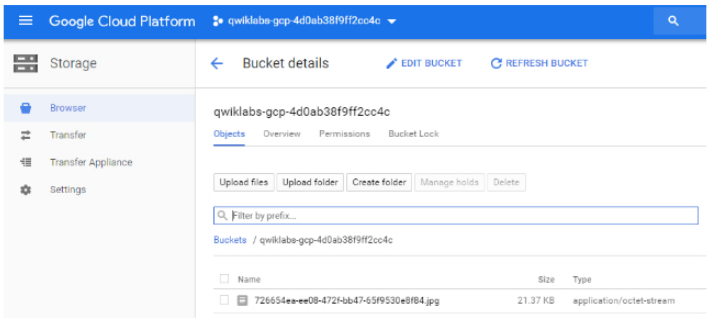
```
./mvnw spring-boot:run -Dspring.profiles.active=cloud
```

6. Open the Cloud Shell web preview on port 8080 and post a message with a small JPEG image.

7. From the GCP console, navigate to **Storage > Browser**.

8. Navigate to your bucket.

9. The uploaded file should appear as in the screenshot. Note that there will also be two folders that were created during the lab startup that contain the demo application source files.



Task 4. Serve the image from Cloud Storage

In this task, you update the frontend application to retrieve and display images associated with guestbook messages.

Add a method to FrontendController to retrieve the requested image and send it to the browser

You edit `FrontendController.java` so that it fetches image files associated with messages if they are found.

1. In the Cloud Shell code editor, open `~/guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java`.

2. Add the following `import` directive immediately below the existing `import` directives:

```
import org.springframework.http.*;
```

3. Insert the following code at the end of the `FrontEndController` class definition, after the closing brace for the `post` method definition and immediately before the final closing brace:

```
// ".+" is necessary to capture URI with filename extension
@GetMapping("/image/{filename:.+}")
public ResponseEntity<Resource> file(
```

```
    @PathVariable String filename) {
        String bucket = "gs://" +
                        projectIdProvider.getProjectId();
        // Use "gs://" URI to construct
        // a Spring Resource object
        Resource image = context.getResource(bucket +
                         "/" + filename);
        // Send it back to the client
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.IMAGE_JPEG);
        return new ResponseEntity<>(
        image, headers, HttpStatus.OK);
  }
```

The end of the `FrontendController.java` file should look like the screenshot:

```
104   // ".+" is necessary to capture URI with filename extension
105   @GetMapping("/image/{filename:.+}")
106   public ResponseEntity<Resource> file(
107       @PathVariable String filename) {
108           String bucket = "gs://" +
109                           projectIdProvider.getProjectId();
110           // Use "gs://" URI to construct
111           // a Spring Resource object
112           Resource image = context.getResource(bucket +
113                            "/" + filename);
114           // Send it back to the client
115           HttpHeaders headers = new HttpHeaders();
116           headers.setContentType(MediaType.IMAGE_JPEG);
117           return new ResponseEntity<>(
118           image, headers, HttpStatus.OK);
119     }
120 }
121
122
```

Update the home page so that it loads the image if present

You edit the `main.container` class to load and display images for messages if they are present.

1. In the Cloud Shell code editor, open `~/guestbook-frontend/src/main/resources/templates/index.html`.

   In the next step, you insert an `<image>` tag in the `messages` class.

2. Add the following `<img>` tag after the second `<span>` tag:

```
            <img th:src="'/image/' + ${message.imageUri}"
                 alt="image" height="40px"
                 th:unless="${#strings.isEmpty(message.imageUri)}"/>
```

The `main container` div class should look like the screenshot:

```
27  <div class="main container">
28      <div class="input">
29          <!-- Set form encoding type to multipart form data -->
30          <form action="/post" method="post" enctype="multipart/form-data">
31              <span>Your name:</span><input type="text" name="name" th:value="${name}"/>
32              <span>Message:</span><input type="text" name="message"/>
33              <!-- Add a file input -->
34              <span>File:</span>
35              <input type="file" name="file" accept=".jpg, image/jpeg"/>
36              <input type="submit" value="Post"/>
37          </form>
38      </div>
39
40      <div th:if="${greeting != null}" class="greeting">
41          <span th:text="${greeting}">Greeting</span>
42      </div>
43
44      <div class="messages">
45          <div th:each="message: ${messages}" class="message">
46              <span th:text="${message.name}" class="username">Username</span>
47              <span th:text="${message.message}" class="message">Message</span>
48              <img th:src="'/image/' + ${message.imageUri}"
49                  alt="image" height="40px"
50                  th:unless="${#strings.isEmpty(message.imageUri)}"/>
51          </div>
52
53      </div>
54
55  </div>
56
```

Task 5. Restart the frontend application to test image retrieval

In this task, you restart the frontend application to activate the new image retrieval and display functionality.

1. Switch to the interactive Cloud Shell tab that is running the frontend application. You should see the following status message on screen.

```
Started FrontendApplication in 24.349 seconds (JVM running for 78.449)
```

2. Press CTRL+C to terminate the running application.

3. Restart the frontend application.

```
cd ~/guestbook-frontend
./mvnw spring-boot:run -Dspring.profiles.active=cloud
```

4. Switch to the web preview browser tab, and refresh the view to verify that messages with uploaded images now display image thumbnails properly.

End your lab

When you have completed your lab, click **End Lab**. Qwiklabs removes the resources you've used and cleans the account for you.

You'll be given an opportunity to rate the lab experience. Select the applicable number of stars, type a comment, and then click **Submit**.

The number of stars indicates your rating:

- 1 star = Very dissatisfied
- 2 stars = Dissatisfied
- 3 stars = Neutral
- 4 stars = Satisfied
- 5 stars = Very satisfied

You can close the dialog box if you don't want to provide feedback.

For feedback, suggestions, or corrections, use the **Support** tab.