



Developer Guide: Storing Files to AWS S3 from Jakarta EE



Contents

Guide Updated: **October 2024**

Introduction	1
Objectives of this Guide	1
Common File Storage Challenges	1
Scalability: Handling Growing Storage Needs	1
Availability: Ensuring Files are Accessible When Needed	2
Durability: Protecting against Data Loss	2
Performance: Fast Read and Write Operations	2
Cost-effectiveness: Managing Storage Expenses	2
Security: Protecting Sensitive Data	2
Why Amazon S3?	3
Considerations for Adopting S3	3
The Implementation	4
Prerequisites	4
Step 1: Set Up AWS Credentials	4
Step 2: Add AWS SDK Dependencies	4
Step 3: Configure AWS S3 Client	5
Step 4: Implement File S3 Service	6
Step 5: Use the Upload Service in a JAX-RS Resource	7
Your Application In Payara Cloud	12
Best Practices for Efficient and Secure S3 File Management	13
Secure Credential Management	13
Error Handling and Logging	13
Large File Uploads	13
Caching	13
Content-Type Setting	14
Versioning	14
Access Control	14
Encryption	14
Monitoring and Alerts	14
Cost Optimization	14
Conclusions	15

Introduction

Modern Jakarta EE applications running on Payara Cloud often need to handle file storage for various purposes, such as user uploads, document management or data backups. As applications scale, managing file storage can become challenging, especially when dealing with large volumes of data or high concurrent access. Payara Cloud, the Payara as a Service (PaaS) platform for deploying Jakarta EE and MicroProfile applications, gives application full access to all cloud services they need. This includes uploading files to external storage services. This brief developer guide will show you how to connect to and upload files to Amazon Web Services (AWS) Simple Storage Service from your Jakarta EE application running on Payara Cloud.

Objectives of this Guide

After reading this guide, you will:

- Understand common file storage challenges in enterprise applications
- Learn why Amazon S3 is a popular solution for these challenges
- Get insight into key considerations when adopting S3 for your Jakarta EE application
- Learn how to implement S3 integration in a Jakarta EE application, including:
 - Setting up AWS credentials
 - Configuring the S3 client
 - Implementing file upload functionality
 - Creating a REST endpoint for file uploads
- Learn best practices for using S3 with Jakarta EE applications running on Payara Cloud

Common File Storage Challenges

File storage systems are important to business applications of all sizes, but they also come with their own set of challenges. From managing the ever-growing volume of data to ensuring its security and accessibility, file storage can be complex to implement and maintain. Let's explore some of the most common application file storage challenges out there.

Scalability: Handling Growing Storage Needs

As businesses generate more data, their storage requirements increase exponentially. File storage systems need to be able to scale to accommodate this growth without compromising performance or requiring extensive refactoring. This often involves being able to add more storage capacity on-the-fly, or to distribute data across multiple storage nodes.

Availability: Ensuring Files are Accessible When Needed

Data availability is critical for business continuity. File storage systems need to ensure that files are accessible when needed, even in the face of hardware failures, network outages or other disruptions. This often involves implementing redundancy mechanisms, such as data replication or distributed storage, to ensure that multiple copies of data are available in different locations.

Durability: Protecting against Data Loss

Data loss can be catastrophic for businesses. File storage systems need to protect against data loss due to hardware failures, software errors, or other unforeseen events. This typically involves implementing data backup and recovery mechanisms, as well as data integrity checks, to ensure that data remains intact and recoverable.

Performance: Fast Read and Write Operations

File storage systems need to be able to handle high volumes of read and write operations efficiently to support business applications. This often involves using fast storage media, such as solid-state drives (SSDs), and optimizing data access patterns to minimize latency.

Cost-effectiveness: Managing Storage Expenses

Storage costs can be a significant expense for businesses. File storage systems need to be cost-effective to manage storage expenses while still meeting performance and availability requirements. This often involves using a combination of storage tiers to balance cost and performance, such as hot storage for frequently accessed data and cold storage for archival data.

Security: Protecting Sensitive Data

File storage systems need to protect sensitive data from unauthorized access, theft or modification. This typically involves implementing access control mechanisms, encryption and other security measures to ensure that data remains confidential and secure.

Why Amazon S3?

Amazon Simple Storage Service (S3) is a popular choice for addressing these challenges in Jakarta EE applications. Built from the ground up with modern application needs in mind, Amazon S3 addresses these file storage challenges through its features as follows.

- **Scalability:** S3 provides virtually unlimited storage capacity.
- **Availability:** With 99.99% availability, S3 ensures your files are almost always accessible.
- **Durability:** S3 offers 99.999999999% (11 9's) durability, minimizing the risk of data loss.
- **Performance:** S3 provides low-latency access to data, with options for different storage classes based on access patterns.
- **Cost-effectiveness:** Pay only for what you use, with tiered pricing for larger storage volumes.
- **Security:** S3 offers hierarchical security features, including encryption at rest and in transit, access controls and audit logging.

Considerations for Adopting S3

Before adopting S3 storage into your applications however, there are a few considerations worth keeping in mind, including

- **Data Classification:** Identify which data is suitable for cloud storage and which must remain on-premises due to regulatory or security requirements.
- **Access Patterns:** Understand how your application will access the stored files. This affects your choice of S3 storage class and potential use of caching mechanisms.
- **Compliance:** Absolutely make sure S3 usage complies with relevant regulations (e.g., GDPR, HIPAA) for your industry and region.
- **Costs:** Analyze the cost implications, including storage, data transfer and API request pricing for expected usage by your application. Be sure to factor in projected usage growth for the foreseeable future.
- **Latency:** Consider the network latency between your application servers and AWS regions.
- **Backup and Versioning:** Determine your needs for file versioning and backup strategies.
- **Integration Complexity:** Assess the effort required to integrate S3 into your existing application architecture. Though it should be fairly straightforward if you are using the standard Jakarta EE/Java platforms.
- **Vendor Lock-in:** Consider the implications of tying your application to AWS services and potential migration strategies should the need arise.

The Implementation

Now, let's dive into the technical implementation of integrating S3 with your Jakarta EE application.

Prerequisites

- Jakarta EE 10+ application
- Payara Cloud account ([get a free trial here](#))
- AWS account with S3 access
- AWS SDK for Java
- Basic familiarity with AWS S3 terminology. This guide assumes working familiarity with them. If not, please give the S3 getting started guide a read.

Step 1: Set Up AWS Credentials

- Create an IAM user with S3 access in your AWS account
- Note the Access Key ID and Secret Access Key

Step 2: Add AWS SDK Dependencies

Add the following dependencies to your pom.xml:

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>s3</artifactId>
  <version>2.28.1</version>
</dependency>
```

Step 3: Configure AWS S3 Client

Create a CDI producer for the S3Client, scoped to the lifetime of the running application instance:

```
@ApplicationScoped
public class S3ClientProducer {

    @Inject
    @ConfigProperty(name = "aws.secretAccessKey")
    String accessKey;

    @Inject
    @ConfigProperty(name = "aws.accessKeyId")
    String secretKey;

    @Produces
    @ApplicationScoped
    public S3Client createS3Client() {
        return S3Client.builder()
            .region(Region.US_EAST_1)
            .credentialsProvider(StaticCredentialsProvider.create(
                AwsBasicCredentials.create(accessKey, secretKey)))
            .build();
    }
}
```

We use MicroProfile Config to externalize the security credentials. This helps your application to avoid hard coding the credentials. This way, you can have different credentials for different stages of development environments without needing to change your application code. Also, using the Config API for externalisation allows you to configure your application in the Payara Cloud application dashboard. Using the Jakarta CDI producer construct gives you the ability to inject the S3Client to different components that need it without need to initialize in all those places. Think of the producer mechanism as “produce once use everywhere.”

Step 4: Implement File S3 Service

Now create a service or CDI component to handle file uploads:

```
@ApplicationScoped
public class S3Service {

    @Inject
    private S3Client s3Client;

    public void uploadFile(String bucketName, String key, InputStream
inputStream, long contentLength) {
        PutObjectRequest putObjectRequest = PutObjectRequest.builder()
            .bucket(bucketName)
            .key(key)
            .build();

        s3Client.putObject(putObjectRequest, RequestBody.
fromInputStream(inputStream, contentLength));
    }

    public byte[] downloadFile(String bucketName, String key) {
        GetObjectRequest getObjectRequest = GetObjectRequest.builder()
            .bucket(bucketName)
            .key(key)
            .build();

        ResponseBytes<GetObjectResponse> objectBytes = s3Client.
getObjectAsBytes(getObjectRequest);
        return objectBytes.asByteArray();
    }

    public Map<String, String> getFileMetadata(String bucketName, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
            .bucket(bucketName)
            .key(key)
            .build();
    }
}
```



```
        HeadObjectResponse headObjectResponse = s3Client.  
headObject(headObjectRequest);  
  
        return headObjectResponse.metadata();  
    }  
    public List<S3Object> listFiles(String bucketName) {  
        ListObjectsV2Request listObjectsRequest = ListObjectsV2Request.  
builder()  
            .bucket(bucketName)  
            .build();  
  
        ListObjectsV2Response listObjectsResponse = s3Client.  
listObjectsV2(listObjectsRequest);  
        return listObjectsResponse.contents();  
    }  
}
```

We inject the `S3Client` into the `S3UploadService`. The Jakarta CDI runtime knows it should invoke the producer method we created above for an instance to inject into this field. Of course, the instance is technically a contextual instance, complete with a well-defined scope and context. As the producer method sets the scope of the returned instance to `ApplicationScoped`, there will be only a single instance of the `S3Client` injected into all fields that need it. This single instance will be fully managed by the CDI runtime.

Uploading a file to a given S3 bucket entails calling the `putObject` method on the `S3Client`, passing it a `PutRequestObject` and `RequestBody` instances. The method returns a `PutObjectResponse` that you can use to send metadata about the upload to clients of the `S3UploadService`. We do not use that in this example. The `S3Service` also has methods to download and list files as well as get file metadata. All these methods use similar patterns of `[action]ObjectRequest`. For example, getting a file uses the `GetObjectRequest` and listing files uses the `ListObjectsRequest`.

Step 5: Use the Upload Service in a JAX-RS Resource

Create a REST endpoint to handle file uploads, assuming your application uses Jakarta REST:

```
@Path("/upload")  
public class FileUploadResource {  
  
    @Inject  
    private S3UploadService s3UploadService;
```

```
@POST
@Consumes(MediaType.MULTIPART_FORM_DATA)
public Response uploadFile(MultipartFormDataInput input) {
    try {
        Map<String, List<InputPart>> uploadForm = input.getFormDataMap();
        List<InputPart> inputParts = uploadForm.get("file");

        for (InputPart inputPart : inputParts) {
            MultivaluedMap<String, String> header = inputPart.getHeaders();
            String fileName = getFileName(header);

            InputStream inputStream = inputPart.getBody(InputStream.class,
null);

            long contentLength = inputStream.available();

            s3UploadService.uploadFile("your-bucket-name", fileName,
inputStream, contentLength);
        }

        return Response.status(Response.Status.OK).entity("File uploaded
successfully").build();
    } catch (Exception e) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR).
entity("Failed to upload file").build();
    }
}

private String getFileName(MultivaluedMap<String, String> header) {
    String[] contentDisposition = header.getFirst("Content-Disposition").
split(";");
    for (String filename : contentDisposition) {
        if ((filename.trim().startsWith("filename"))) {
            String[] name = filename.split("=");
            String finalFileName = name[1].trim().replaceAll("\\\"", "");
            return finalFileName;
        }
    }
    return "unknown";
}
```

```
@GET
@Path("/download/{fileName}")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response downloadFile(@PathParam("fileName") String fileName) {
    try {
        byte[] fileContent = s3Service.downloadFile("your-bucket-name",
fileName);
        return Response.ok(fileContent)
            .header("Content-Disposition", "attachment; filename=\"" +
fileName + "\"")
            .build();
    } catch (Exception e) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
            .entity("Failed to download file: " + e.getMessage())
            .build();
    }
}

@GET
@Path("/metadata/{fileName}")
@Produces(MediaType.APPLICATION_JSON)
public Response getFileMetadata(@PathParam("fileName") String fileName) {
    try {
        Map<String, String> metadata = s3Service.getFileMetadata("your-
bucket-name", fileName);
        return Response.ok(metadata).build();
    } catch (Exception e) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
            .entity("Failed to retrieve metadata: " + e.getMessage())
            .build();
    }
}

@GET
@Path("/list")
@Produces(MediaType.APPLICATION_JSON)
public Response listFiles() {
    try {
        List<S3Object> files = s3Service.listFiles("your-bucket-name");
        return Response.ok(files).build();
    } catch (Exception e) {
```

```
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
            .entity("Failed to list files: " + e.getMessage())
            .build();
    }
}
```

The Jakarta REST endpoint exposes rest resources to match the methods of the `S3UploadService`. Of course, if your application does not use Jakarta REST, you can also directly inject the `S3UploadService` into any component that needs to interact with files. For example, your application may have a component or service that generates documents as part of a specific workflow like policy documents for issuing an insurance policy. In this case, the policy service will declare a dependency on `S3UploadService` and directly upload the generated policy documents to the policies bucket as shown below.

```
@ApplicationScoped
public class PolicyService {

    @Inject
    private S3Service s3Service;

    private static final String POLICY_BUCKET = "insurance-policies";

    public String generateAndUploadPolicy(String customerName, String
policyType) {

        String policyNumber = generatePolicyNumber();
        String policyDocument = generatePolicyDocument(customerName,
policyType, policyNumber);
        InputStream inputStream = new ByteArrayInputStream(policyDocument.
getBytes(StandardCharsets.UTF_8));
        String key = "policies/" + policyNumber + ".txt";
        s3Service.uploadFile(POLICY_BUCKET, key, inputStream, policyDocument.
length());

        return policyNumber;
    }

    private String generatePolicyNumber() {
```

```
        return "POL-" + UUID.randomUUID().toString().substring(0,
8).toUpperCase();
    }

    private String generatePolicyDocument(String customerName, String
policyType, String policyNumber) {
        // This is a simplified version. In a real-world scenario, this would
be much more complex
        // and might involve templates, calculations, etc.

        return String.format("""

            Insurance Policy
            -----

            Policy Number: %s
            Customer Name: %s
            Policy Type: %s

            This document certifies that the above-named customer
            is insured under the specified policy type.

            Please refer to the full terms and conditions for details.

            """, policyNumber, customerName, policyType);
    }

    public byte[] retrievePolicy(String policyNumber) {
        String key = "policies/" + policyNumber + ".txt";
        return s3Service.downloadFile(POLICY_BUCKET, key);
    }

    public void archivePolicy(String policyNumber) {
        String sourceKey = "policies/" + policyNumber + ".txt";
        String destinationKey = "archived-policies/" + policyNumber + ".txt";

        // First, download the policy
        byte[] policyContent = s3Service.downloadFile(POLICY_BUCKET,
sourceKey);
```

```
// Then, upload it to the archive location

InputStream inputStream = new ByteArrayInputStream(policyContent);
s3Service.uploadFile(POLICY_BUCKET, destinationKey, inputStream,
policyContent.length);

// Finally, delete the original file
s3Service.deleteFile(POLICY_BUCKET, sourceKey);
}
}
```

The `PolicyService` is an application component that relies on the `S3Service` as part of the policy issue workflow. As the `S3Service` encapsulates all the needed methods, `PolicyService` directly uses those methods to upload and archive policy documents. All these inter-application dependencies are handled in a clean, loosely coupled way through the Jakarta CDI.

Your Application In Payara Cloud

So far all the examples we have seen have nothing specific to Payara Cloud. They all use the AWS Java SDK and Jakarta EE constructs to upload, download, delete and archive files in S3. As Payara Cloud applications are pure Jakarta EE Web Profile applications, you can use the range of libraries, SDKs and utilities available in the Java ecosystem in your projects. Payara Cloud only provides you with a hosted Payara Server to deploy your applications to the internet without needing to manage the server and infrastructure manually.

Your applications have full outbound access to connect to any service on the internet. As we have seen in this quick guide, there is nothing Payara Cloud specific we have used. Everything has been 100% Java and Jakarta EE based. To see for yourself, sign up for a free trial of Payara Cloud, launch a new application with Payara Starter and take the service for a test drive.

Best Practices for Efficient and Secure S3 File Management

Now let's explore some best practices for using S3 in your Jakarta EE applications. This list is by no means exhaustive, but it aims to give broad pointers to things to keep in mind when adopting S3 into your applications.

Secure Credential Management

Never hardcode AWS credentials, or any credentials for that matter, directly into your code. Instead, use secure mechanisms like AWS Secrets Manager or environment variables through the MicroProfile Config to store and retrieve credentials during runtime. This helps to protect your AWS account from unauthorized access and potential security breaches.

Error Handling and Logging

Implement proper error handling and logging mechanisms within your S3 file management code. We did not cover that in the `S3Service`, but in your production application, that class should have ample logs and try-catch blocks to be able to log meaningful messages and aid in debugging. This will ensure that any errors or exceptions encountered during S3 operations are gracefully handled and logged appropriately. Comprehensive logging supports efficient troubleshooting and helps to identify the root cause of issues quickly.

Large File Uploads

When dealing with large files exceeding 100MB in size, it's recommended to employ S3's multipart upload feature or use the AWS SDK's [TransferManager](#). These approaches enable efficient and reliable handling of large file uploads by breaking them down into smaller parts, allowing for parallel uploads and improved resilience to network interruptions.

Caching

Implement caching strategies to store frequently accessed files locally or in a caching layer. This will reduce the number of requests made to S3, improving performance and reducing latency, particularly for read-heavy workloads. Caching can significantly enhance the user experience by providing faster access to commonly requested files. It can also help reduce your usage bills.

Content-Type Setting

Always set the appropriate Content-Type header when uploading files to S3. This ensures that browsers and other clients can correctly interpret and handle the file type, enabling features like proper rendering of images, videos, or documents when directly linked to from S3.

Versioning

Enable S3 bucket versioning for critical data to safeguard against accidental deletions or overwrites. Versioning maintains a history of all object versions within a bucket, allowing you to restore previous versions if needed. It acts as a safety net for data protection and enables you to roll back changes if necessary. However, beware that this could have some cost implications. So do check the latest pricing page of the S3 service to be sure.

Access Control

Implement fine-grained access control using S3 bucket policies and IAM roles. This ensures that only authorized users and applications have access to specific S3 resources. Restricting access based on need-to-know principles helps to maintain data confidentiality and integrity.

Encryption

Protect sensitive data by enabling server-side encryption for S3 objects. This will ensure that your application data at rest within S3 is encoded and can only be accessed with the appropriate decryption keys. This protects your data from unauthorized access even in the event of physical storage compromise.

Monitoring and Alerts

Set up CloudWatch metrics and alerts to monitor S3 operations, track usage patterns and detect potential issues proactively. Monitoring allows you to gain insights into S3 performance, identify anomalies and take corrective action before they impact your application or users.

Cost Optimization

Use S3 lifecycle policies to automatically transition less frequently accessed data to cheaper storage classes like S3 Intelligent-Tiering, S3 Standard-Infrequent Access (S3 Standard-IA), or S3 Glacier Instant Retrieval. This will help optimize your storage costs without sacrificing data availability or retrieval performance.

Conclusions

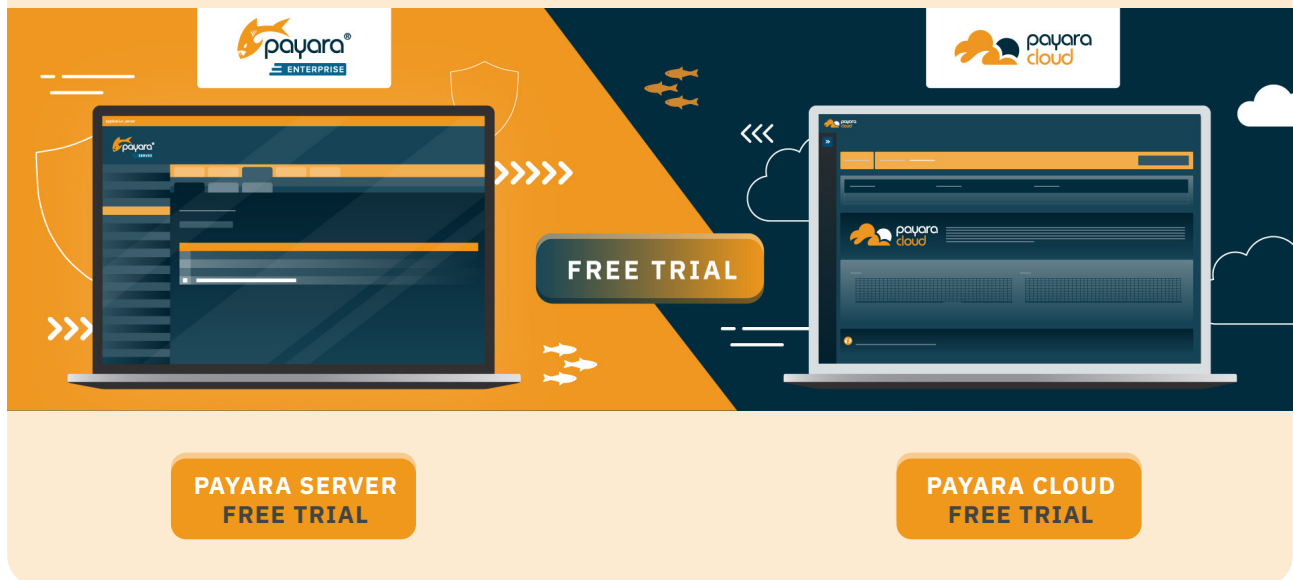
This guide provides a comprehensive overview on how to integrate Amazon S3 with Jakarta EE applications for efficient file storage, with a particular focus on deployment using Payara Cloud. Let's recap the key objectives we've covered:

- We explored common file storage challenges in enterprise applications, including scalability, availability, durability, performance, cost-effectiveness and security.
- We discussed why Amazon S3 is a popular solution, highlighting its virtually unlimited scalability, high availability and durability, performance options, cost-effectiveness and robust security features.
- We outlined key considerations for adopting S3, such as data classification, access patterns, compliance, costs, latency, backup strategies, integration complexity and vendor lock-in concerns.
- We provided a step-by-step implementation guide for integrating S3 into a Jakarta EE application, including:
 - Setting up AWS credentials
 - Configuring the S3 client using CDI
 - Implementing a comprehensive S3 service
 - Creating REST endpoints for file operations
- We demonstrated how Payara Cloud seamlessly supports Jakarta EE applications with S3 integration:
 - Payara Cloud provides a managed Payara Server environment, allowing you to focus on your application logic rather than infrastructure management.
 - The platform supports standard Jakarta EE and Java libraries, enabling straightforward integration with AWS SDK and other third-party services.
 - Payara Cloud applications have full outbound access, facilitating connections to external services like Amazon S3.
- We shared best practices for using S3 with Jakarta EE applications on Payara Cloud, covering security, performance optimization, error handling and cost management.

After reading this guide, you should now be able to integrate S3 into your Jakarta EE applications and deploy them on Payara Cloud. Remember to continually evaluate and optimise your S3 implementation to ensure maximum value, performance and security.

Payara Cloud's managed environment provides full outbound connectivity, allowing you to focus on developing Jakarta EE applications with any third-party integrations without worrying about infrastructure management. To experience this integration first-hand, sign up for a Payara Cloud free trial. Use Payara Starter to quickly deploy your S3-integrated Jakarta EE application and see how Payara Cloud simplifies cloud deployment.

Interested in Payara? Try Before You Buy



sales@payara.fish



UK: +44 800 538 5490
Intl: +1 888 239 8941



www.payara.fish

Payara Services Ltd 2024 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ