(../..)

Initial setup

In the `moviefun` project, checkout the `write-to-blobstore-start` tag.

```
git checkout write-to-blobstore-start
```

You will need Minio to perform this lab. Minio is a blob store that has an S3 compatible API and runs locally very easily.

You can install Minio with Homebrew on MacOS,

```
brew install minio
```

or if you are using Windows you can download it here (https://github.com/minio/minio#microsoft-windows).

Explore the new feature

Build and run the application locally. Look at the `/albums` page. An album's details page should allow you to upload an album cover. Refreshing the page should display the uploaded cover.

Deploy to PCF. Can you add an album cover?

Now scale the application to 2 instances. Return to the album's details page in your browser. In your browser disable/reset the cache: in Chrome, go to the dev tools, select `Settings` (F1), and enable `Preferences > Network > Disable cache (while DevTools is open)`. What happens? Why?

Look at the new code

Have a look at the `AlbumsController` class. It implements album cover persistence. It reads from and writes files to the file system.

Introduce the `BlobStore` abstraction

Let's create a new `BlobStore` interface:

```
public interface BlobStore {

    void put(Blob blob) throws IOException;

    Optional<Blob> get(String name) throws IOException;
}
```

And the `Blob` class:

```
public class Blob {
    public final String name;
    public final InputStream inputStream;
    public final String contentType;

    public Blob(String name, InputStream inputStream, String contentType) {
        this.name = name;
        this.inputStream = inputStream;
        this.contentType = contentType;
    }
}
```

Now create a `FileStore` implementation:

```
public class FileStore implements BlobStore {

    @Override
    public void put(Blob blob) throws IOException {
        ...
    }

    @Override
    public Optional<Blob> get(String name) throws IOException {
        ...
    }
}
```

Extract the code from the `AlbumsController` into the `FileStore`. Replace the existing implementation in the `AlbumsController` by using the newly implemented `FileStore`.

Create an Amazon S3 implementation of the `BlobStore`

Follow these steps:

- Add the AWS Java SDK to your `pom.xml`.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-java-sdk</artifactId>
    <version>1.11.15</version>
</dependency>
```

- Add S3 configurations to your `application.yml`.

```
s3:
  endpointUrl: http://127.0.0.1:9000
  accessKey: <ignore this for now>
  secretKey: <ignore this for now>
  bucketName: moviefun
```

- Create a `@Bean` function for your upcoming `BlobStore` implementation.

```
@Value("${s3.endpointUrl}") String s3EndpointUrl;
@Value("${s3.accessKey}") String s3AccessKey;
@Value("${s3.secretKey}") String s3SecretKey;
@Value("${s3.bucketName}") String s3BucketName;

@Bean
public BlobStore blobStore() {
    AWSCredentials credentials = new BasicAWSCredentials(s3AccessKey, s3SecretKey);
    AmazonS3Client s3Client = new AmazonS3Client(credentials);

    s3Client.setEndpoint(s3EndpointUrl);

    return new S3Store(s3Client, s3BucketName);
}
```

- Implement the `S3Store` class.

  Refer to the S3 library documentation (https://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/s3/AmazonS3Client.html) for more details.

Testing it locally.

Create a local folder for the Minio server and the bucket, `moviefun` we will use.

```
mkdir -p ~/shared/moviefun
```

Run the Minio server, it will output some credentials for you to use:

```
minio server ~/shared
```

Setup the `application.yml`.

```
s3:
  endpointUrl: http://localhost:9000
  accessKey: <enter the minio access key id>
  secretKey: <enter the minio secret access key>
  bucketName: moviefun
```

Start the application.

```
mvn spring-boot:run
```

Now verify that album upload works and correctly writes file into the `~/shared/moviefun` folder. You can use, for example, `open ~/shared/moviefun/covers/13` to open an image that would be at that path.

Testing it on PCF

Deploy the application without starting it, then create the service binding.

```
cf push moviefun --random-route --no-start -p target/moviefun.war
cf create-service aws-s3 standard moviefun-s3
cf bind-service moviefun moviefun-s3
```

Now let's look at the environment variables to configure the connection:

```
cf env moviefun
```

```
Getting env variables for app moviefun in org pal-student / space my-space as dlb@pivotal.io...
OK

System-Provided:
{
 "VCAP_SERVICES": {
  "aws-s3": [
   {
    "credentials": {
     "access_key_id": "AKIAJENM",
     "bucket": "cf-f40e7905",
     "region": "us-east-1",
     "secret_access_key": "6jNtUooI26"
    },
    "label": "aws-s3",
    "name": "moviefun-s3",
    "plan": "standard",
    "provider": null,
    "syslog_drain_url": null,
    "tags": [],
    "volume_mounts": []
   }
  ]
 }
}
```

Inside the `credentials` structure you can see all the data we need to configure in the environment. Considering the above data, we would need to enter the following commands:

Make sure you update this commands with the result of the output of `cf env`

```
cf set-env moviefun S3_ENDPOINTURL http://s3.amazonaws.com
cf set-env moviefun S3_ACCESSKEY AKIAJENM
cf set-env moviefun S3_SECRETKEY 6jNtUooI26
cf set-env moviefun S3_BUCKETNAME cf-f40e7905
```

Now start the application and test it.

```
cf start moviefun
```

Check that the application is working as expected.

Assignment

Once you are done with the lab and the application is deployed and working on PCF, you can submit the assignment using the `submitReplatformingRemovingPersistenceToFileSystem` gradle task. It requires you to provide the `movieFunUrl` project property. For example:

```
cd ~/workspace/assignment-submission
./gradlew submitReplatformingRemovingPersistenceToFileSystem -PmovieFunUrl=http://my-movie-fun.cfapps.io
```

(https://pivotal.io)