Request Processor and Concrete Adapters

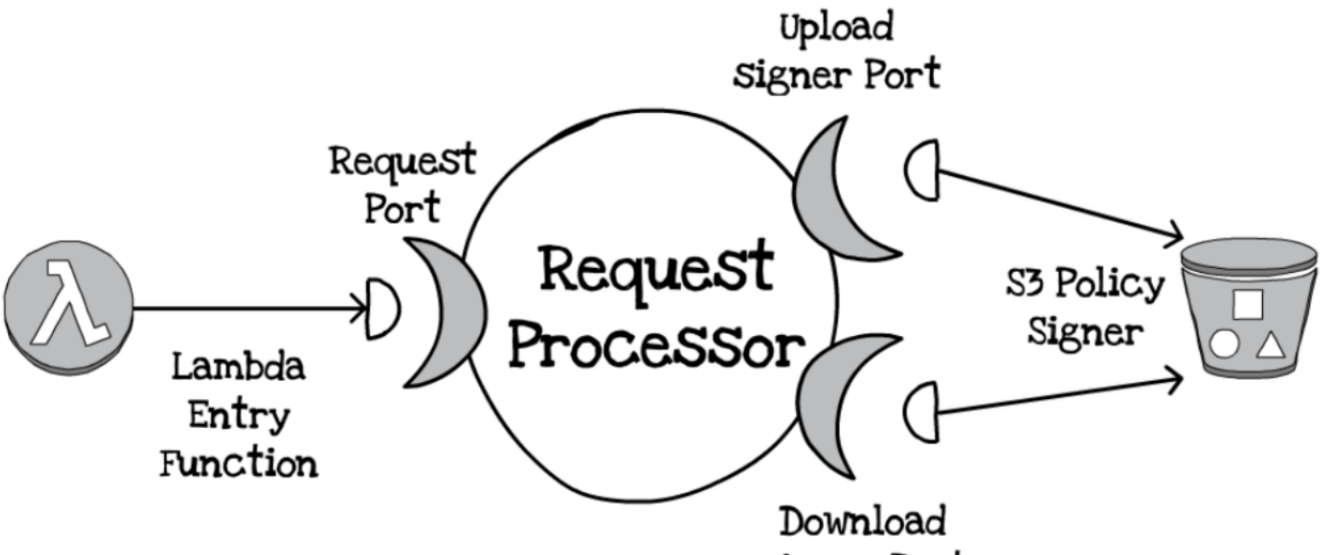
In this lesson, you'll see how to go about implementing the request processor along with concrete adapters using the Ports and Adapters design pattern.

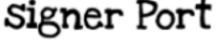
WE'LL COVER THE FOLLOWING

- Writing unit tests for core components
- Excluding test resources from deployment
- Infrastructure adapters

For the `ShowFormFunction` handler, the core business logic is to validate that a requested upload has an extension and that the extension belongs to a list of approved file types. For valid extensions, the function needs to create a unique upload file name based on the request ID and the extension, and produce a signed upload policy and a signed download policy. This business core will have three ports (shown in the figure provided below):

1. The first port should provide the request ID and the extension, starting the process, and format the resulting signatures accordingly.
2. The second port should sign upload policies based on a key.
3. The third port should sign download policies based on a key.





Ports and adapters for signing S3 policies using Lambda functions

The core business logic should not care about AWS interfaces, but instead describe the workflow using concepts important for a specific use case, such as signing uploads and downloads. You can create a new file called `request-processor.js` in the `user-form` directory with the code from the following listing.

```
module.exports = class RequestProcessor {
  constructor(uploadSigner, downloadSigner, uploadLimitInMB, allowedExtensions) {
    this.uploadSigner = uploadSigner;
    this.downloadSigner = downloadSigner;
    this.uploadLimitInMB = uploadLimitInMB;
    this.allowedExtensions = allowedExtensions;
  };
  processRequest(requestId, extension) {
    if (!extension) {
      throw `no extension provided`;
    }
    const normalisedExtension = extension.toLowerCase();
    const isImage = this.allowedExtensions.includes(normalisedExtension);
    if (!isImage) {
      throw `extension ${extension} is not supported`;
    }
    const fileKey = `${requestId}.${normalisedExtension}`;
    return {
      upload: this.uploadSigner.signUpload(fileKey, this.uploadLimitInMB),
      download: this.downloadSigner.signDownload(fileKey)
    };
  };
};
```

code/ch12/user-form/request-processor.js

Writing unit tests for core components

You can test the `RequestProcessor` class in isolation, without network access or talking to actual AWS services. During the test setup, you could create simple in-memory adapters for the ports required by the workflow. The next listing shows what a unit test would look like using `jest`, a popular JavaScript unit testing tool.

```
const RequestProcessor = require('../request-processor');
describe('request-processor', () => {
  let uploadSigner, downloadSigner,
    uploadLimit, allowedExtensions, underTest;
  beforeEach(() => {
```

```
  beforeEach(() => {
    uploadSigner = {
      signUpload: jest.fn((key, limit) => `upload-max-${limit}-${key}`)
    };
    downloadSigner = {
      signDownload: jest.fn((key) => `download-${key}`)
    };
    uploadLimit = 10;
    allowedExtensions = 'jpg,gif';
    underTest = new RequestProcessor(
      uploadSigner, downloadSigner, uploadLimit, allowedExtensions
    );
  });
  describe('processRequest', () => {
    test('rejects request without an extension', () => {
      expect(() => underTest.processRequest('req-id'))
        .toThrow('no extension provided');
    });
    test('rejects requests with an unsupported extension', () => {
      expect(() => underTest.processRequest('req-id', 'xls'))
        .toThrow('extension xls is not supported');
    });
    test('signs a request for supported extensions', () => {
      expect(underTest.processRequest('req-id', 'jpg'))
        .toEqual({
          "download": "download-req-id.jpg",
          "upload": "upload-max-10-req-id.jpg"
        });
    });
  });
});
```

code/user-form/tests/request-processor.test.js

Of course, this is just a start. To keep things simple in this course, you've only seen the basic test cases. In a real project, you would add examples for important edge cases.

Isolating the business logic core from infrastructure allows you to quickly check various boundary conditions, and providing simple in-memory implementations for port interfaces lets you reason about the system under test easily. You don't have to worry about the complexity of the SIGV4 process, differences between AWS endpoints in various regions, and network communications. All of that can be tested separately, with an integration test for actual infrastructure adapters.

Excluding test resources from deployment

Once you start putting tests and testing tools into the project, you need to reconfigure Lambda packaging. There's no reason to include test cases, libraries, or tools (such as `jest`) in the deployment package SAM sends to

Lambda. SAM can help create a minimal clean package during the `sam build` step, but it is not smart enough to guess all the possible testing tools. It does,

however, understand project manifests for many popular programming languages, so it's really important to properly specify the type of testing dependencies.

For JavaScript, that means using the `devDependencies` section of `package.json` (or installing NPM packages using the `-D` flag) for third-party tools you want to exclude from Lambda packaging. In the case of Node.js functions, SAM also uses the standard `npm pack` configuration to decide whether to include source files in a package. This means that you can explicitly list required files in the `files` section of `package.json`, or exclude directories and files by adding them to a file called `.npmignore` in the project directory. For example, the package manifest from the following listing would ensure that only root-level javascript files are included (so excluding the `tests` subdirectory) and that `jest` is available for development but never uploaded to Lambda.

```
{
  "name": "user-form",
  "version": "1.0.0",
  "license": "MIT",
  "files": [
    "*.js"
  ],
  "scripts": {
    "test": "jest"
  },
  "devDependencies": {
    "got": "^9.6.0",
    "jest": "^24.8.0"
  }
}
```

code/ch12/user-form/package.json

Infrastructure adapters

You need a concrete implementation of the upload and download signer ports which can talk to AWS S3. These two operations are closely related, so you can put them both in the same class. This class is specific to S3, so it can use the AWS SDK classes. A file called `s3-policy-signer.js` is created in the `user-form` Lambda directory, with the code from the following listing.

```
const aws = require('aws-sdk');
```

```
const s3 = new aws.S3();
module.exports = class S3PolicySigner {
  constructor(bucketName, expiry) {

    this.bucketName = bucketName;
    this.expiry = expiry || 600;
  };
  signUpload(key, uploadLimitInMB) {
    const uploadParams = {
      Bucket: this.bucketName,
      Expires: this.expiry,
      Conditions: [
        ['content-length-range', 1, uploadLimitInMB * 1000000]
      ],
      Fields: { acl: 'private', key: key }
    };
    return s3.createPresignedPost(uploadParams);
  };
  signDownload(key) {
    const downloadParams = {
      Bucket: this.bucketName,
      Key: key,
      Expires: this.expiry
    };
    return s3.getSignedUrl('getObject', downloadParams);
  };
};
```

code/ch12/user-form/s3-policy-signer.js

Similar to the request processor test, you could set up a small integration test for this class that uploads a file to S3, generates a signature, and tries to download it just to make sure that the whole process works end to end. The following listing shows what such a test would look like using `jest`.

```
const aws = require('aws-sdk'),
  get = require('got'),
  s3 = new aws.S3(),
  S3PolicySigner = require('../s3-policy-signer');
describe('s3-policy-signer', () => {
  let bucketName;
  beforeAll(() => {
    jest.setTimeout(10000);
    bucketName = `test-signer-${Date.now()}`;
    return s3.createBucket({Bucket: bucketName}).promise();
  });
  afterAll(() => {
    return s3.deleteBucket({Bucket: bucketName}).promise();
  });
  describe('signDownload', () => {
    let fileKey;
    beforeEach(() => {
      fileKey = `test-file-${Date.now()}`;
      return s3.putObject({
        Bucket: bucketName,
        Key: fileKey,
        Body: 'test-file-contents'
      }).promise();
```

```
    });
    afterEach(() => {
      return s3.deleteObject({
        Bucket: bucketName, Key: fileKey
      }).promise();
    });
    test('produces a URL allowing direct HTTPS access', () => {
      const underTest = new S3PolicySigner(bucketName, 600);
      const url = underTest.signDownload(fileKey);
      return get(url)
        .then(r => expect(r.body).toEqual('test-file-contents'));
    });
  });
});
```

ch12/user-form/tests/s3-policy-signer.test.js

This test will run several orders of magnitude more slowly than the request processor unit test, because it uses network infrastructure. It also creates and removes files as well as buckets. You could, in theory, provide a test stub class for the AWS SDK to speed things up, but you would lose a lot of confidence. Using a stub would only prove that your adapter calls the SDK as expected, not that the whole process works. For integration tests, it is often best to test the actual infrastructure.

The Ports and Adapters pattern enabled you to separate business logic code from integration code, so you can also separate business logic tests from integration tests. You can run lots of different checks and evaluate expectations for business logic code quickly, without access to any network infrastructure, then run a few slow infrastructural tests independently from that.

If you want to avoid using the actual AWS stack in your tests, and prefer to have a local simulation, check out the LocalStack project.

Get ready to learn about the Lambda utility methods in the next lesson!