HTML to PDF using Typescript & AWS Lambda

Phil Blenkinsop · Follow

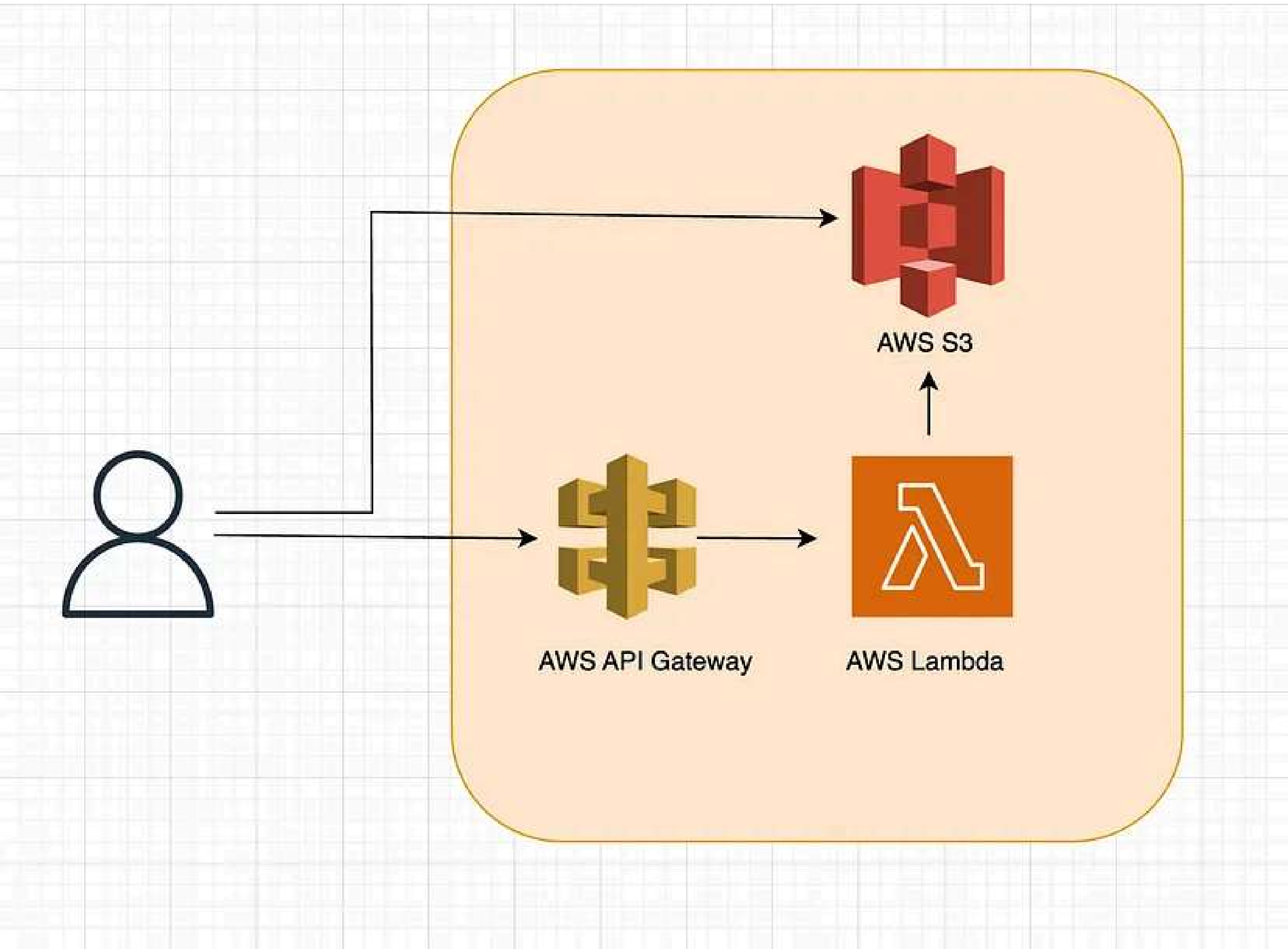
7 min read · Nov 3, 2023

Creating a PDF is a requirement of many applications whether it be a receipt or report. They can often have complex layouts making HTML and CSS a great candidate for designing them.

In Javascript, there are a few packages for creating PDFs such as jsPDF and PDFKit which require you to add content to the PDF using Javascript functions like `doc.text("Hello world!", 10, 10)` . This makes it tricky to generate new PDFs without lots of code changes. However, Puppeteer is an ideal candidate as it can convert HTML to a PDF using a headless browser.

Unfortunately, Puppeteer is tricky to run in a serverless environment due to the binaries it needs to run a headless browser. `@sparticuz/chromium` solves this problem by providing the binaries needed to run chromium in a serverless environment.

For this tutorial we will be building an API that will accept a HTML string, create a PDF of the HTML, upload it to S3 and then return a download url for the PDF.



Architecture Diagram

1. **Project Setup and Dependencies**

Before we begin, we need to make sure we have Node.js installed. I would recommend using node version manager to do this and for this tutorial we are using version 18.18.2 of node. Now we also need to install the AWS CDK in order to use the CLI to initialise and deploy our app. I also prefer using yarn as the package manager so to install both we can run the following command.

```
npm i -g aws-cdk yarn
```

Once installed, make sure you have configured your AWS account with the CDK. https://docs.aws.amazon.com/cdk/v2/guide/getting_started.html

Next we’re going to create a directory to house the project and then initialise it using the AWS CDK CLI. This will create all the files and folders we need to organise the project.

```
mkdir html-pdf-serverless && cd "$_" && cdk init app --language typescript
```

Once the project is initialised, we can install the dependencies needed for our lambda function to generate a PDF and upload it to S3. As we are using yarn we can delete the `package-lock.json` before installing.

Note it’s important to install fixed versions of puppeteer-core and @sparticuz/chromium as puppeteer ships with a preferred version of chromium which are detailed on the puppeteer support page.

```
yarn add @aws-sdk/client-s3 @aws-sdk/s3-request-presigner @aws-sdk/lib-storage p
```

There are also some dev dependencies needed for build and typescript support.

```
yarn add -D @types/aws-lambda aws-lambda  esbuild
```

We need to make two changes to the `tsconfig.json` by adding `skipLibCheck: true` and `esModuleInterop: true` to the compiler options object. This allows our installed dependencies to work with typescript.

The last setup step we need to do is get the binaries needed to run the chromium headless browser in AWS Lambda. To achieve this, we will use a lambda layer which contains the necessary binaries torun chromium.As we installed version 19.6.0 of puppeteer, we need version 110.0.0 of chromium. Head to https://github.com/Sparticuz/chromium/releases, find version 110.0.0 and download `chromium-v110.0.0-layer.zip` , saving it to the root of the repository.

2. Create the API

Our rest API will be made up of 2 routes, one as a health check so we can confirm it has deployed properly, and the second to generate the PDF.

We'll start by creating the healthcheck endpoint and deploying it. Create a healthcheck lambda handler in `lambdas/healthcheck-lambda/index.ts` and add the following code to return a successful response.

```
import { APIGatewayProxyHandler } from "aws-lambda"

export const handler: APIGatewayProxyHandler = async () => {
    return {
        statusCode: 200,
        body: "Ok"
    }
  }
```

Now we can create the CDK resources to handle the lambda code. We modify `lib/html-pdf-serverless-stack.ts` to add a rest API, lambda and lambda integration. This will mean our healthcheck lambda is executed at the root path of the api.

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs'
import { Construct } from 'constructs';
import { LambdaIntegration, RestApi } from 'aws-cdk-lib/aws-apigateway';

export class HtmlPdfServerlessStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    const healthcheckLambda = new NodejsFunction(this, 'HealthcheckLambda', {
      entry: 'lambdas/healthcheck-lambda/index.ts'
    })

    const api = new RestApi(this, 'HtmlToPdfRestApi', {
      restApiName: 'HTML PDF API',
    })

    api.root.addMethod("GET", new LambdaIntegration(healthcheckLambda, {
      requestTemplates: { 'application/json': '{ "statusCode": "200" }' }
    }))
  }
}
```

Now we can run `cdk deploy` and our rest API will be deployed to AWS. The stack should output the url of the api which we can use to test. Opening the outputted url in a browser should return the 'Ok' message we configured in the handler code.

```
Ok
```

3. Add the HTML to PDF Lambda

We can now create the handler for our HTML to PDF lambda in `lambdas/html-pdf-lambda/index.ts` . In the following code we take in the request body, convert the HTML string to a PDF buffer, upload it to S3 and then generate a presigned url to download the saved PDF. The request body the API accepts will include the HTML string and the name of the file that will be stored in S3.

```
import { S3Client, GetObjectCommand } from '@aws-sdk/client-s3'
import { getSignedUrl } from '@aws-sdk/s3-request-presigner'
import { Upload } from '@aws-sdk/lib-storage'
import { APIGatewayProxyEvent, APIGatewayProxyHandler } from 'aws-lambda'
import { generatePdfBuffer } from './generatePdfBuffer'

// process.env.AWS_REGION is a default env var provided in lambdas by AWS
const s3Client = new S3Client({ region: process.env.AWS_REGION })

export const handler: APIGatewayProxyHandler = async (event: APIGatewayProxyEven
    if (!event.body) {
        return {
            statusCode: 400,
            body: "Invalid request body"
        }
    }
    try {
        const requestBody = JSON.parse(event.body) as { html: string, s3Key: str

        const pdfBuffer = await generatePdfBuffer(requestBody.html)

        if(!pdfBuffer) {
            throw new Error('Failed to created PDF buffer from HTML')
        }

        const s3Upload = new Upload({
            client: s3Client,
            params: {
                Bucket: process.env.S3_PDF_BUCKET,
                Body: pdfBuffer,
                Key: requestBody.s3Key
            }
        })
        s3Upload.on("httpUploadProgress", (progress) => {
            console.log(progress);
        });
        await s3Upload.done()

        const presignedUrl = await getSignedUrl(
            s3Client,
            new GetObjectCommand({ Bucket: process.env.S3_PDF_BUCKET, Key: reque
            { expiresIn: 3600 }
        )

        return {
            statusCode: 200,
            body: JSON.stringify({
                pdfUrl: presignedUrl
            })
        }
    } catch (error) {
        console.log("Error converting HTML to PDF", error)
        return {
            statusCode: 500,
            body: "Internal server error"
        }
    }
}
```

We also need to add the function that will handle converting the HTML string to a PDF buffer. Add the following code to `lambdas/html-pdf-lambda/generatePdfBuffer.ts` to launch a headless browser, set the content of the page as the provided HTML and then convert it to a PDF buffer.

```
import chromium from '@sparticuz/chromium'
import puppeteer from 'puppeteer-core'
```

```
export const generatePdfBuffer = async (
  html: string,
): Promise<Buffer | undefined> => {
  let result = undefined
  let browser = null
  try {
    console.log('Launching browser')
    browser = await puppeteer.launch({
      args: chromium.args,
      defaultViewport: chromium.defaultViewport,
      executablePath: await chromium.executablePath(),
      headless: chromium.headless,
      ignoreHTTPSErrors: true,
    })

    console.log('Browser launched')
    const page = await browser.newPage()

    await page.setContent(html, {
      waitUntil: ['domcontentloaded', 'networkidle0', 'load'],
    })

    await page.evaluate('window.scrollTo(0, document.body.scrollHeight)')

    result = await page.pdf({ format: 'a4', printBackground: true })
  } catch (e) {
    console.log('Chromium error', { e })
  } finally {
    if (browser !== null) {
      await browser.close()
    }
  }
  return result
}
```

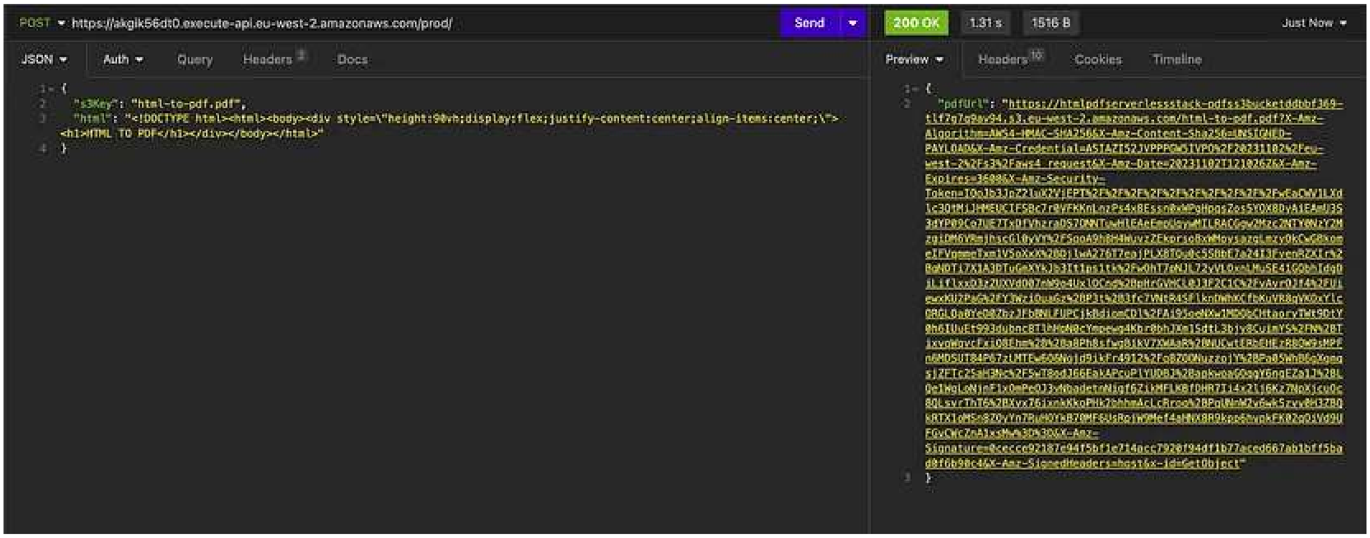
Now our lambda code is ready, we can wire it up to the API we created in the previous step. We need to add the S3 bucket to save the PDFs to, the HTML to PDF lambda, the chromium lambda layer and the rest api integration to wire it together. This leaves our `lib/html-pdf-serverless-stack.ts` looking like the below:

```
import { Duration, Stack, StackProps } from 'aws-cdk-lib';
import { Code, LayerVersion, Runtime } from 'aws-cdk-lib/aws-lambda';
import { Bucket } from 'aws-cdk-lib/aws-s3';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs'
import { Construct } from 'constructs';
import { LambdaIntegration, RestApi } from 'aws-cdk-lib/aws-apigateway';

export class HtmlPdfServerlessStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    const pdfsS3Bucket = new Bucket(this, 'PDFsS3Bucket')

    const chromeAwsLambdaLayer = new LayerVersion(this, 'ChromeAWSLambdaLayer',
      layerVersionName: 'ChromeAWSLambdaLayer',
      compatibleRuntimes: [
        Runtime.NODEJS_18_X
      ],
      code: Code.fromAsset('chromium-v110.0.0-layer.zip')
    })

    const htmlToPdfLambda = new NodejsFunction(this, 'HtmlToPdfLambda', {
      entry: 'lambdas/html-pdf-lambda/index.ts',
      environment: {
        S3_PDF_BUCKET: pdfsS3Bucket.bucketName
      },
      layers: [chromeAwsLambdaLayer],
      bundling: {
        externalModules: [
          'aws-sdk'
        ],
        nodeModules: ['@sparticuz/chromium'],
      },
      timeout: Duration.seconds(30),
      runtime: Runtime.NODEJS_18_X,
      memorySize: 1024
    })
    pdfsS3Bucket.grantReadWrite(htmlToPdfLambda)

    const healthcheckLambda = new NodejsFunction(this, 'HealthcheckLambda', {
      entry: 'lambdas/healthcheck-lambda/index.ts'
    })

    const api = new RestApi(this, 'HtmlToPdfRestApi', {
      restApiName: 'HTML PDF API',
    })

    api.root.addMethod("GET", new LambdaIntegration(healthcheckLambda, {
      requestTemplates: { 'application/json': '{ "statusCode": "200" }' }
    }))
    api.root.addMethod("POST", new LambdaIntegration(htmlToPdfLambda, {
      requestTemplates: { 'application/json': '{ "statusCode": "200" }' }
    }))

  }
}
```

On the HTML to PDF lambda, we've added the name of the S3 bucket as an environment variable so we can upload the PDFs to it. Adding `@sparticuz/chromium` to the nodeModules array of the bundling config tells the bundler not to include this package during the bundling process as it's already available in the runtime due to our lambda layer. Finally, we've registered the lambda layer so that it'll be attached during runtime and increased the timeout and memory size of the lambda. This is because launching the browser is an intensive task that needs processing power and we don't want our lambda to return before it has finished.

Again we can run `cdk deploy` and our changes will be deployed to AWS. Using the same outputted url from the stack, we can test it using a REST client such as Insomnia or Postman with the following body.

```
{
 "s3Key": "html-to-pdf.pdf",
 "html": "<!DOCTYPE html><html><body><div style=\"height:90vh;display:flex;justi
}
```



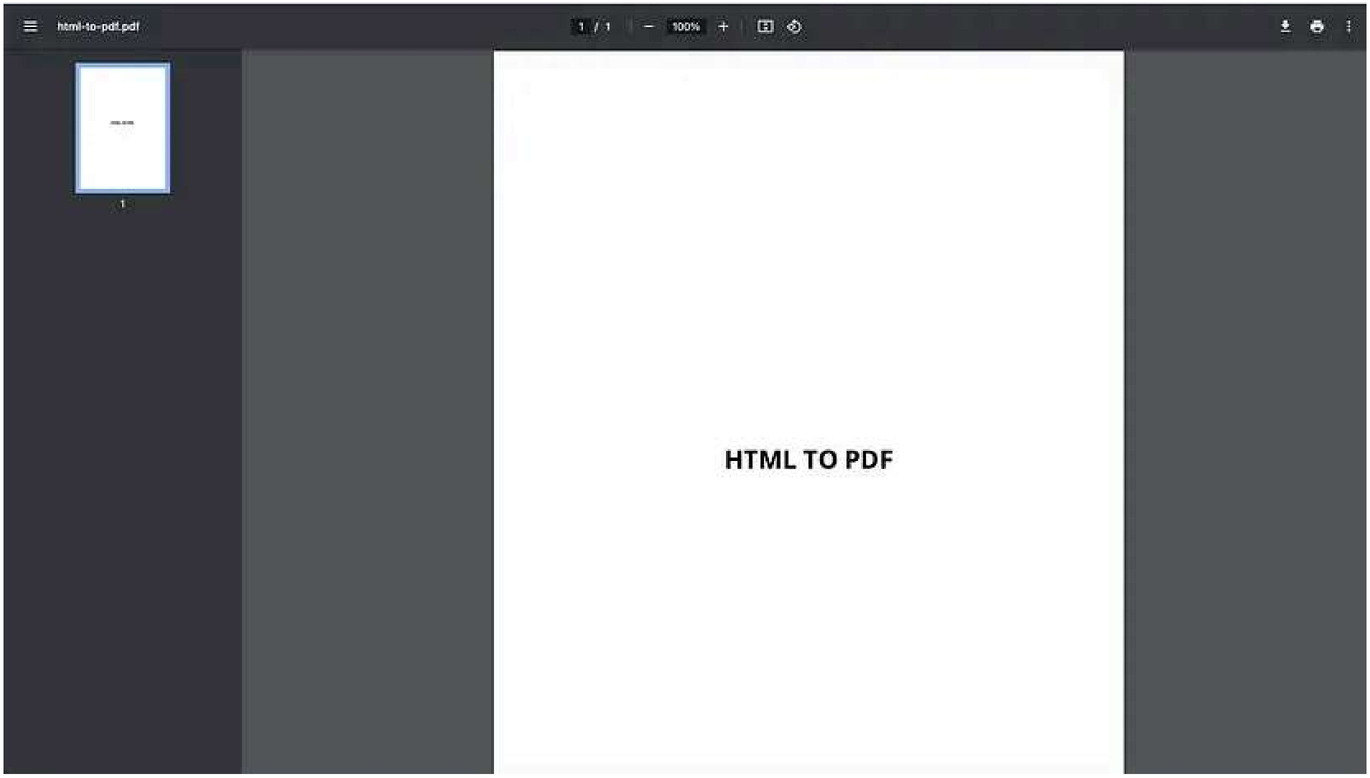
https://akgik56dt0.execute-api.eu-west-2.amazonaws.com/prod/

```
{
  "s3Key": "html-to-pdf.pdf",
  "html": "<!DOCTYPE html><html><body><div style=\"height:90vh;display:flex;justify-content:center;align-items:center;\">
<h1>HTML TO PDF</h1></div></body></html>"
}
```

```
{
  "pdfUrl": "https://htmlpdfserverlessstack-pdfss3bucketddbbf369-
tlf7g7g9av94.s3.eu-west-2.amazonaws.com/html-to-pdf.pdf?X-Amz-
Algorithm=AWS4-HMAC-SHA256&X-Amz-Content-Sha256=UNSIGNED-
PAYLOAD&X-Amz-Credential=ASIAZT52JVPPPGW5IVPO%2F20231102%2Feu-
west-2%2Fs3%2Faws4_request&X-Amz-Date=20231102T121026Z&X-Amz-
Expires=3600&X-Amz-Security-
Token=IQoJb3JpZ2luX2VjEPT%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCWV1LXd
lc3QtMiJHMEUCIFSBc7r0VFKKnLnzPs4xBEssn0xWPgHpgsZos5YOX8DyAiEAmU3S
3dYP09Co7UE7TxDfVhzraDS7ONNTuwHlEAeEmpUgywMILRACGgw2MzcZNTY0NzY2M
zgiDM6YRmjhscGl0yVY%2FSooA9hBH4WuvzZEkprsoBxWMoysazgLmzy0kCwG8kom
eIFVgmmeTxm1V5oXxX%2BDjlwA276T7eajPLX8TGu0c5SBbE7a24I3FvenRZXIr%2
BaNOTi7X1A3DTuGmXYkJb3It1ps1tk%2FwOhT7eNJL72yVLOxnLMuSE41GObhIdq0
iLiflxxD3z2UXVd007nM9o4UxlOCnd%2BpHrGVMCL013F2C1C%2FvAvr0J44%2FUi
ewxKU2PaGh2FY3Wzi0uaGz%2BP3t%2B3fc7VNtR4SFlknDWhKCfbKuVR8gVKOxYlc
ORGL0a0YeD0ZbzJFbNNLFUPCjkBdiomCDl%2FAi95oeNXw1MD0bCHtaoryTWt9DtY
0h6IUuEt993dubncBT1hHpN0cYmpewg4Kbr0bhJXm15dtL3bjy8CuimYS%2FN%2BT
ixvgWovcFxi08Ehm%2B%2Ba8Ph8sfwg8ikV7XWAaR%2BNUCwtERbEHEzRBOW9sMPF
nAMDSUT84P67zLMTEw606NojJd9ikFr4912%2Fq8ZO0NuzzojY%2BPa0SWh86gXqmg
sjZFTc2SaH3Nc%2FSwT8odJ66EakAPcuPlYUDBJ%2BapkwoaGOqgY6ngEZa1J%2BL
Qe1WgLoNjnF1xOmPeOJ3vNbadetnNigf6ZikMFLKBfDHR7I14x2Ij6Kz7NpXjcu0c
8QLsvrThT6%2BXvx76ixnkKkoPHk7bhhmAcLcRroo%2BPqWnW2v6wk5zvy0H3ZBQ
kRTX1oMSn8Z0yYn7BuHOYkB70MF6UsBoiW9Mef4aHNXBR9kpp6hvpkFK0zqQiVd9U
FGvCWcZnA1xsMw%3D%3D&X-Amz-
Signature=0cecce92187e94f5bf1e714acc7920f94df1b77aced667ab1bff5ba
d0f6b98c4&X-Amz-SignedHeaders=host&x-id=GetObject"
}
```

Test using Insomina

Then clicking the returned presigned URL to download the PDF shows us our HTML converted into the PDF document!



Generated PDF from HTML

Thanks for reading and you can check out the sample repo for the full code! https://github.com/blenky36/html-pdf-serverless

AWS Pdf AWS Lambda Aws Cdk Typescript

Written by Phil Blenkinsop

10 Followers · 11 Following

Follow

Responses (2)

What are your thoughts?

Respond

Kundan

8 days ago

•••

how to solve issue of CORS.?

Reply

A

Atul Goel

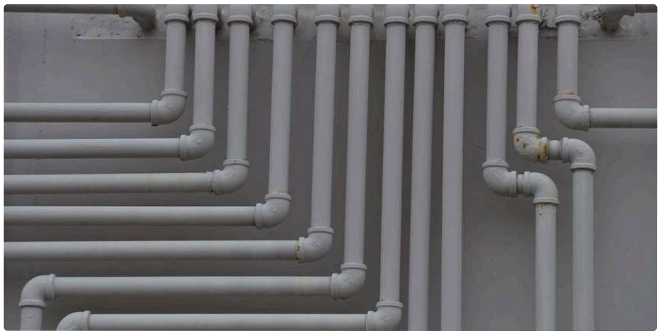
11 months ago

•••

This was really helpful. Thanks a lot!

Reply

More from Phil Blenkinsop

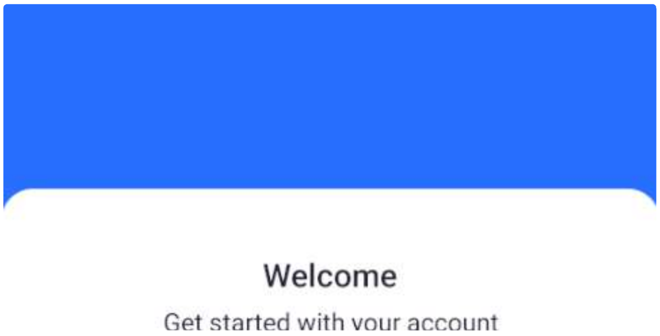


Phil Blenkinsop

CircleCI Config Splitting

CircleCI is one of the most popular CI/CD providers and offers a lot of features when it…

Dec 7, 2023



Welcome

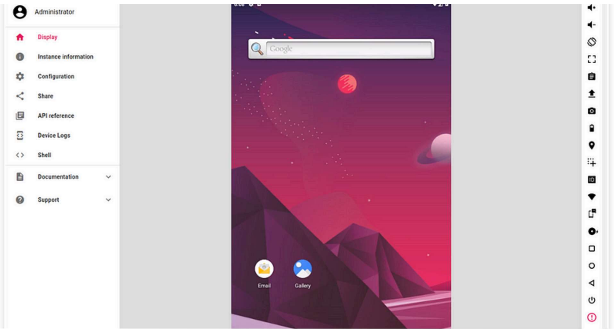
Get started with your account

In Econify by Phil Blenkinsop

(1/2) E2E Testing Android Apps With ADB

Testing your main user journeys to find any issues before they reach your end users is a…

Dec 8, 2023 67 1



Phil Blenkinsop

(2/2) E2E Testing Android Apps With ADB In The Cloud Using...

This article follows on from my first post about E2E Testing Android Apps With ADB. I...

Dec 18, 2023 7 1

In Dunelm Technology by Phil Blenkinsop

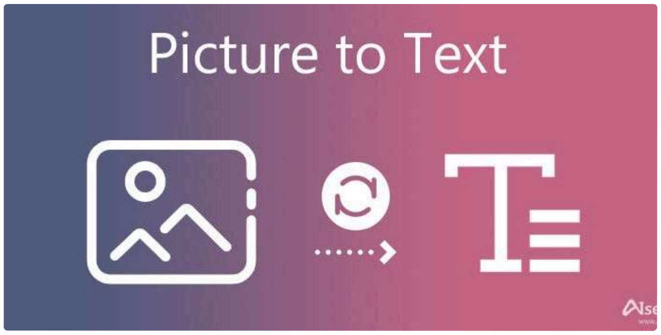
Deploying Typescript Lambdas to AWS with the CDK

It's never been easier to write typescript lambdas thanks to the @aws-cdk/aws-...

Jan 26, 2022 163

See all from Phil Blenkinsop

Recommended from Medium



Mohammed shamseer pv

Building an Image to Text OCR Application in Node.js Using...

Optical Character Recognition (OCR) is a powerful technology that extracts text from...

Sep 25

Harendra

How I Am Using a Lifetime 100% Free Server

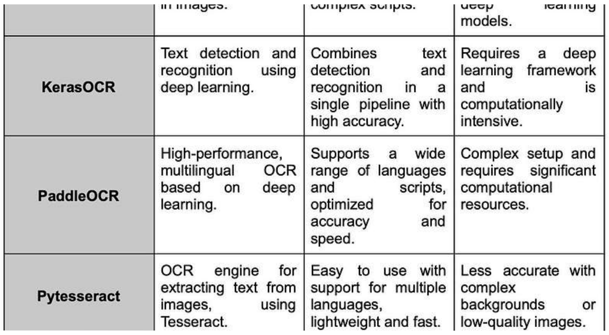
Get a server with 24 GB RAM + 4 CPU + 200 GB Storage + Always Free

Oct 26 8.3K 135

Lists

Natural Language Processing

1881 stories · 1514 saves



	in images.	complex scripts.	deep learning models.
KerasOCR	Text detection and recognition using deep learning.	Combines text detection and recognition in a single pipeline with high accuracy.	Requires a deep learning framework and is computationally intensive.
PaddleOCR	High-performance, multilingual OCR based on deep learning.	Supports a wide range of languages and scripts, optimized for accuracy and speed.	Complex setup and requires significant computational resources.
Pytesseract	OCR engine for extracting text from images, using Tesseract.	Easy to use with support for multiple languages, lightweight and fast.	Less accurate with complex backgrounds or low-quality images.

Shah Vansh

Comparison of text detection techniques: easyOCR vs kerasOC...

In today's digital age, ability to extract text from all images and document. This includes...

Aug 16 3

```
const [err, res] ?=
  await fetch('https://codingbeautydev.com');
```

In Coding Beauty by Tari Ibaba

This new JavaScript operator is an absolute game changer

Say goodbye to try-catch

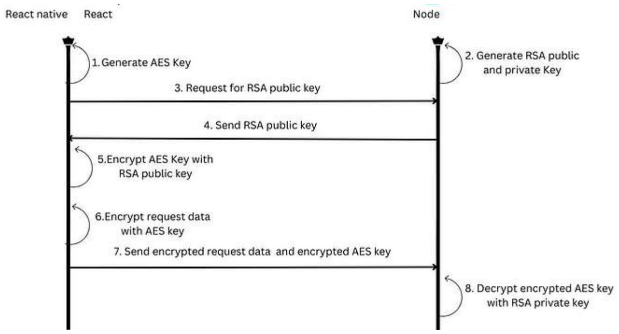
Sep 18 6.4K 93

In AWS Tip by Mehdi BAFDIL

I Reduced AWS Bills by 80% Just by Optimizing Node.js Code

The Art of AWS Cost-Cutting: A Node.js Optimization Journey

Dec 19 327 5


React native
React
Node
1.Generate AES Key
2. Generate RSA public and private Key
3. Request for RSA public key
4. Send RSA public key
5.Encrypt AES Key with RSA public key
6.Encrypt request data with AES key
7. Send encrypted request data and encrypted AES key
8. Decrypt encrypted AES key with RSA private key

Anshu Sharma

Encryption in React JS and Native and Node js using node-forge

Choose the right module for encryption

Sep 17 11

See more recommendations