WordMaps

Serverless Web Application for Word Graph Exploration

Juan López de Hierro
Sergio Muela Santos

January 2026

Resumen

Este documento presenta WordMaps, una aplicación web serverless que implementa un sistema de exploración de grafos de palabras en inglés de 3, 4 y 5 letras. La aplicación utiliza algoritmos de búsqueda en grafos para encontrar caminos óptimos entre palabras que difieren en un solo carácter. El sistema está construido con una arquitectura moderna basada en AWS utilizando Lambda, API Gateway, S3 y CloudFront, demostrando las capacidades y ventajas de la computación serverless para aplicaciones web especializadas.

Palabras clave: Serverless, AWS Lambda, Grafos, BFS, Spring Boot, React, API Gateway, CloudFront

Índice general

Capítulo 1

Introducción

1.1. Motivación

El problema de transformación de palabras es un desafío clásico en ciencia de la computación que combina teoría de grafos, estructuras de datos y algoritmos de búsqueda. WordMaps aplica este concepto en un entorno web moderno, permitiendo a los usuarios explorar interactivamente las relaciones entre palabras de manera intuitiva y visual.

1.2. Objetivos

Los objetivos principales del proyecto son:

- Implementar un grafo de palabras eficiente en memoria donde cada nodo representa una palabra de 3, 4 o 5 letras
- Desarrollar algoritmos de búsqueda óptimos para encontrar caminos entre palabras
- Crear una interfaz web interactiva y responsiva
- Demostrar el uso de arquitectura serverless en AWS
- Minimizar costos operacionales utilizando servicios de AWS Free Tier
- Implementar un pipeline de CI/CD automatizado con GitHub Actions

1.3. Alcance

El proyecto abarca:

- Backend desarrollado en Spring Boot 3 con Java 17
- Frontend construido con React 18 y Vite
- Infraestructura completamente serverless en AWS
- Implementación del algoritmo BFS para búsqueda de caminos
- API RESTful para operaciones de grafos

- Documentación técnica completa del proceso y decisiones arquitectónicas
- Aplicación en producción accesible públicamente

Capítulo 2

Arquitectura del Sistema

2.1. Arquitectura General

WordMaps implementa una arquitectura de tres capas distribuida en servicios de AWS:

1. **Capa de Presentación**: Aplicación React servida desde S3 a través de CloudFront
2. **Capa de Aplicación**: Función Lambda ejecutando Spring Boot
3. **Capa de Enrutamiento**: API Gateway HTTP API v2

El diseño serverless garantiza escalabilidad automática, sin necesidad de gestionar servidores, y pago basado en consumo real.

2.2. Componentes AWS

2.2.1. CloudFront Distribution

CloudFront actúa como CDN global, distribuyendo contenido frontend estático desde ubicaciones edge cercanas a los usuarios. Proporciona:

- Terminación automática de SSL/TLS
- Caché de contenido estático
- Compresión GZIP/Brotli
- Protección básica contra DDoS
- Latencia global reducida

2.2.2. S3 Bucket

El bucket S3 almacena archivos estáticos del frontend (HTML, CSS, JavaScript). La configuración incluye:

- Acceso restringido mediante Origin Access Identity (OAI)
- Configuración de sitio web para enrutamiento SPA
- Versionado deshabilitado para reducir costos
- Política de acceso público bloqueada

2.2.3. API Gateway HTTP API

API Gateway HTTP API v2 ofrece ventajas sobre REST API tradicional:

- Latencia más baja (hasta 60 % más rápido)
- Costo significativamente menor (70 % más barato)
- Integración nativa con Lambda
- Formato de carga optimizado versión 2.0
- Soporte automático de CORS

2.2.4. Lambda Function

La función Lambda ejecuta la aplicación Spring Boot con las siguientes características:

- **Runtime**: Java 17 (Amazon Corretto)
- **Memoria**: 2048 MB
- **Timeout**: 30 segundos
- **SnapStart**: Habilitado para reducir cold starts
- **Variables de entorno**: Optimizaciones JVM

2.3. Flujo de Datos

El flujo típico de una petición es:

1. El usuario accede a `https://d2euump6bzpha5.cloudfront.net`
2. CloudFront sirve el frontend desde S3
3. La aplicación React realiza petición AJAX a la API
4. API Gateway recibe la petición y la transforma al formato HTTP API v2
5. Lambda procesa la petición con Spring Boot
6. Algoritmo BFS encuentra el camino óptimo
7. Respuesta JSON se devuelve al usuario
8. React renderiza el resultado y D3.js visualiza el grafo

Capítulo 3

Stack Tecnológico

3.1. Backend

3.1.1. Spring Boot 3

Spring Boot fue elegido por:

- Ecosistema maduro y bien documentado
- Inyección de dependencias robusta
- Soporte nativo para AWS Lambda mediante `aws-serverless-java-container`
- Facilidad para testing
- Auto-configuración inteligente

3.1.2. Java 17

Java 17 LTS ofrece:

- Records para modelado de datos inmutables
- Pattern matching mejorado
- Mejor rendimiento que Java 11
- Soporte completo en AWS Lambda
- Largo período de soporte (hasta septiembre 2029)

3.2. Frontend

3.2.1. React 18

React proporciona:

- Arquitectura basada en componentes
- Virtual DOM para renderizado eficiente

- Hooks para gestión de estado
- Ecosistema extenso de librerías
- Hot Module Replacement con Vite

3.2.2. Vite

Vite reemplaza Webpack con:

- Hot Module Replacement instantáneo
- Tiempos de construcción significativamente más rápidos
- Configuración mínima
- Tree-shaking optimizado
- Bundling optimizado para producción

3.2.3. TailwindCSS

TailwindCSS facilita:

- Desarrollo rápido con clases utilitarias
- Reducción de tamaño de bundle con purging
- Diseño responsivo sin media queries personalizadas
- Consistencia visual en toda la aplicación

3.2.4. D3.js

D3.js proporciona:

- Visualización interactiva del grafo
- Simulación de fuerzas para posicionamiento
- Animaciones suaves
- Interactividad mejorada del usuario

3.3. Infraestructura

3.3.1. AWS SAM

SAM (Serverless Application Model) facilita:

- Definición declarativa de infraestructura
- Testing local de funciones Lambda
- Despliegue automatizado
- Integración con CloudFormation
- Validación de sintaxis YAML

3.3.2. GitHub Actions

GitHub Actions proporciona:

- CI/CD nativo integrado en GitHub
- Automatización sin servidores adicionales
- Triggers en push, pull requests y schedule
- Secretos seguros para credenciales
- Workflows declarativos en YAML

Capítulo 4

Implementación

4.1. Estructura del Grafo

El grafo de palabras se representa mediante una lista de adyacencia implementada con HashMap para acceso O(1):

```
private Map<String, List<String>> adjacencyList;

public void buildGraph(Set<String> words) {
    for (String word : words) {
        adjacencyList.put(word, new ArrayList<>());
        for (String otherWord : words) {
            if (differsByOneLetter(word, otherWord)) {
                adjacencyList.get(word).add(otherWord);
            }
        }
    }
}

private boolean differsByOneLetter(String w1, String w2) {
    if (w1.length() != w2.length()) return false;
    int differences = 0;
    for (int i = 0; i < w1.length(); i++) {
        if (w1.charAt(i) != w2.charAt(i)) differences++;
    }
    return differences == 1;
}
```

Listing 4.1: Construcción del grafo de palabras

4.2. Algoritmo BFS

La implementación de Breadth-First Search garantiza encontrar el camino más corto:

```
public Route findFastestRoute(String origin, String destination)
   {
    Queue<String> queue = new LinkedList<>();
    Map<String, String> parent = new HashMap<>();
```

```
    Set<String> visited = new HashSet<>();

    queue.offer(origin);
    visited.add(origin);

    while (!queue.isEmpty()) {
        String current = queue.poll();

        if (current.equals(destination)) {
            return reconstructPath(origin, destination, parent);
        }

        for (String neighbor : getNeighbors(current)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                parent.put(neighbor, current);
                queue.offer(neighbor);
            }
        }
    }

    return null; // No path found
}
```

Listing 4.2: Algoritmo BFS para búsqueda de camino más corto

Complejidad: O(V + E) donde V es el número de palabras y E el número de aristas.

4.3. Lambda Handler

El handler de Lambda integra Spring Boot con AWS:

```
public class LambdaHandler implements RequestStreamHandler {
    private static SpringBootLambdaContainerHandler<
        HttpApiV2ProxyRequest, AwsProxyResponse> handler;

    static {
        try {
            handler = SpringBootLambdaContainerHandler
                .getHttpApiV2ProxyHandler(
                    WordMapsApplication.class);
        } catch (ContainerInitializationException e) {
            throw new RuntimeException(
                "Could not initialize Spring Boot", e);
        }
    }

    @Override
    public void handleRequest(
        InputStream inputStream,
        OutputStream outputStream,
        Context context) throws IOException {
```

```
21          handler.proxyStream(inputStream,
22              outputStream, context);
23      }
24  }
```

Listing 4.3: Lambda Handler para Spring Boot

4.4. API Endpoints

La API expone los siguientes endpoints:

Método	**Ruta**	**Descripción**
GET	`/api/routes/fastest`	Encontrar camino óptimo
GET	`/api/graph/stats`	Estadísticas del grafo
GET	`/api/graph/top-connected`	Palabras más conectadas
GET	`/api/graph/isolated`	Palabras aisladas
GET	`/api/words/{word}/exists`	Verificar existencia
GET	`/api/words/{word}/neighbors`	Obtener palabras vecinas
GET	`/api/words/search`	Búsqueda por patrón

Cuadro 4.1: Endpoints disponibles de la API

Capítulo 5

Proceso de Despliegue

5.1. Pipeline de Despliegue

El despliegue sigue estos pasos:

5.1.1. Compilación Backend

```
cd wordmaps-backend
mvn clean package -DskipTests
```

Listing 5.1: Compilación Maven

Genera un JAR todo-en-uno con todas las dependencias incluidas.

5.1.2. Build y Despliegue SAM

```
sam build
sam deploy
```

Listing 5.2: Build y despliegue con SAM

SAM empaqueta la aplicación y crea/actualiza el stack de CloudFormation.

5.1.3. Compilación Frontend

```
cd wordmaps-frontend
npm install
npm run build
```

Listing 5.3: Build frontend con Vite

Vite genera assets optimizados en el directorio `dist/`.

5.1.4. Sincronización S3

```
aws s3 sync ./dist s3://wordmaps-frontendbucket \
    --delete
```

Listing 5.4: Sincronizar con S3

5.1.5. Invalidación CloudFront

```
aws cloudfront create-invalidation \
    --distribution-id E3PT8S6ST3CUNS \
    --paths "/*"
```

Listing 5.5: Invalidar cache CloudFront

5.2. Problemas Resueltos

5.2.1. 502 Bad Gateway

Causa: Incompatibilidad entre `PayloadFormatVersion: "1.0"` en template y `HttpApiV2ProxyHand` en código.

Solución: Actualizar `template.yaml` a `PayloadFormatVersion: "2.0"`.

5.2.2. Indentación YAML

Causa: Bloque Environment incorrectamente indentado bajo Architectures.

Solución: Corregir indentación moviendo Environment al mismo nivel que Architectures.

5.2.3. Caché de CloudFront

Causa: CloudFront sirviendo versiones en caché del frontend antiguo.

Solución:

- Configurar headers Cache-Control en index.html
- Invalidar caché de CloudFront después de cada despliegue

Capítulo 6

Métricas de Rendimiento

6.1. Tiempos de Respuesta

Las mediciones muestran:

Endpoint	Cold Start	Warm Start
`/api/routes/fastest`	2.5s	120ms
`/api/graph/stats`	2.3s	65ms
`/api/words/{word}/neighbors`	2.2s	45ms

Cuadro 6.1: Latencias medidas (con SnapStart habilitado)

6.2. Optimizaciones Aplicadas

1. **SnapStart**: Reduce cold starts aproximadamente 60 %
2. **Tiered Compilation**: Optimización JVM para startup rápido
3. **2048 MB Memory**: Balance entre costo y rendimiento
4. **CloudFront Caching**: TTL de 24 horas para assets estáticos
5. **Compresión**: GZIP en API Gateway y CloudFront

6.3. Costos Operacionales

Bajo AWS Free Tier, el proyecto opera sin costo:

- **Lambda**: 1M invocaciones/mes (gratis)
- **API Gateway**: 1M requests/mes (gratis primeros 12 meses)
- **S3**: 5GB almacenamiento (gratis)
- **CloudFront**: 1TB transferencia/mes (gratis primeros 12 meses)

Después de Free Tier, el costo estimado es aproximadamente $2-5/mes con moderada actividad.

Capítulo 7

Resultados

7.1. Funcionalidad Verificada

La aplicación desplegada en producción demuestra:

- Búsqueda exitosa de caminos entre palabras (ej: CAT → FAT)
- Respuestas consistentes y correctas del algoritmo BFS
- Alta disponibilidad a través de servicios AWS gestionados
- Rendimiento aceptable para usuarios finales
- Escalado automático bajo carga

7.2. Ejemplo de Transformación

Para la búsqueda de CAT a FAT, el sistema devuelve:

```
{
  "origin": "CAT",
  "destination": "FAT",
  "path": ["CAT", "FAT"],
  "steps": 1,
  "routeType": "FASTEST",
  "difficulty": "EASY",
  "transformations": [
    {
      "from": "CAT",
      "to": "FAT",
      "stepNumber": 1,
      "description": "Change 'C' -> 'F' at position 1"
    }
  ]
}
```

Listing 7.1: Respuesta JSON de transformación

7.3. Análisis de Complejidad

7.3.1. Complejidad de Espacio

- **Grafo**: O(V + E) donde V = número de palabras, E = conexiones
- **BFS**: O(V) para estructuras auxiliares (cola, visitado, padres)
- **Total**: O(V + E) complejidad espacial

7.3.2. Complejidad de Tiempo

- **Construcción del grafo**: $O(V^2 \cdot L)$ donde L = longitud palabra (3)
- **BFS**: O(V + E) caso peor
- **Reconstrucción de camino**: O(V) caso peor

Para el conjunto típico de 5000 palabras de 3-5 letras, estos tiempos son negligibles.

Capítulo 8

Conclusiones

8.1. Logros del Proyecto

1. Implementación exitosa de arquitectura serverless completa en AWS
2. Integración efectiva de Spring Boot con AWS Lambda
3. Desarrollo frontend moderno con React y Vite
4. Resolución de problemas técnicos complejos durante despliegue
5. Documentación técnica completa del proceso y decisiones arquitectónicas
6. Aplicación en producción con alta disponibilidad y rendimiento
7. Pipeline de CI/CD completamente automatizado

8.2. Lecciones Aprendidas

- La compatibilidad de versiones entre API Gateway y handlers Lambda es crítica
- SnapStart ofrece mejoras significativas en tiempos de cold start para Java
- El caché de CloudFront requiere estrategias cuidadosas de invalidación
- SAM simplifica considerablemente el despliegue de aplicaciones serverless
- El monitoreo temprano con CloudWatch es esencial para debugging
- La documentación del código mejora significativamente el mantenimiento

8.3. Trabajo Futuro

Posibles extensiones del proyecto:

1. **Base de datos**: Migrar desde grafo en memoria a Amazon Neptune
2. **Autenticación**: Implementar AWS Cognito para gestión de usuarios
3. **Caché**: Añadir ElastiCache Redis para resultados frecuentes

4. **Monitoreo**: Configurar CloudWatch Dashboards y alarmas
5. **Análisis**: Implementar tracking de eventos con CloudWatch Analytics
6. **Multiidioma**: Soporte para español y otros idiomas
7. **Algoritmos**: Implementar Dijkstra, A* para diferentes métricas
8. **Escalabilidad**: Soporte para diccionarios más grandes (6-7 letras)

Referencias

1. AWS Lambda Documentation: `https://docs.aws.amazon.com/lambda/`
2. Spring Boot AWS Serverless: `https://github.com/awslabs/aws-serverless-java-container`
3. AWS SAM Documentation: `https://docs.aws.amazon.com/serverless-application-model/`
4. React Documentation: `https://react.dev/`
5. Vite Documentation: `https://vitejs.dev/`
6. AWS CloudFront: `https://docs.aws.amazon.com/cloudfront/`
7. Graph Theory - Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*

Apéndice A

Configuración del Template SAM

Fragmento clave de `template.yaml`:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Globals:
  Function:
    Timeout: 30
    MemorySize: 2048
    Runtime: java17
    Environment:
      Variables:
        JAVA_TOOL_OPTIONS: >
          -XX:+TieredCompilation
          -XX:TieredStopAtLevel=1

Resources:
  WordMapsFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: wordmaps-backend/
      Handler: com.wordmaps.config.LambdaHandler::handleRequest
      SnapStart:
        ApplyOn: PublishedVersions
      Events:
        ApiEvents:
          Type: HttpApi
          Properties:
            PayloadFormatVersion: '2.0'
            Path: /{proxy+}
            Method: ANY
            ApiId: !Ref WordMapsApi

  WordMapsApi:
    Type: AWS::Serverless::HttpApi
    Properties:
      CorsConfiguration:
        AllowMethods:
```

```
          - GET
          - POST
          - PUT
          - DELETE
        AllowHeaders:
          - '*'
        AllowOrigins:
          - '*'

  FrontendBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: wordmaps-frontendbucket
      PublicAccessBlockConfiguration:
        BlockPublicAcls: true
        BlockPublicPolicy: true
        IgnorePublicAcls: true
        RestrictPublicBuckets: true

  CloudFrontDistribution:
    Type: AWS::CloudFront::Distribution
    Properties:
      DistributionConfig:
        Enabled: true
        DefaultRootObject: index.html
        Origins:
          - DomainName: !GetAtt FrontendBucket.RegionalDomainName
            Id: S3Origin
            S3OriginConfig:
              OriginAccessIdentity: !Sub
                'origin-access-identity/cloudfront/${OAI}'
        DefaultCacheBehavior:
          ViewerProtocolPolicy: redirect-to-https
          AllowedMethods:
            - GET
            - HEAD
            - OPTIONS
          CachedMethods:
            - GET
            - HEAD
          TargetOriginId: S3Origin
          ForwardedValues:
            QueryString: false
            Cookies:
              Forward: none

  OAI:
    Type: AWS::CloudFront::CloudFrontOriginAccessIdentity
    Properties:
      CloudFrontOriginAccessIdentityConfig:
        Comment: OAI for WordMaps Frontend
```

Listing A.1: Configuración SAM completa

Apéndice B

URLs del Proyecto

- **Frontend**: `https://d2euump6bzpha5.cloudfront.net`
- **API Backend**: `https://e23y9088lc.execute-api.us-east-1.amazonaws.com/api`
- **Repositorio GitHub**: `https://github.com/JuanLopezdeHierro/wordmaps_project`
- **Documentación deepwiki**: `https://deepwiki.com/JuanLopezdeHierro/wordmaps_project`

Apéndice C

Estructura del Proyecto

```
wordmaps_project/
        template.yaml               # SAM/CloudFormation template
        samconfig.toml             # SAM deployment configuration
        .github/
              workflows/
                  deploy.yml         # GitHub Actions CI/CD
        wordmaps-backend/
              pom.xml
              src/
                    main/java/com/wordmaps/
                        WordMapsApplication.java
                        config/
                              LambdaHandler.java
                        controller/
                              RouteController.java
                        service/
                              RouteFinderService.java
                              GraphService.java
                              ClusterService.java
                        model/
                            Route.java
                            Transformation.java
              target/
                  wordmaps-backend-1.0-SNAPSHOT.jar
        wordmaps-frontend/
            package.json
            vite.config.js
            tailwind.config.js
            src/
                App.jsx
                services/
                      api.js
                components/
                      SearchForm.jsx
                      GraphVisualization.jsx
```

```
                Stats.jsx
            dist/                       # Build output
                index.html
                assets/
                ...
```

Listing C.1: Estructura de directorios del proyecto