Universidad Politécnica de Valencia

Máster Universitario en Computación Paralela y Distribuida

Desarrollo de un sistema de consulta de información sobre productos, con soporte de un servicio cloud.

Trabajo fin de máster

Autor: Vicente Jesús Bas Abad
Tutor: Jordi Bataller Mascarell

28 de septiembre de 2014

Agradecimientos

Me gustaría aprovechar estas primeras lineas para mostrar mi más profundo agradecimiento a todas aquellas personas que han hecho posible este trabajo.

En primer lugar agradecer a Jordi Bataller Mascarell, tutor de este trabajo fin de máster, su comprensión y ayuda durante el desarrollo.

De igual modo, agradecer a todos los profesores del máster su tiempo y esfuerzo a lo largo del curso, transmitiéndonos todos esos conocimientos sin los que esto no hubiese sido posible.

Por último agradecer a Germán Moltó, el GRyCAP y Amazon, la aportación de recursos sobre los que se han realizado las pruebas y el análisis de la arquitectura en la nube.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

En una sociedad en la que cada vez estamos más conectados, se hace evidente el uso de una arquitectura estándar con la que podamos comunicar la gran variedad de dispositivos móviles, portátiles o incluso "wereables" que inundan nuestros bolsillos. De esta necesidad nacen los servicios web, consiguiendo con ellos una interoperabilidad universal al estar basados en protocolos estándar. Permitiendo de este modo, con una cantidad mínima de infraestructura, integrar aplicaciones de manera flexible.

Por otra parte, destacar la necesidad de un sistema fiable y seguro. La interrupción del servicio en una aplicación web puede suponer perdidas millonarias. Para que esto no suceda, existen herramientas que ofrecen la posibilidad de diseñar y crear aplicaciones distribuidas altamente escalables, replicadas y consistentes. Un buen ejemplo de ello, es la infraestructura diseñada para la candidatura de Obama en las elecciones presidenciales de los Estados Unidos de 2012[1].

1.2. Objetivos y contexto

El objetivo de este trabajo final de máster es implementar un servicio web que proporcione un catalogo de productos y la información sobre los mismos.

Aunque podríamos implementar el sistema de forma abstracta, nos centraremos en un tema especifico: las alergias alimentarias. Ello facilitará la comprensión del sistema y proporcionará una aplicabilidad inmediata a nues-

tro desarrollo. En concreto, desarrollaremos un repositorio de alimentos que producen alergias alimentarias que pueda ser consultado desde cualquier dispositivo. Este servicio proporciona a sus usuarios información crucial, a la hora de hacer la compra, para saber si puede o no consumir un determinado producto. Su uso debe ser tan sencillo como escanear el código de barras en el envase e inmediatamente obtener la información pertinente.

La aplicación tendrá una arquitectura cliente-servidor. Estos interaccionan en base a una interfaz REST (ver sección 2.4). El cliente que desarrollaremos como demostración, será una aplicación en Android capaz de interactuar con el sistema de consulta. Por su parte, para desarrollar el servidor, pondremos en práctica muchos de los conocimiento adquiridos durante la realización del máster. Mas específicamente en las asignaturas de diseño de arquitecturas en la nube:

- Cloud Computing: Tecnologías y Arquitectura de Servicios
- Infraestructuras Avanzadas en Cloud
- Modelos de Programación en Cloud
- Programación en Sistemás Cloud
- Seguridad en Sistemás Distribuidos
- Rendimiento y Escalabilidad

1.3. Organización de esta memoria

Veamos ahora la organización de este documento. Dividido en 7 capítulos, cada uno de ellos aportará nuevos conocimientos sobre el trabajo, utilizando como base los anteriores.

Capítulo 1: Capitulo actual. Introducción.

Capítulo 2: Realizará una revisión de las tecnologías utilizadas. Detallaremos cuales han sido y que funcionalidad aporta cada una al proyecto.

Capítulo 3: Enumeración de los requerimientos del sistema.

Capítulo 4: Explicación del diseño de la arquitectura cloud y software.

Capítulo 5: Aspectos más importantes de la implantación. Que pasos se deben tomar para poder desplegar este tipo de aplicaciones.

Capítulo 6: Rendimiento del sistema. Estudio mediante gráficas del comportamiento de este ante un incremento en el número de peticiones por segundo.

Capítulo 7: Conclusiones y posibles trabajos futuros.

Capítulo 2

Tecnologías

2.1. AWS (Amazon Web Services)

AWS no es mas que la colección de servicios de computación y almacenamiento en la nube de Amazon. Permite desplegar grandes infraestructuras con aprovisionamiento dinámico de recursos sin necesidad de inversión inicial. Los pagos se hacen por el tiempo de uso de cada recurso y varían según la región en la que despleguemos la infraestructura. Amazon es el pionero en cloud computing desde 2006, ofreciendo:

- Bajo coste (sus precios bajan varias veces cada año) .
- Agilidad y elasticidad (permite modificar la cantidad de recursos sin apenas tiempos de espera).
- Seguridad.
- SLA (Acuerdo de nivel de servicio ver en [15]) acorde a las necesidades del cliente.

A continuación vamos a enumerar los servicios que hemos utilizado durante el trabajo.

2.1.1. EC2 (Elastic Compute Cloud)

EC2 es el servicio de Amazon que proporciona los recursos de computación en la nube. Para ello, nos permite desplegar maquinas virtuales a

Tipo	vCPU	ECU	Mem.(GB)	Alm.(GB)	Precio
t2.micro	1	Variable	1	Solo EBS	0.013
t2.small	1	Variable	2	Solo EBS	0.026
t2.medium	2	Variable	4	Solo EBS	0.052
m3.medium	1	3	3.75	1 x 4 SSD	0.070
m3.large	2	6.5	7.5	1 x 32 SSD	0.140
m3.xlarge	4	13	15	2 x 40 SSD	0.280
m3.2xlarge	8	26	30	2 x 80 SSD	0.560

Cuadro 2.1: Tabla de precios EC2 Linux en dolares por hora.

través de imágenes. Estas imágenes, también llamadas AMI pueden ser elegidas de una lista de imágenes predefinidas o pueden ser creadas por el usuario. Una vez esta imagen está funcionando como maquina virtual, Amazon se refiere a ella como instancia.

Instancia

Al crear cada una de estas instancias, podemos elegir su capacidad de computo. Esta viene dada por la cantidad de ECUs con la que despleguemos la instancia.

Una ECU proporciona aproximadamente la capacidad de calculo de una cpu con un unico procesador a Xeon de 2007 entre 1 y 1.2Ghz.

Para desplegar una de estas instancias, necesitamos:

- Seleccionar la imagen AMI a ejecutar.
- Configurar la seguridad (Crear un grupo de seguridad indicando los puertos en los que será accesible la instancia).
- Seleccionar el tipo de instancia.
- Determinar en que localizaciones se ejecutara la instancia.
- Pagar únicamente por los recursos que consumamos.

2.1.2. Elastic Load Balancing

En una aplicación cloud, el escalado, permite no solo aumentar la capacidad de cada instancia, sino también aumentar el numero de instancias, esto

es lo que se conoce como escalado horizontal. Para ello, necesitamos alguien que se encargue de distribuir la carga entre las instancias. Esta es la función que realiza Elastic Load Balancing.

Distribuye la carga de forma equitativa entre las diferentes zonas de disponibilidad. Las zonas de disponibilidad son ejecuciones independientes en infraestructura. Ayudan de este modo a conseguir alta disponibilidad. Es interesante que cada zona, tenga un numero similar de instancias. Si solo tenemos una zona esto no es un problema. Para distribuir la carga utiliza un algoritmo similar a round robin.

Las ventajas que nos aporta ELB son:

- Disponibilidad: Se consiguen niveles mas altos de tolerancia a fallos, ya que ELB garantiza que solo las instancias con buena salud reciban trafico.
- Elasticidad: Al integrase con Auto Scaling (del que hablaremos a continuación) permite que la capacidad del back-end varíe de forma dinámica con el fin de satisfacer las necesidades de los distintos niveles de trafico.
- Seguridad: Permite crear una arquitectura de varias capas con balanceadores internos unidos a diferentes grupos de seguridad.
- Admite conexiones seguras a través de SSL.
- Es capaz de mantener la sesión de un usuario en una instancia (esta no es una practica muy recomendable a la hora de programar en la nube).

Como en todos los servicios integrados en AWS los precios van en relación al uso, según tiempo y cantidad de trafico. Considerando el caso típico de la region EE.UU Este.

- 0.025 USD por hora (u hora parcial) de Elastic Load Balancer.
- 0.008 USD por GB de datos procesados por Elastic Load Balancer.

2.1.3. Auto Scaling

Auto Scaling permite escalar automáticamente la capacidad de EC2, para aumentarla o reducirla, de acuerdo a unas reglar definidas por el usuario.

Permite establecer métricas a seguir para el escalado. Si hay alguna instancia en mal estado o inaccesible la sustituye por otra. Tiene un servicio de notificaciones para avisar al usuario, pues desde que surgió la computación en la nube, apareció un nuevo tipo de ataque basado en obligar a la aplicación a aumentar su capacidad de computo para que aumente el coste.

2.1.4. DynamoDB

Servicio de bases de datos NoSQL de Amazon, muy rápido. Todo su contenido esta almacenado sobre discos SSD. Garantiza una latencia de milisegundos. También cabe destacar la API, ya que DynamoDB la ofrece para la mayoría de lenguajes de programación actuales, entre ellos Node.JS como mas adelante veremos.

DynamoDB permite especificar el rendimiento para cada tabla de forma independiente. La capacidad tanto en lo referente a volumen de datos como a rendimiento es ilimitada. De esta forma facilita el trabajo al desarrollador. Olvidándose de crear particiones (sharding) en la base de datos para aumentar el rendimiento o la capacidad. Solo nos tenemos que centrar en la implementación.

El precio para la region EE.UU Este es de:

- Rendimiento de escritura: 0.0065 USD por hora por cada 10 unidades de capacidad de escritura (capacidad suficiente para realizar hasta 36.000 escrituras por hora).

- Rendimiento de lectura: 0.0065 USD por hora por cada 50 unidades de capacidad de lectura (capacidad suficiente para realizar hasta 180.000 lecturas de coherencia alta o 360.000 de lecturas de coherencia eventual, por hora).

La forma de calcular las necesidades en nuestras tablas se mide con las unidades de lectura y escritura. Una unidad de capacidad de escritura permite realizar una escritura por segundo, de elementos con tamaño de un máximo de 1kb. En el caso de las unidades de lectura, permite realizar una lectura altamente coherente (o dos con coherencia eventual) de elementos con hasta 4kb de tamaño.

2.1.5. CloudFormation

CloudFormation ofrece de forma gratuita un método sencillo para crear la infraestructura cloud. Permite escribir plantillas en formato JSON, donde se describen los recursos AWS a desplegar, sus características y dependencias. Una vez están en funcionamiento los recursos, pueden ser modificados de forma normal.

Principales características de CloudFormation.

- Archivo de texto escrito en JSON donde se describe la infraestructura.
- Permite gestionar relaciones entre grupos EC2, Elastic Load Balancing y mas.
- Las plantillas se puede reutilizar modificando valores para las diferentes implementaciones de la aplicación.
- Ofrece la posibilidad de salidas de los resultados. Por ejemplo, cuando se incluye una instancia devolver el nombre y la dirección ip de esta.
- La identificación por nombres lógicos evita conflictos de nomenclatura entre diferentes recursos.
- Compatible con diferentes lenguajes de programación mediante APIs.

2.2. Node.JS

Segun wikipedia [4].

> Node.JS es un entorno de programación en la capa del servidor basado en el lenguaje de programación Javascript, con I/O de datos en una arquitectura orientada a eventos y basado en el motor Javascript V8. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web.

El principal objetivo de Node.JS [5] es “proporcionar una manera fácil para construir programas de red escalables”. En los actuales servidores la utilización de memoria RAM y CPU es elevada. Limitando el numero de clientes por servidor, y aumentando el precio final de las aplicaciones a gran escala.

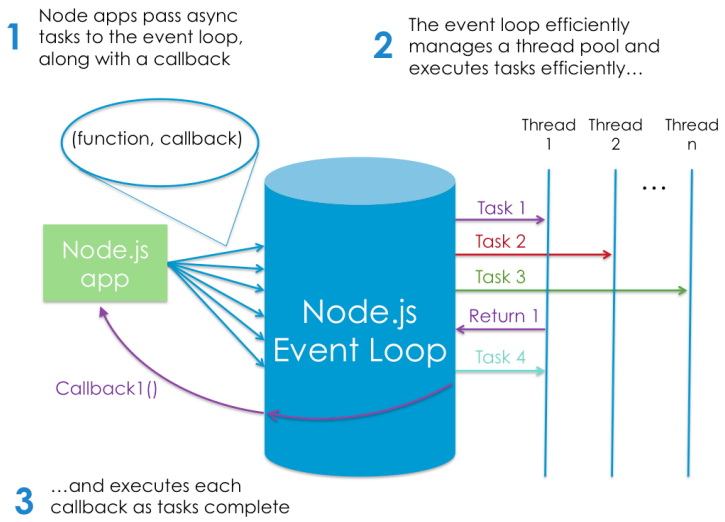

Figura 2.1: Ciclo de vida de los eventos en Node.JS [3]

Node utiliza javascript en la parte del servidor basándose en eventos que se ejecutan de manera asíncrona. Estas ejecuciones asíncronas son engorrosas si se gestionan en diferentes hilos. Para solucionar este problema Node mantiene un event loop que gestiona todas las operaciones asíncronas. Cuando necesita realizar una operación con bloqueo (por ejemplo operaciones con archivos) envía una tarea asíncrona al event loop, junto con un callback y luego continúa. El event loop realiza un seguimiento de la operación asíncrona, y ejecuta el callback cuando esta finaliza.

Otra parte interesante de Node.JS es el gran apoyo de la comunidad y sus repositorios. En poco tiempo han aparecido una gran cantidad de módulos con diferentes fines que podemos utilizar en nuestra aplicación con un simple import.

Por ello Node.JS es ideal para servidores. En casos concretos con linkedIn al migrar la plataforma desde ruby on rails a Node.JS, disminuyeron el numero de servidores necesarios de 30 a 3, algo que económicamente aporta un gran beneficio.

2.2.1. Express

Express[6] es un framework para el desarrollo de aplicaciones en Node.JS. Inspirado en sinatra [7] permite enrutar peticiones HTTP (POST,GET,PUT,DELETE) hacia el código que deseemos ejecutar.

2.2.2. Passport

Passport [8] es un middleware de autenticación para Node. Esta diseñado para el propósito de autenticar peticiones. Se integra en el código de las peticiones, permitiendo inicios de sesión al proporcionarle un usuario y contraseña, o incluso autenticación OAuth con diferentes proveedores.

Dentro de passport, podemos elegir diferentes estrategias de autenticación, permitiéndonos escoger entre mantener sesiones persistentes o desactivar las sesiones.

2.2.3. Crypto

Crypto es el modulo de Node.JS encargado de la encriptación. Ofrece una colección de algoritmos de cifrado.

2.2.4. JWT-Simple

JWT(JSON Web Token), es un modulo Node.JS utilizado para encriptar y desencriptar cadenas. Por defecto utiliza un algoritmo HS256 pero se puede configurar para utilizar HS384, HS512 y RS256.

2.3. HTTPS

HTTPS es la versión securizada del Hypertext Transfer Protocol. Con ella se pretende evitar que un tercero pueda escuchar la conversación entre cliente y servidor.

Utiliza un sistema de cifrado basado en SSL o TLS para crear un canal seguro, consiguiendo evitar que si alguien capta la transmisión pueda entender que estamos enviando. El puerto por defecto es el 443. Para poder utilizar este tipo de transmisión segura, se necesita un certificado de clave publica firmado por una autoridad certificadora. De este modo, el cliente puede comprobar quien es el servidor y garantizar que nadie lo está suplantando.

Estos certificados de clave publica pueden ser revocados si han expirado o si hay evidencias de que su clave privada ha sido comprometida.

2.3.1. Certificados

Utilizan un método de cifrado asimétrico, se emplean dos claves, una publica y una privada. Cuando el cliente cifra el mensaje con el certificado (clave publica firmada por una autoridad certificadora), el único que será capaz de descifrarlo es el dueño de ese certificado. Con ello se consigue que si alguien intercepta una comunicación con datos sensibles no sea capaz de leerla.

2.3.2. TLS/SSL

Son sistemas de cifrado para proporcionar conexiones seguras sobre Internet en arquitecturas cliente servidor. Puede utilizar los algoritmos de cifrado RC2, RC4, IDEA, DES, Triple DES y AES. Sus fases son:

1. El cliente envía una petición de sesión segura.
2. El servidor responde con el certificado de clave publica.
3. El cliente comprueba si el certificado está en una CA.
4. El cliente genera una clave simétrica aleatoria y la cifra utilizando la clave publica del servidor.
5. Ambos conocen la clave simétrica y cifran/descifran los mensajes mientras dure la sesión.

2.4. REST

REST es un estilo de arquitectura o patrón de diseño basado en los estándares HTTP y URL. Existen recursos que deben tener un identificador global. Se crea un recurso para cada servicio y cada uno de estos recursos se identifica con una URL.

Soporta diferentes tipos de datos/MIME types.

- HTML
- TEXT

- JSON
- XML
- etc.

Otra de las principales ventajas de REST a la hora de construir una arquitectura escalable es que las comunicaciones son sin estado (stateless). De este modo se consigue no tener dependencia con el servidor.

Un servicio REST correctamente diseñado debe utilizar los códigos de error y las excepciones HTTP.

- 1xx : iformación
- 2xx : éxito
- 3xx : redirección
- 4xx : error cliente
- 5xx : error servidor

Dentro de REST, existen dos tipos de operaciones, seguras y inseguras.

Las operaciones seguras son aquellas no modifican el estado de los recursos, suelen ser operaciones de consulta, por muchas veces que se reenvíen no cambia el resultado.

Las operaciones inseguras son aquellas que tras ejecutarlas se modifica el estado de algún recurso, operaciones de actualización, creación o borrado. Para las operaciones inseguras se debe controlar que sucede en caso de reenvío.

Por otro lado, la idempotencia es la capacidad de que una operación siempre del mismo resultado se ejecute una o n veces. En las operaciones idempotentes aunque haya reenvios no cambia el resultado. Por el contrario, en una operación no idempotente si que varía.

Los servicios REST proporcionan soporte a todas las comunicaciones web usadas en la API HTTP. De este modo, las operaciones serían las siguientes:

- GET
    - Devuelve información (Retrieve).
    - Operación segura e idempotente.

- POST
  - Añade información (Create).
  - Operación insegura y no idempotente.
- PUT
  - Actualiza la información (Update).
  - Operación insegura e idempotente.
- DELETE
  - Borra información (Delete).
  - Operación insegura e idempotente.
- HEAD*
  - Información resumida.
  - Operación segura e idempotente.
- OPTIONS*
  - Operaciones invocables.
  - Operación segura e idempotente.

*Estas operaciones no se suelen utilizar, pero es interesante conocerlas.

2.5. Android

Android es un sistema operativo para dispositivos móviles basado en linux. El desarrollo de aplicaciones para Android se realiza en una variante de Java llamada Dalvik. El sistema, a través del SDK proporciona las interfaces para que los desarrolladores tengan acceso a las funciones del teléfono (GPS,cámara, agenda, etc...).

Para el desarrollo de aplicaciones existe un Android SDK (Android Software Development Kit) especifico, aunque también esta disponible un kit de desarrollo nativo. Como IDE existen varias opciones, las mas conocidas son Eclipse y Android studio.

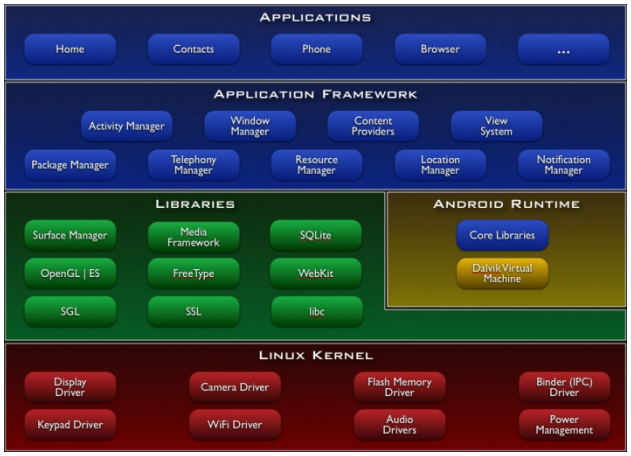

Figura 2.2: Arquitectura sistema operativo Android [9]

2.5.1. Volley

Volley es una librería HTTP para Android que facilita las comunicaciones web. Los principales beneficios son:

- Programación (agenda) automática para las solicitudes de red.
- Soporta conexiones múltiples.
- Cache coherente.
- Priorización de peticiones.
- Facilidad de personalización (reintentos).
- Asíncrono.

Entre sus principales usos destacan las operaciones de tipo RPC para completar huecos en interfaces de usuario. Es totalmente compatible con diferentes tipos como cadenas de texto o JSON.

2.5.2. ZXing

ZXing es una librería de procesado de imágenes multi-formato, tanto en 1D como 2D. Distribuido bajo una licencia de codigo abierto. Es capaz de reconocer los formatos de código de barras UPC-A, UPC-E, EAN-8, EAN-13,

Códigos 39, 93, 128, ITF, Codabar, RSS-14, Matriz de datos (Data Matrix), Aztec, PDF 417 y QR.

Funciona con cualquier SDK Android superior a la versión 4.0 y entre sus cualidades destacar la facilidad para modificarlo y configurar al tipo de código que se quiera escanear.

Capítulo 3

Análisis de requerimientos

En este trabajo vamos a cubrir la necesidad de identificación de alérgenos en alimentos. Permitiendo de esta forma a las personas con diferentes alergias alimentarias, identificar de modo rápido con su teléfono inteligente Android, si pueden ingerir determinado alimento. Para ello tan solo tendrán que escanear el código de barras de dicho alimento utilizando la cámara de su teléfono.

El sistema podrá llevar a cabo cualquier operación de registro. Tanto de usuarios como de alimentos, ingrediente y alérgenos. Dicho de otro modo, podrá realizar operaciones de alta, baja o modificaciones en cada uno de los recursos anteriormente citados. También deberá ofrecer un sistema de inicio de sesión sobre los usuarios. De modo que en cualquiera de las anteriores acciones, quede registrado el nombre del usuario que ha realizado la operación.

Para poder llevar a cabo estas acciones se necesitan los siguientes datos:

- Usuario: Para el alta o modificación de usuario se proporcionará una clave a modo de seudónimo, además del nombre real, correo electrónico y una contraseña.

- Alimento: Para el alta o modificación de un alimento deberemos proporcionar un código de barras ean-13 como clave, un nombre y los ingredientes que lo componen. En el caso que no existan dichos ingredientes, deberán ser dados de alta de forma previa al alimento.

- Ingrediente: Para el alta o modificación de un ingrediente, indicaremos el nombre como clave y los alérgenos. Como en el caso de los alimentos,

si estos alérgenos no existen. Deberán ser dados de alta de modo previo al ingrediente.

- Alérgeno: Para el alta o modificación de un alérgeno, se debe proporcionar el nombre como clave, los síntomás y el tratamiento.

Veamos ahora para cada tipo de usuario, en que posible escenario puede tomar parte.

Usuario no registrado.

- Registro, permite registrar un nuevo usuario.
- Consultar recurso, devuelve información del recurso al realizar una petición sobre la clave de este.

Usuario registrado.

- Consultar recurso, devuelve información del recurso al realizar una petición sobre la clave de este.
- Modificar usuario propio, tras iniciar sesión, permite modificar algún campo del usuario previamente registrado.
- Insertar nuevo recurso, permite añadir un nuevo recursos al sistema.
- Modificar recurso, permite modificar la información de un recuso al realizar una petición de modificación sobre la clave primaria de este, se le enviarán los nuevos valores.
- Eliminar recurso, permite eliminar un recurso.

Quedando el diagrama de casos de uso como se puede ver en la figura 3.1.

Con los actores y los casos de uso ya definidos, veamos ahora los requisitos no funcionales.

Para el desarrollo del repositorio buscamos una aplicación que sea en primer lugar escalable (ver [11]), disponible (ver [13]) y consistente (ver [14]). Este último no será un requisito indispensable, pues en el planteamiento que vamos a realizar de la aplicación será suficiente con una consistencia débil (ver [12]).

Por el contrario, la escalabilidad es un punto muy importante. Será necesario que el sistema sea capaz de aprovisionar recursos de forma dinámica.

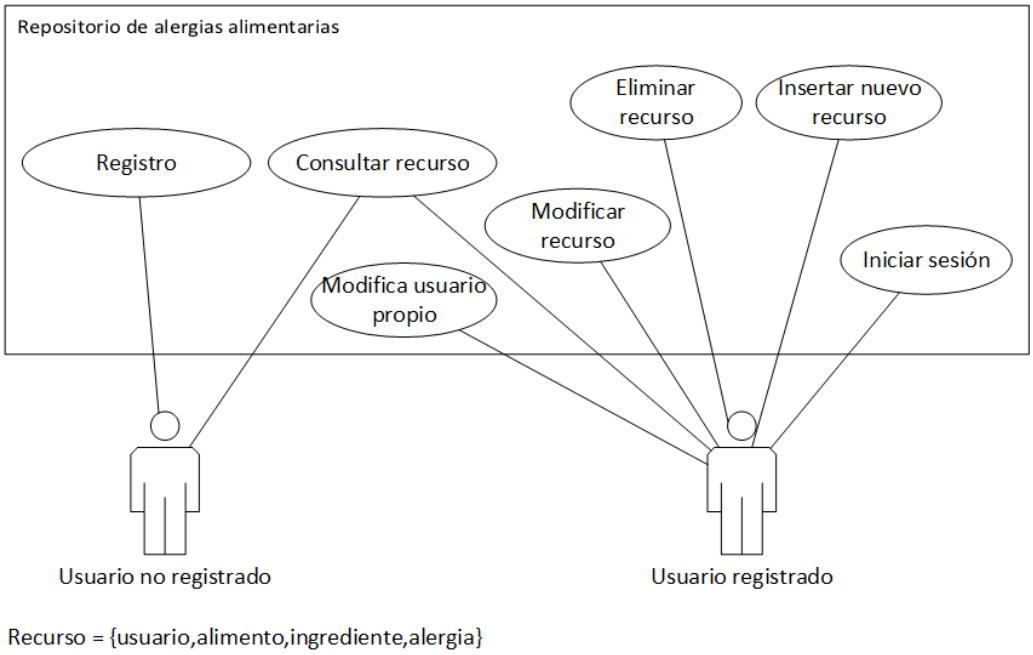

Figura 3.1: Diagrama de casos de uso

Garantizando así el correcto funcionamiento durante su tiempo de vida. Correcto funcionamiento entendemos por el modo donde el sistema sea capaz de responder a todas las peticiones de los clientes de forma correcta y sin un retraso anormal.

Referente a la disponibilidad, necesitaremos que el sistema esté en funcionamiento aunque se produzca algún fallo. Deberá ser capaz de auto comprobar el correcto funcionamiento y en caso de fallo tomar medidas para solucionarlo.

Otro requisito importante para un sistema de este tipo es la seguridad. Debe tener la capacidad de evitar que terceras personas lean la información de sus usuarios mediante ataques "man in the middle" (ver [16]) o que la puedan modificar.

Para finalizar, tendremos en cuenta la viabilidad económica. Que sea viable económicamente está ligado a la escalabilidad. Si conseguimos que el sistema no desaproveche recursos estaremos ahorrando dinero.

Por todas estas características se ha escogido montar la aplicación sobre una arquitectura cloud. Con ella evitaremos gastos iniciales, pues solo

pagaremos por lo que utilicemos en cada momento y podremos cumplir los requerimientos de escalabilidad y disponibilidad.

Capítulo 4

Diseño de la aplicación

Tras haber visto las tecnologías que vamos a utilizar, a continuación analizaremos el problema a resolver. En este capitulo veremos detenidamente el diseño de cada una de las partes que componen la aplicación y como interactúan entre sí. Para ello dividiremos el diseño en dos partes: infraestructura cloud y arquitectura software.

4.1. Diseño arquitectura cloud

Pasemos ahora a detallar la infraestructura cloud. Como vimos en la sección 2.1.1 dentro de AWS existe gran variedad de tipos de instancia. Estas tienen rendimientos y propósitos bastante dispares. Con el fin de buscar una instancia con rendimiento aceptable y un precio contenido nos hemos decidido por las del tipo m3.medium pues las instancias micro, aunque en determinados casos pueden ser suficientes, no son demásiado adecuadas debido a que no ofrecen un rendimiento constante, simplemente picos de CPU. Este es el caso de t1.micro y t2.micro.

Los dos puntos interesantes de este tipo de instancia son la memoria RAM, con una cantidad considerable si lo comparamos a las instancias basicas (0.6 vs 3.75) y el disco duro. Ofrecen un disco de estado solido de 4gb. Esto es

Tipo	vCPU	ECU	Memoria(GB)	Almacenamiento(GB)	Precio
m3.medium	1	3	3.75	1 x 4 SSD	0.070

Cuadro 4.1: Características y precio por hora de la instancia m3.medium.

muy útil por si queremos guardar un log en el sistema de ficheros, Ya que con estas características no tendremos problemás gracias a las velocidades de escritura en disco.

Recordemos, que por la definición del problema, el número de instancias debe ser variable para adaptar el rendimiento a las condiciones de carga. Para ello utilizaremos un grupo de auto escalado (sección 2.1.3). Este grupo, junto a una alarmás preestablecidas monitorizará la carga de nuestras instancias y tomará las decisiones oportunas para añadir o quitarlas instancias de modo que el sistema se adapte a las condiciones. El número máximo y mínimo de instancias es definido por el administrador del sistema, en nuestro caso estará comprendido entre 1 y 4. Las alarmás que controlan la creación o destrucción de instancias se regirán por unos porcentajes de uso de CPU, de modo que cuando todas las instancias del grupo de escalado superen el 70 % de CPU y el número de estas sean inferior a 4 se crea una nueva. En el caso contrario, si alguna instancia tiene un uso inferior al 20 % y hay más de 1 instancia en el grupo, esta se destruye.

No olvidemos también, que las instancias deberán estar asociadas a un grupo de seguridad que autorice el tráfico por un determinado puerto con los clientes. De este modo, la primera aproximación a la arquitectura queda como se puede ver en la figura 4.1.

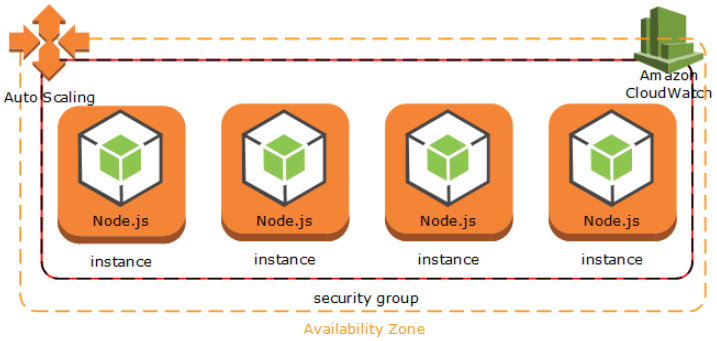

Figura 4.1: Primera parte arquitectura cloud

Ahora bien, con esta estructura las peticiones se deberían hacer de forma manual a cada una de las instancias. Para solucionar este problema, añadiremos un balanceador de carga (sección 2.1.2) que se encargará de recoger todas las peticiones y distribuirlas siguiendo un algoritmo de reparto similar a round robin entre las instancias que en ese momento tengamos en el gru-

po de auto escalado. Debemos tener en cuenta que si el servicio es seguro, este balanceador de carga también deberá serlo. Le indicaremos el protocolo, HTTPS en nuestro caso, y añadiremos el certificado firmado por una Autoridad Certificadora.

Para terminar, necesitaremos una base de datos donde guardar la información de los recursos. Como hemos visto en el apartado de tecnologías, vamos a utilizar dynamoDB (sección 2.1.4). Con DynamoDB cubriremos todas las necesidades de rendimiento. Entre sus puntos fuertes, destacar la alta velocidad de lectura al escoger un modelo de coherencia final. Esto significa que en las tres replicas donde se almacena cada tabla. Siempre llegara algún momento de la ejecución donde existirá coherencia entre los datos almacenados.

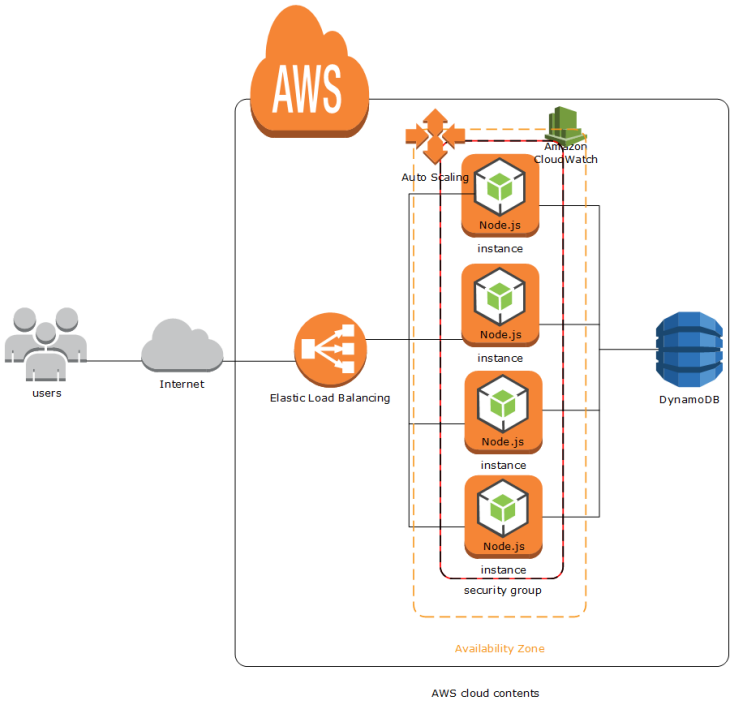

Figura 4.2: Arquitectura cloud completa con 4 instancias

Como ya se ha comentada anteriormente, el rendimiento viene dado por unidades de lectura y escritura. Para un primer cálculo aproximado de las unidades de lectura y escritura, supondremos que cada una de ellas tiene un tamaño de 1kb, el ratio de estas es de 100:1 y cada tabla tiene una

media de 1000 peticiones por segundo. De esta forma y siguiendo las formulas disponibles en el FAQ de dynamoDB.

Unidades de capacidad necesarias para escrituras = Número de escrituras de elementos por segundo x tamaño del elemento (redondeando hasta el KB más próximo).

Unidades de capacidad necesarias para lecturas = Número de lecturas de elementos por segundo x tamaño del elemento (redondeando hasta el KB más próximo) / 2.

$UCE = 10 * 1 = 10$

$UCL = \frac{1000*1}{2} = 500$

Con todo lo comentado hasta ahora ya tenemos completa la arquitectura cloud de un modo similar a la figura 4.2.

4.2. Diseño arquitectura software

4.2.1. Servicio REST

A la hora de diseñar un servicio REST debemos planificar las interacciones del usuario con cada uno de los recursos, así como el tipo de peticiones que acepta el recurso. En el caso de nuestro repositorio, los recursos principales son “user”, “aliment”,“ingredient” y “allergen”. A parte de estos existen unos recursos para permitir el registro de usuarios y el inicio de sesión llamados “signin” y “login”.

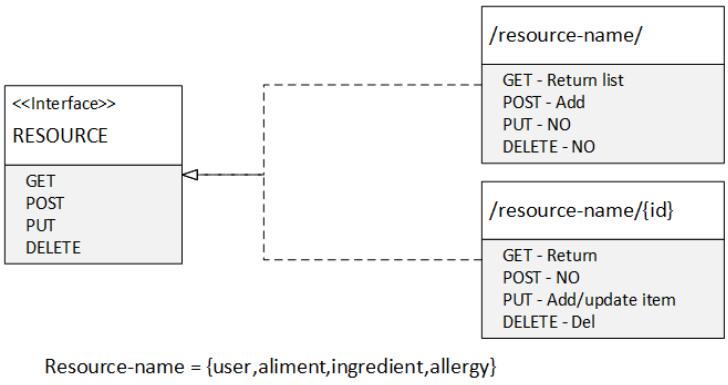

Figura 4.3: Peticiones en recursos REST

Cada recurso acepta peticiones del tipo GET, POST, PUT y DELETE,

tras recibir cualquiera de estas, la procesa y una vez tiene el resultado envía una respuesta al cliente con un JSON en el que se le devuelve el código del resultado y el mensaje.

Un ejemplo de una petición incorrecta:

- Payload: {"CODE":"FALSE","MSG":"DB ERROR"}
- Status: 500

Un ejemplo de una petición correcta:

- Payload: {"CODE":"1001","MSG":"TOKENDELOGINDEUSUARIO"}
- Status: 200

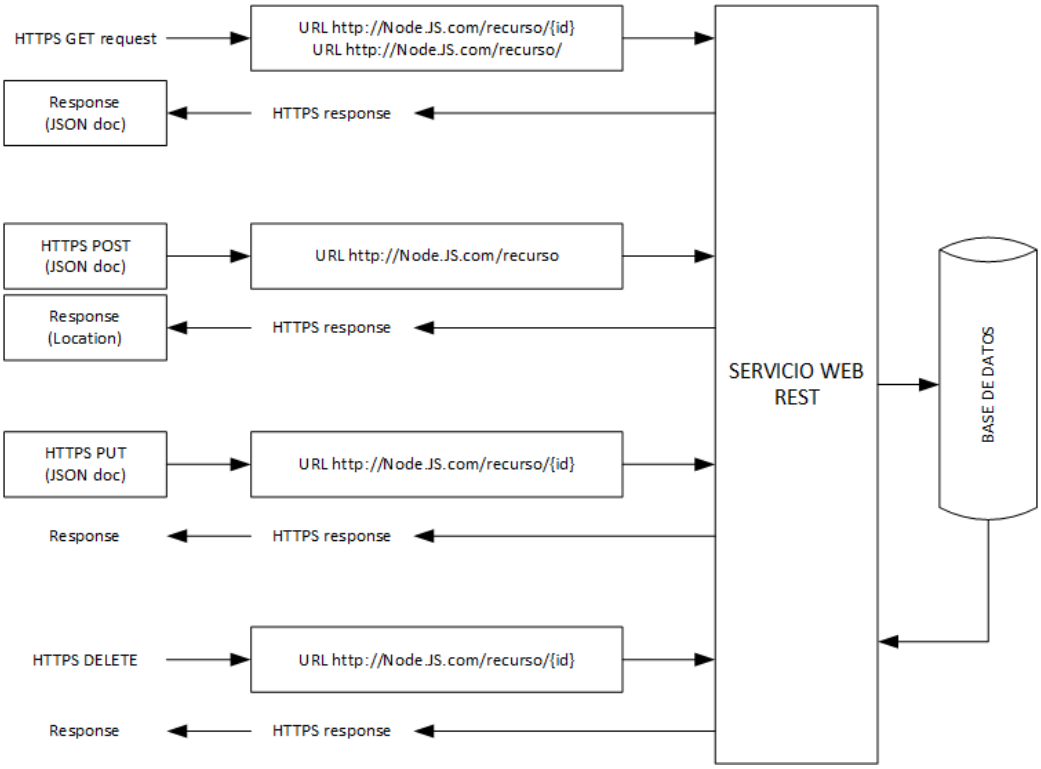

Figura 4.4: Arquitectura REST

El envío de datos al servicio se realiza mediante el payload con el "Content-Type" en formato "application/JSON". También cabe comentar que hay

cierto tipo de peticiones POST y PUT que solo pueden realizar usuarios registrados. Para ellas se debe añadir en la cabecera de la petición (HEAD) el token de autenticación devuelto por el recurso "login". Quedando este como access_token: eyJ0eXAiOiJKV1Q...

4.2.2. Allergy finder (Android)

La aplicación Android esta compuesta por un interface gráfico del tipo "Navigation drawer activity". Desde este menú de navegación se puede acceder a las diferentes acciones. Estas son:

- Escanear código de barras (pantalla de inicio).
- Registro y inicio de sesión de usuarios.
- Añadir alérgeno.
- Añadir ingrediente.
- Añadir alimento.
- Opciones (desde las opciones añadiremos a que alérgenos tenemos hipersensibilidad).

Al ejecutar cualquiera de estas acciones, se invocará una llamada remota al servicio REST. Cuando recibamos la respuesta de esta llamada remota, de modo asíncrono un método gestionará la respuesta según el código enviado.

Capítulo 5

Implantación

El capítulo que nos ocupa llevará al lector a comprender como crear de forma rapida una arquitetura cloud. En él explicaremos las diferencias entre hacerlo manualmente y automatizar el despliegue utilizando CloudFormation (anexo **??**).

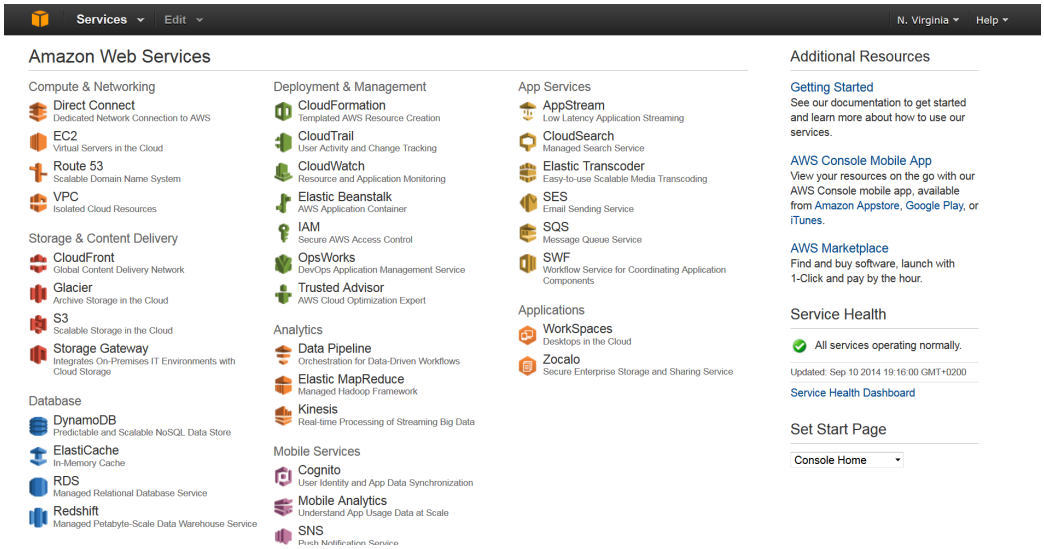

Figura 5.1: Pantalla principal de AWS

5.1. Empezando con AWS

El primer paso para poder desplegar una infraestructura en la nube sobre AWS es registrarnos. Para ello nos dirigimos a la web principal de [2, AWS] y accedemos al correspondiente apartado de inscripción. Tras completar los diferentes formularios con nuestros datos e insertar el número de la tarjeta de crédito, accederemos a la consola de administración (figura 5.1). Desde esta pagina principal tendremos acceso a los diferentes servicios que componen AWS.

El próximo pasó será dirigirnos al servicio EC2, donde desde la opción "Key Pairs" crearemos un par de claves con los que podremos conectarnos a nuestras instancias. Con las claves creadas, podemos empezar a trabajar. Siguiendo en el servicio EC2 nos dirigimos a "Auto Scaling Groups" y pulsamos sobre el botón "Create Auto Scaling group". De este modo crearemos un grupo de auto escalado en la region que nos encontremos. Para saber en que región estamos, podemos observar la parte superior derecha de la pagina web como sucede en la figura 5.1. En este caso estamos en N. Virginia.

Definiremos ahora una plantilla (template) para el lanzamiento de nuevas instancias. Cada plantilla consta necesita los siguiente parámetros:.

1. AMI. Como se comente en la sección 2.1.1. Las AMI son imágenes predefinidas con diferentes sistemás operativos. Por lo tanto escogeremos en sistema operativo y la arquitectura deseados. En nuestro caso hemos optado por un Ubuntu de 64 bits.

2. Tipo de instancia. Según los requerimientos en capacidad de cálculo. Para nuestra arquitectura se ha escogido una instancia de propósito general m3.medium (sección 4.1).

3. Configuración.

4. Almacenamiento. Llegados a este punto seleccionaremos el almacenamiento de cada instancia. En nuestro caso continuaremos con el almacenamiento por defecto de las instancias m3.medium (8+4gb).

5. Grupo de seguridad. Configuraremos los puertos a través de los que podrán pasar peticiones a la instancia. Seleccionaremos el protocolo SSH, y además añadiremos los puertos 8080 (para conexiones sin cifrar) y 44401 (puerto utilizado por nuestro repositorio para las conexiones cifradas).

Con la plantilla correctamente definida podemos proceder a crear el grupo de auto escalado.

1. Elegiremos el nombre del grupo de auto escalado, con cuantas instancias arranca y las zonas de disponibilidad. Es aconsejable escoger al menos dos zonas para una aplicación robusta.

2. Configuración políticas de escalado. Seleccionaremos la opción "Use scaling policies to adjust the capacity of this group". Escogeremos el mínimo y máximo de instancias, en nuestro caso 1 y 4. Y procedemos a crear las alarmás pulsando sobre "Add new alarm". Llegados a este punto crearemos dos alarmás donde estableceremos el porcentaje de uso superior e inferior de la CPU (figura 5.2).

3. Podemos añadir notificaciones. De esta forma nos avisara por correo electrónico.

4. Por último comprobamos que todo sea correcto y creamos el grupo de auto escalado.

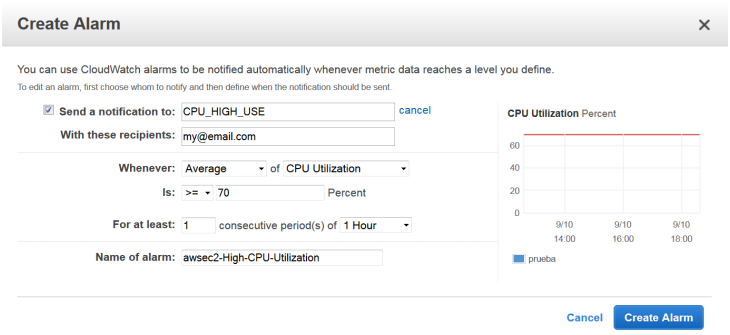

Figura 5.2: Creación de una nueva alarma. Porcentaje de uso superior en la CPU

Llegados a este punto, tendríamos una arquitectura similar a la vista en la figura 4.1. Para completarla nos falta añadir la base de datos y el balanceador de carga.

Siguiendo en la consola de EC2 escogeremos ahora "Load Balancers".

1. Tras pulsar en "Create Load Balancer", escogeremos el nombre, los protocolos y los puerto que queremos que balancee.

2. En la siguiente pantalla establecemos la configuración para el "Health check". Esta consiste en elegir un puerto y un protocolo contra el que el balanceador pueda hacer pings para comprobar si la aplicación esta funcionando.

3. Asignamos el grupo de seguridad que habíamos creado anteriormente.

4. Escogemos la instancia creada anteriormente de modo que quede dentro del balanceador de carga.

5. Comprobamos la correcta configuración y pulsamos sobre "create".

El último paso será añadir la base de datos. Volvemos a la consola de administración de AWS, pulsaremos sobre DynamoDB.

Tras acceder al menú de DynamoDB, comprobamos que la única acción que permite es crear tablas. De este modo, Amazon trata de conseguir que el desarrollador tenga las cosas lo más simple posible. Y es que con la elasticidad de las bases de datos NoSQL y la infraestructura creada por Amazon, únicamente nos hemos de preocupar de crear tablas. Con DynamoDB olvidamos problemás de falta de espacio o rendimiento. Solo nos hemos de centrar en el contenido.

Los pasos para crear una tabla son:

1. Teclear el nombre de la tabla y de la clave primaria. Esta puede ser de dos tipos, hash o hash and range. Elegiremos el primero si la clave primera solo esta compuesta por un elemento. Por el contrario elegimos el segundo si esta compuesta por dos elementos.

2. Si lo necesitamos, podemos añadir indices secundarios.

3. Establecemos la capacidad de rendimiento que debemos aprovisionar. Para ello introducimos el número de unidades de escritura y unidades de lectura. La formula para calcularas se ha visto en la sección 2.1.4.

4. En este paso añadiremos alarmás. Estas no son tan potentes como las vistas en EC2, ya que ahora mismo no permiten que escale automáticamente la base de datos. Pero no por ello dejan de ser interesantes, ya que si tenemos un aviso de número alto de peticiones, podemos acceder a la consola de Amazon y escalarla manualmente.

5. Pulsamos en "Create" y ya tenemos creada nuestra nueva tabla.

5.2. Aplicando CloudFormation

El método anterior puede ser validado para el despliegue de una aplicación ligera o como toma de contacto con AWS. Por el contrario, si queremos desplegar grandes aplicaciones donde puede coexistir diferentes tipos de instancia y cada una tenga una configuración diferente debemos utilizar CloudFormation.

Como se vio en la sección 2.1.5 CloudFormation permite la definición de toda la infraestructura mediante texto en formato estructurado (JSON).

Siguiendo como referencia el anexo **??** veremos que todas las partes comentadas en el apartado anterior se encuentran definidas en este fichero. En primer lugar se definen los parámetros de la aplicación. Estos se utilizan para items que se vayan a llamar en repetidas ocasiones, como puertos, unidades de lectura/escritura, etc...

El siguiente paso es definir los recursos. Se puede utilizar cualquiera de los servicios de AWS. Y para ello tan solo hemos de darle un nombre, elegir el tipo de recurso y definir sus propiedades.

Una vez la plantilla está completa, accedemos a la consola de administración de CoudFormation.

1. Al acceder al servicio pulsamos en “Create New Stack”.

2. Introducimos el nombre y pulsamos sobre Examinar para subir nuestra plantilla. Elegimos la plantilla en nuestro disco local y pulsamos en “Next”.

3. Comprobamos los parámetros, pulsamos “Next” y “Create” mientras sea necesario.

Tras completar estos pasos, esperaremos a que CloudFormation nos avise del despliegue correcto. En caso de error, recibiremos un mensaje con el fallo y destruirá toda la arquitectura. Por el contrario, si todo ha salido bien, veremos las salidas (outputs) que hayamos definido.

5.3. Android

La puesta en funcionamiento de la aplicación Android será tan sencilla como copiar el fichero APK al teléfono móvil e instalarla. Tras la instalación ya la podremos ejecutar e interactuar con el servicio REST.

Capítulo 6

Análisis

Analicemos ahora como se comporta el sistema con diferentes cantidades de carga. Estudiaremos las tres posibles causas de cuello de botella y que comportamiento tienen mientras se van incrementando el número de peticiones por segundo.

Para poder ejecutar el análisis se han simulado peticiones al servicio REST de una manera similar a como lo harían los usuarios que están accediendo a la aplicación. Para lograrlo, hemos aprovisionado 4 maquinas EC2 del mismo tipo que nuestras instancias (m3.medium). Estas maquinas realizan peticiones al balanceador de carga, que se encarga de repartir las peticiones entre las diferentes instancias servidor. Al mismo tiempo, cuando una instancia supera el porcentaje máximo de uso de CPU por un periodo de tiempo establecido, se crea una nueva instancia siempre que el número sea inferior a 4.

El modo de realizar las peticiones ha sido implementando un script (anexo **??**) que realiza conexiones utilizando CURL. Se ha respetado el porcentaje de conexiones, siguiendo así un ratio de 100 lecturas / 1 escritura.

La forma de interactuar del script con el recurso es realizando 100 lecturas y 1 escritura por segundo. Pasado un intervalo de 90 segundos el número de lecturas se incrementa en 100 hasta teóricamente llegar a 1000 por segundo.

Al mismo tiempo las instancias servidor están ejecutando la aplicación DSTAT en background y guardando en un fichero el porcentaje de uso de la CPU, cantidad de memoria RAM utilizada y utilización de la red. Para tener otra referencia, también hemos guardado las gráficas generadas por AWS CloudWatch. Permitiéndonos de esta forma contrastar resultados.

6.1. Porcentaje CPU

El primer parámetro que vamos a analizar es el porcentaje de uso en la CPU de cada instancia. Recordemos que en el instante de tiempo 0 solo hay una instancia y que estas van siendo añadidas según la carga del sistema.

Figura 6.1: Porcentaje uso de CPU

De este modo, como se puede observar en la figura 6.1, al principio solo tenemos una instancia funcionando. Esta recibe todo el tráfico que llega al balanceador. Su porcentaje de uso en la CPU aumenta rápidamente y tras saltar la alarma por alta carga en CPU se añade una nueva instancia al grupo de escalado.

Observese, como aproximadamente en el minuto 4 la segunda instancia esta activa, y aunque van aumentando el número de peticiones por segundo baja ligeramente el porcentaje de uso, ya que este pasa a estar repartido entre las dos instancias activas.

De nuevo en el minuto 14 y tras entrar la tercera instancia en juego observamos un comportamiento similar. Las tres instancias moderan su uso de procesador.

Llegados a este punto, observamos que se produce un periodo en el que apenas se incrementa el uso de procesador. La causa es que en este momento habíamos llegado a la cantidad máxima de tráfico que eran capaces de generar las instancias cliente hacia el sistema. Para solucionarlo y conseguir llegar al

máximo número de instancias servidor, se han añadido dos instancias más sobre el minuto 22, de esta forma, se ha conseguido alcanzar los requisitos de uso de procesador para que saltase de nuevo la alarma y añadiese la ultima instancia.

Tras introducir la cuarta maquina en el grupo de escalado se ha producido un desajuste en el rendimiento, dejando a dos instancias con carga alta y otras dos con carga baja. No sabemos con exactitud a que se debe este fenómeno, pero es posible que sea por el tipo de algoritmo de reparto en el balanceador o simplemente algo interno a estas instancias.

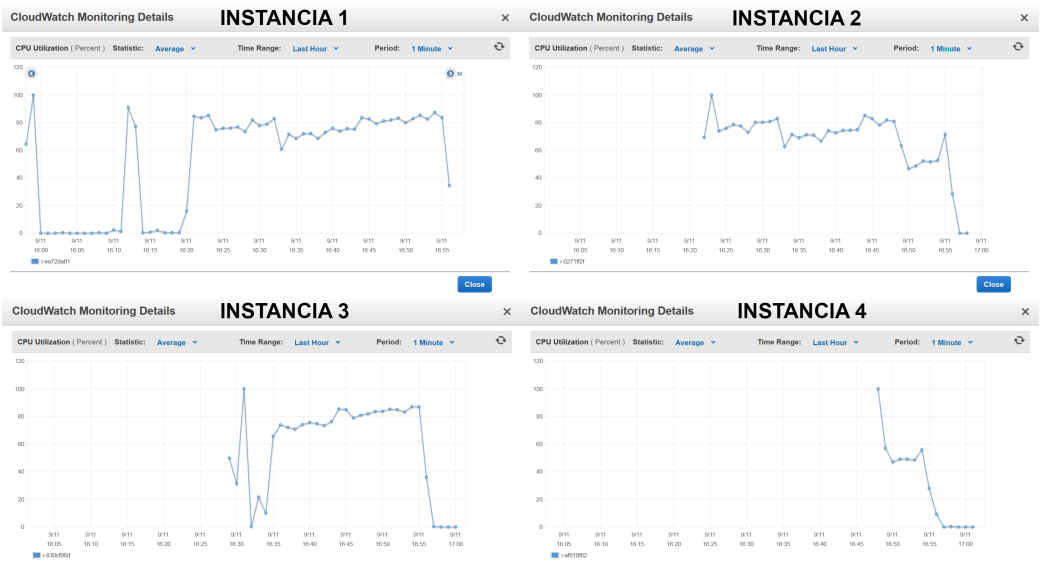

Figura 6.2: Graficos CPU CloudWatch

Fijándonos en la figura 6.2 observamos que el gráfico creado por CloudWatch es muy similar al que hemos obtenido. Pero hay un dato curioso, y es que los porcentajes de uso son bastante más elevados. Al definir las alarmás habíamos establecido que se crease una nueva instancia al superar un 70 % de utilización durante 60 segundos. Si volvemos a observar nuestra gráfica nos damos cuenta que esta condición no se dá en ningún momento. Este tipo de alarma se ha definido con esos parámetros por ser una aplicación experimental. En un entorno de producción buscaríamos intervalos más largos, ya que al crear una instancia, por poco tiempo que se use pagamos una hora completa.

6.2. Cantidad de memoria RAM en uso

Un aspecto en el que nos ha sorprendido gratamente Node.JS, ya se anunciaba en la sección 2.2 que por su forma de trabajar asíncrona y orientada a eventos, el consumo de memoria era muy bajo. Aun así para nada esperabamos unos resultados tan sorprendentes.

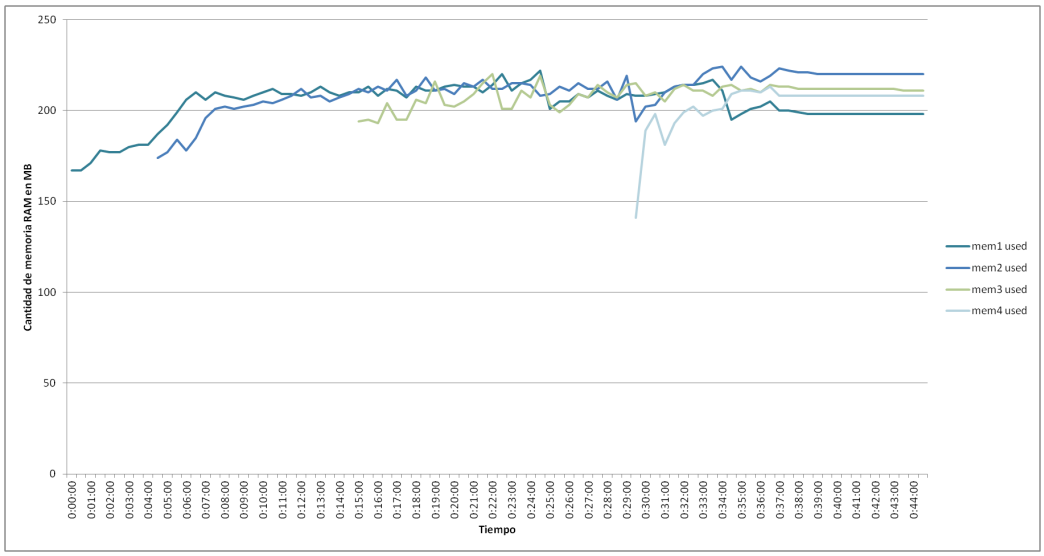

Figura 6.3: Consumo de memoria RAM en MB

Y es que en este caso, apenas cambia al añadir nuevas instancias, pero el dato sorprendente es el bajísimo consumo de memoria. Entre el sistema operativo en ejecución (AMI ubuntu 14.04) y el servicio REST no se han superado los 250mb, algo que parece excesivamente bajo. Viendo estos resultados estamos en condiciones de asegurar que podríamos ahorrar dinero escogiendo una instancia con menos cantidad de memoria. Pues los 3,75 GB de la instancia elegida se antojan demásiado grandes.

6.3. Uso de la red

Vamos ahora a analizar el uso de la red. En la siguiente figura podemos observar la cantidad de información enviada y recibida por segundo. El gráfico esta realizado al mismo tiempo que los anteriores, por que lo que de nuevo observamos los instantes en los que se añaden nuevas instancias.

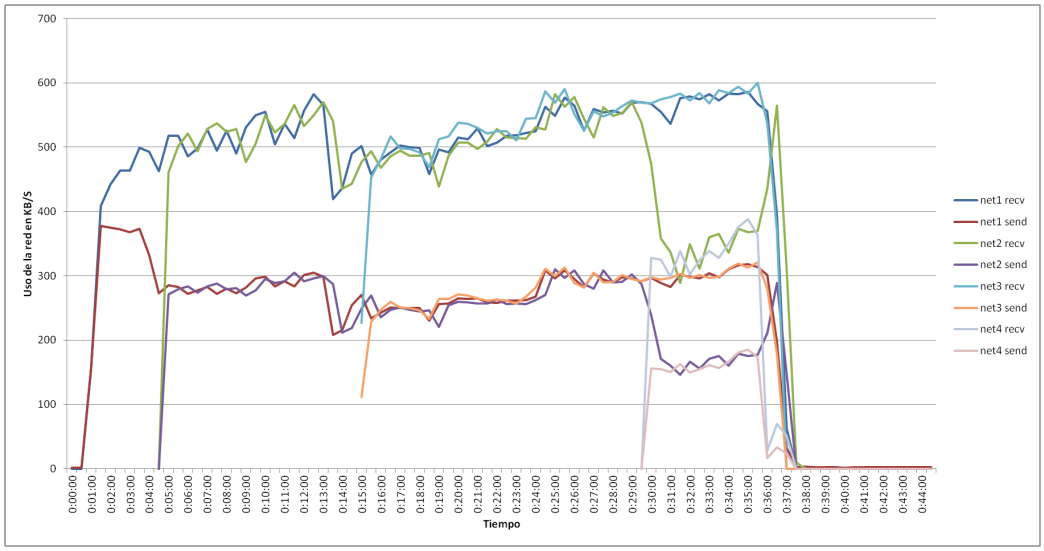

Figura 6.4: Uso de la red

Si menospreciamos las escrituras realizadas y contamos solo las lecturas podemos realizar un cálculo aproximado del número de peticiones por segundo que procesa cada instancia. Ya que las peticiones enviadas (send en la gráfica) tenían un peso aproximado de 700 bytes. Considerando que el pico máximo de envió son 380kb/s, calculamos:

$\frac{380*1024}{700} \simeq 555 Peticiones por segundo$

De esta forma deducimos que 555 peticiones es el máximo de peticiones por segundo que puede procesar una de nuestras instancias. Este es un cálculo estimativo, ya que deberíamos realizarlo con una sola instancia y en un entorno más controlado. Pero en estos momentos vamos a tomarlo como cierto para utilizarlo en la próxima sección, donde estimaremos de forma aproximada los costes del servicio REST.

6.4. Costes

Veamos una aproximación de los precios en diferentes escenarios. Para ello utilizando la calculadora de AWS [10]. Consideremos un primer caso con los siguientes parámetros.

- Una única instancia con una media de 280 peticiones por segundo.
- Petición media de 700 bytes.
- Con una base de datos de 1GB.
- Las unidades de lectura y escritura de la base de datos se gestionan manualmente y están establecidas en 500 y 10 respectivamente.
- La BDD con consistencia eventual.

Calculemos primero el total del tráfico saliente mensual. El entrante no nos importa por que es gratuito.

$280 * 700 * 60 * 60 * 24 * 30 = 508032000000 bytes menusales \simeq 473 GB$

Tras introducir todas estas características en la calculadora obtenemos un coste de 101.65$ mensuales. Parece un buen precio si consideramos que no tenemos ninguna inversión inicial. No necesitamos tener un lugar donde guardar los servidores. Y tampoco pagar la conexión a Internet de los servidores.

⊖	Amazon EC2 Service (US-East)		$ 73.33
	Compute:	$ 51.24	
	Elastic LBs:	$ 18.30	
	Data Processed by Elastic LBs:	$ 3.79	
⊖	Amazon DynamoDB Service (US-East)		$ 28.32
	Provisioned Throughput Capacity:	$ 28.08	
	Indexed Data Storage:	$ 0.24	
⊖	AWS Support (Basic)		$ 0.00
	Support for all AWS services:	$ 0.00	
Total Monthly Payment:		**$**	101.65

Figura 6.5: Precio una instancia al 50 %

Si considerásemos un escenario donde el sistema trabajase al máximo de su carga el resultado es bastante distinto. Vamos a utilizar unos valores similares a los anteriores.

- Cuatro instancias con una media de 555 peticiones por segundo.
- Petición media de 700 bytes.
- Con una base de datos de 1GB.
- Las unidades de lectura y escritura de la base de datos se gestionan manualmente y están establecidas en 2000 y 40 respectivamente.
- La BDD con consistencia eventual.

$$555 * 4 * 700 * 60 * 60 * 24 * 30 = 4027968000000 bytes menusales \simeq 3751 GB$$

⊖	Amazon EC2 Service (US-East)		$ 253.27
	Compute:	$ 204.96	
	Elastic LBs:	$ 18.30	
	Data Processed by Elastic LBs:	$ 30.01	
⊖	Amazon DynamoDB Service (US-East)		$ 128.24
	Provisioned Throughput Capacity:	$ 128.00	
	Indexed Data Storage:	$ 0.24	
⊖	AWS Support (Basic)		$ 0.00
	Support for all AWS services:	$ 0.00	
Total Monthly Payment:		**$**	381.51

Figura 6.6: Precio de cuatro instancias al 100 %

Un precio que a nuestro parecer sigue siendo bastante atractivo para el alto rendimiento que ofrece. Una opción interesante de AWS es que permite comprar con antelación recursos. Si supiésemos que el sistema siempre va a tener este tipo de carga y comprásemos los recursos en packs de 3 años, el precio baja considerable.

Calculemos ahora de nuevo el caso anterior, con la pre-compra de las instancias a 3 años.

⊖	Amazon EC2 Service (US-East)		$ 1440.23
	Compute:	$ 43.92	
	Reserved Instances (One-time Fee):	$ 1348.00	
	Elastic LBs:	$ 18.30	
	Data Processed by Elastic LBs:	$ 30.01	
⊖	Amazon DynamoDB Service (US-East)		$ 128.24
	Provisioned Throughput Capacity:	$ 128.00	
	Indexed Data Storage:	$ 0.24	
⊖	AWS Support (Business)		$ 156.85
	AWS Support Plan Minimum:	$ 100.00	
	Support for Reserved Instances (One-time Fee):	$ 56.85	
Total One-Time Payment:		**$**	1404.85
Total Monthly Payment:		**$**	320.47

Figura 6.7: Precio de cuatro instancias al 100 % con pre-compra de recursos a 3 años

Capítulo 7

Conclusión

Durante la realización de este trabajo fin de máster hemos conseguido resolver el problema planteado. Se ha dado respuesta a la necesidad de implementar un repositorio de datos sobre un servicio cloud. Para ello se escogieron diferentes tecnologías. Cada una de ellas relacionada con una o varias asignaturas del máster. De este modo, todos los conocimientos adquiridos durante el curso en la rama de tecnologías cloud han sido aplicados en el trabajo.

Se han logrado los objetivos de escalabilidad, disponibilidad y consistencia. Consiguiendo un sistema robusto y tolerante a fallos. Esta parte ha sido posible gracias a la elección de AWS. Ya que nos ha ofrecido de forma sencilla, la capacidad de generar una arquitectura que cumpliese todos los requerimientos de nuestro sistema.

También se han cumplido los requisitos de seguridad gracias a la implementación de un protocolo HTTPS con el que obtenemos unas comunicaciones seguras.

Por último el apartado del análisis ha sido un éxito, especialmente en cuanto a viabilidad económica. Tras obtener un cálculo aproximado de la caga, hemos sido capaces de establecer aproximaciones a los costes mensuales en diferentes escenarios de rendimiento.

Destacar durante el desarrollo de la parte software la rápida curva de aprendizaje en Node.JS. Las facilidades que ofrece al realizar un proyecto de este tipo. Y el gran apoyo que recibe por parte de la comunidad ofreciendo librerías desinteresadamente.

7.1. Trabajo futuro

Como trabajo futuro ha quedado pendiente mejorar la aplicación Android de forma que dejase de ser un demostración de acceso a los recursos REST y pudiese ser una plataforma estable. Si esto se llegase a completar tendríamos una aplicación sobre la que una agencia como AEPNAA (Asociación Española de Alérgicos a Alimentos y Látex), podría ofrecer a sus socios un método rápido y fiable para conocer que alimentos pueden comer. Para llegar a esta función no solo se tendría que haber mejorado la parte actual, añadiendo nuevos requisitos para la comodidad del usuario. Se debería también implementar una funcionalidad donde ciertos administradores aceptasen o denegasen el nuevo contenido. Consiguiendo de este modo, ofrecer confiabilidad a los usuarios.

Otro punto importante y que puede suponer un gran ahorro en el coste mensual de la aplicación, es la base de datos. En estos momentos DynamoDB no ofrece una escalabilidad dinámica automática. Ahora mismo, podemos configurar alarmás que nos vayan avisando del nivel de carga. Pero no cambian de forma automática el aprovisionamiento de recursos de la base de datos. Para modificarlo debemos utilizar llamadas a la API o acceder a la consola de administración y hacerlo manualmente. Y en este último punto es donde entra la mejora del sistema, sería interesante implementar un demonio que fuese capaz de monitorizar la cantidad de peticiones por segundo que recibimos en el sistema. Con este número de peticiones podríamos implementar un algoritmo que utilizando las llamadas a la API fuese cambiando el rendimiento de la base de datos de forma dinámica, adaptándola así a las necesidades del sistema y ahorrando recursos.

Bibliografía

[1] AWS Case Study: Obama for America Campaign http://aws.Amazon.com/es/solutions/case-studies/obama/, 2012

[2] Amazon Web Services (AWS) Servicios de informatica en la nube http://aws.Amazon.com/es/

[3] Alok Adhao, Anuradha Gaikwad: Asynchronous and non-blocking input/outputusing backbone.js and Node.JS.js http://www.ijaiem.org/volume2issue11/IJAIEM-2013-11-20-058.pdf, IJAIEM Volumen 2, Issue 11, November 2013

[4] Node.JS Wikipedia http://es.wikipedia.org/wiki/Node.JS

[5] Node.JS http://Node.JSjs.org/

[6] express http://expressjs.com/

[7] Sinatra http://www.sinatrarb.com/

[8] Passport http://passportjs.org/

[9] Android http://developer.Android.com/index.html

[10] SIMPLE MONTHLY CALCULATOR http://calculator.s3.Amazonaws.com/index.html

[11] Mark D. Hill: "What is Scalability?", ACM SIGARCH Computer Architecture News, 18(4):18?21, December 1990.

[12] Werner Vogels: "Eventually consistent". Commun. ACM 52(1): 40-44 (2009)

[13] Flaviu Cristian: "Understanding Fault-Tolerant Distributed Systems", Communications of the ACM, 34(2):57-78, febrero 1991.

[14] Susan B. Davidson, Hector Garcia-Molina, Dale Skeen: "Consistency in Partitioned Networks". ACM Comput. Surv. 17(3): 341-370 (1985)

[15] D. Kyriazis: "Cloud Computing Service Level Agreements: Exploitation of Research Results", informe disponible en: http://ec.europa.eu/digital-agenda/en/printpdf/67211, Directorio General de Redes de Comunicacion, Contenido y Tecnologia, Unidad E2, Programás y Servicios "Cloud", Comision Europea, Bruselas, julio 2013. 61 pgs.

[16] F. Callegati, W. Cerroni, M. Ramilli :"Man-in-the-Middle Attack to the HTTPS Protocol", IEEE Security and Privacy Volume 7 Issue 1, January 2009 Pages 78-81

[17] Amazon Web Services, Inc: "Amazon CloudWatch - Developer Guide". Guia disponible en `http://awsdocs.s3.Amazonaws.com/AmazonCloudWatch/latest/acw-dg.pdf`, agosto 2010.

[18] Amazon Web Services, Inc: "Amazon Auto Scaling - Developer Guide". Guia disponible en `http://awsdocs.s3.Amazonaws.com/AutoScaling/latest/as-dg.pdf`, enero 2011.

[19] Amazon Web Services, Inc: "Elastic Load Balancing". Documentacion disponible en: `http://aws.Amazon.com/elasticloadbalancing/`, febrero 2014.

Apéndice A

Anexo

A.1. server.js

```
/**
* Aplication server
*/
var fs = require('fs');
var http = require('http');
var https = require('https');
var express     = require('express');
var routes      = require('./routes');
var key = fs.readFileSync('../certificados/ec.key');
var cert = fs.readFileSync('../certificados/ec.pem');
var options = {
      key: key,
      cert: cert
};
var passport = require('passport');
var util = require('util');
var BearerStrategy = require('passport-http-bearer').Strategy;
var AWS = require('aws-sdk');
AWS.config.loadFromPath('./config.json');
var dd = new AWS.DynamoDB();

var Server = function() {

  var self = this;

  // variables for listen
  self.startVariables = function(){
```

```
    self.port = 44401;
    self.portNoSec = 8080;
    self.ip    = "127.0.0.1";
  }

  //close server with signal
  self.terminator = function(sig){
    if (typeof sig === "string") {
      console.log('%s: Received %s - terminating server ...',
          Date(Date.now()), sig);
      process.exit(1);
    }
    console.log('%s: Node server stopped.', Date(Date.now()));
  };

  self.setupTerminationHandlers = function(){
    //  close aplication
    process.on('exit', function() { self.terminator(); });

    // signals
    ['SIGHUP', 'SIGINT', 'SIGQUIT', 'SIGILL', 'SIGTRAP', '
        SIGABRT',
    'SIGBUS', 'SIGFPE', 'SIGUSR1', 'SIGSEGV', 'SIGUSR2', '
        SIGTERM'
    ].forEach(function(element, index, array) {
    process.on(element, function() { self.terminator(element);
        });
    });
  };

  //compare user token in db
  self.findByToken = function(token, fn) {
    dd.scan({
      'AttributesToGet' : ['user'],
      'TableName':'user',
      'ScanFilter':{ 'token':{ 'AttributeValueList':[{'S':token
          }],'ComparisonOperator':'EQ'}},
      'Limit': '1'
    }, function(err, data) {
      if (err){
        console.log("null");
        return fn(null, null);
      }
      else{
        return fn(null, data.LastEvaluatedKey.user.S);
```

```
    }
  });
}

passport.use(new BearerStrategy({
  },
  function(token, done) {
    process.nextTick(function () {
      self.findByToken(token, function(err, user) {
        if (err) { return done(err); }
        if (!user) { return done(null, false); }
        return done(null, user);
      })
    });
  }
));

// create url routes and actions
self.createRoutes = function() {
  self.app.get('/', routes.home);
  self.app.get('/ingredient', routes.resourceGet);
  self.app.get('/allergen', routes.resourceGet);
  self.app.get('/aliment', routes.resourceGet);
  self.app.post('/login', routes.loginPost);
  self.app.post('/signin', routes.signinPost);
  self.app.post('/ingredient', passport.authenticate('bearer',
      { session: false }), routes.resourcePost);
  self.app.post('/allergen', passport.authenticate('bearer', {
      session: false }), routes.resourcePost);
  self.app.post('/aliment', passport.authenticate('bearer', {
     session: false }), routes.resourcePost);
  self.app.put('/user/:id', passport.authenticate('bearer', {
     session: false }), routes.resourceIdPut);
  self.app.delete('/user/:id', passport.authenticate('bearer',
      { session: false }), routes.resourceIdDel);
  self.app.get('/ingredient/:id', routes.resourceIdGet);
  self.app.put('/ingredient/:id', passport.authenticate('
     bearer', { session: false }), routes.resourceIdPut);
  self.app.delete('/ingredient/:id', passport.authenticate('
     bearer', { session: false }), routes.resourceIdDel);
  self.app.get('/allergen/:id', routes.resourceIdGet);
  self.app.put('/allergen/:id', passport.authenticate('bearer
     ', { session: false }), routes.resourceIdPut);
  self.app.delete('/allergen/:id', passport.authenticate('
     bearer', { session: false }), routes.resourceIdDel);
```

```
    self.app.get('/aliment/:id', routes.resourceIdGet);
    self.app.put('/aliment/:id', passport.authenticate('bearer',
        { session: false }), routes.resourceIdPut);
    self.app.delete('/aliment/:id', passport.authenticate('
        bearer', { session: false }), routes.resourceIdDel);
  };

  self.initializeServer = function() {
    // Setup and configure Express http server. Expect a
        subfolder called "static" to be the web root.
    self.app = express();
    self.app.use(express.urlencoded());
    self.app.use(express.json());
    self.app.use(passport.initialize());
    https.createServer(options, self.app).listen(self.port,
        function(){
      console.log(' ');
      console.log
          ('#-----------------------------------------------#');
      console.log(' ');
      console.log('Secure restServer listening on port ' + self.
          port);
      console.log(' ');
    });
    http.createServer(self.app).listen(self.portNoSec, function
        (){
      console.log(' ');
      console.log
          ('#-----------------------------------------------#');
      console.log(' ');
      console.log('Unsecure restServer listening on port ' +
          self.portNoSec);
      console.log(' ');
    });
  };

  // Start all server aplication
    self.initialize = function() {
    self.startVariables();
    self.setupTerminationHandlers();
    self.initializeServer();
    self.createRoutes();
  };

};
```

```
var serv = new Server();
serv.initialize();
```

A.2. index.js

```
var AWS = require('aws-sdk');
var jwt = require('jwt-simple');
AWS.config.loadFromPath('./config.json');
var dd = new AWS.DynamoDB();
var privatekey = 'pruebaclaveprivada';
var verbose = true;

exports.home = function(req, res){
        res.send("NODEJS REST SERVER");
};
//
//Codigo a ejecutar cuando un usuario se logea en el server
//recibe el usuario y la clave y en caso afirmativo devuelve
//un token para poder acceder a la aplicacion y el token se
//inserta en la base de datos
//
exports.loginPost = function(req, res){
  try{
    var item = req.body;
    if (item.user && item.password){
      var ID = item.user.S;
      var PASS = item.password.S;
      dd.getItem({
        'TableName':'user',
        'Key':{ 'user': { 'S': ID } }
      }, function(err, data) {
        if (err){
          console.log(err, err.stack);
          res.status(500);
          res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
        }
        else {
          if (data.Item){
            if (data.Item.user.S == ID && data.Item.password.S
                == crypt(PASS,privatekey)){
              req.body.microtime = ({'N': new Date().getTime().
                  toString()});
```

```
            var token = jwt.encode(req.body, privatekey);
            data.Item.token = ({'S': token});
            dd.putItem({
              'TableName': 'user',
              'Item': data.Item
            }, function(err, data) {
              if (err){
                console.log(err, err.stack);
                res.status(500);
                res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
              }else {
                if (verbose){
                  console.log("<< LOGIN - 1001");
                  console.log(data.Item);
                }
                res.send('{"CODE":"1001","MSG":"'+token+'"}');
              }
            });
          }else {
            res.status(401);
            res.send('{"CODE":"FALSE","MSG":"INCORRECT
               PASSWORD"}');
          }

        }else{
          res.status(401);
          res.send('{"CODE":"FALSE","MSG":"USER DO NOT EXIST
              "}');
        }
      }
    });
  }else{
    res.status(401);
    res.send('{"CODE":"FALSE","MSG":"NICKNAME AND PASSWORD
       REQUIRED"}');
  }
}catch(err){
  res.status(500);
  res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
}
};
//
//Codigo a ejecutar para registrar un usuario
//
exports.signinPost = function(req, res){
```

```
try{
  var table = "user";
  var item = req.body;
  if (item.user && item.password && item.email){
    var ID = item.user.S;
    item.microtime = ({'N': new Date().getTime().toString()});
    item.password.S = crypt(item.password.S,privatekey);
    dd.getItem({
      'TableName':'user',
      'Key':{ 'user': { 'S': ID } }
    }, function(err, data) {
      if (err){
        console.log(err, err.stack);
        res.status(500);
        res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
      }else{
        if(data.Item)
        {
          res.status(401);
          res.send('{"CODE":"FALSE","MSG":"NICKNAME REGISTERED
              , CHOOSE OTHER"}');
        }else{
          dd.putItem({
            'TableName': table,
            'Item': item
          }, function(err, data) {
            if (err){
              console.log(err, err.stack);
              res.status(500);
              res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
            }else {
              if (verbose){
                console.log("<< SIGNIN - 1002");
                console.log(item);
              }
              res.send('{"CODE":"1002","MSG":"NEW USER CREATED
                  , YOU CAN LOGIN NOW"}');
            }
          });
        }
      }
    });
  }else{
    res.status(401);
```

```
      res.send('{"CODE":"FALSE","MSG":"NICKNAME, PASSWORD AND E-
          MAIL REQUIRED"}');
    }
  }catch(err){
    res.status(500);
    res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
  }
};
//
//Codigo a ejecutar cuando se realiza un get sobre un recurso
//
exports.resourceGet = function(req, res){
  try{
    var table = replaceAll(req.url,"/","");
    dd.scan({
      'TableName': table
    },function(err, data) {
      if (err){
        console.log(err, err.stack);
        res.status(500);
        res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
      }
      else {
        if (verbose){
          console.log(">> GET - 1003");
          console.log(data);
        }
        res.send('{"CODE":"1003","MSG":'+JSON.stringify(data.
            Items)+'}');
      }
    });
  }catch(err){
    res.status(500);
    res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
  }
};
//
//Codigo a ejecutar cuando se realiza un post sobre un recurso
//
exports.resourcePost = function(req, res){
  try{
    var table = replaceAll(req.url,"/","");
    var item = req.body;
    if ( (item.barcode && item.name && item.ingredients) || (
        item.ingredient && item.allergens) || (item.allergen &&
```

```
  item.symptoms) ){
item.microtime = ({'N': new Date().getTime().toString()});
token = req.headers.access_token;
params = {'TableName': table};
if(table == "allergen") params.Key = { 'allergen': { 'S':
    item.allergen.S } };
else if(table == "aliment") params.Key = { 'barcode': { 'N
    ': item.barcode.N } };
else if(table == "ingredient") params.Key = { 'ingredient
    ': { 'S': item.ingredient.S } };
dd.scan({
  'TableName':'user',
  'ScanFilter':{ 'token':{ 'AttributeValueList':[{'S':
     token}],'ComparisonOperator':'EQ'}},
}, function(err, data) {
  if (err){
    console.log(err, err.stack);
    res.status(500);
    res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
  }
  else{
    item.user = ({'S': data.Items[0].user.S});
    dd.getItem(params, function(err, data) {
      if (err){
        console.log(err, err.stack);
        res.status(500);
        res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
      }
      else {
        if (data.Item){
          res.status(406);
          res.send('{"CODE":"FALSE","MSG":"DUPLICATED KEY
              "}');
        }else{
          dd.putItem({
            'TableName': table,
            'Item': item
          }, function(err, data) {
            if (err){
              console.log(err, err.stack);
              res.status(500);
              res.send('{"CODE":"FALSE","MSG":"DB ERROR
                  "}');
            }
            else {
```

```
                    if (verbose){
                      console.log("<< POST - 1004");
                      console.log(item);
                    }
                    res.send('{"CODE":"1004","MSG":"SAVE CORRECT
                        "}');
                  }
                });
              }
            }
          });
        }
      });
    }else{
      res.status(406);
      res.send('{"CODE":"FALSE","MSG":"SOME KEY MISSED"}');
    }
  }catch(err){
    res.status(500);
    res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
  }
};
//
//Codigo a ejecutar cuando se realiza un get sobre el id de un
    recurso
//
exports.resourceIdGet = function(req, res){
  try{
    var ID = req.params.id;
    var table = replaceAll(req.url,"/","").replace(req.params.id
        .replace(" ","%20"),"");
    params = {'TableName': table};
    if(table == "allergen") params.Key = { 'allergen': { 'S': ID
        } };
    if(table == "aliment") params.Key = { 'barcode': { 'N': ID }
        };
    if(table == "ingredient") params.Key = { 'ingredient': { 'S
        ': ID } };
    dd.getItem(params, function(err, data) {
      if (err){
        console.log(err, err.stack);
        res.status(500);
        res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
      }
      else {
```

```
        if (verbose){
          console.log(">> GET ID - 1005");
          console.log(data);
        }
        if (data.Item) res.send('{"CODE":"1005","MSG":'+JSON.
            stringify(data.Item)+'}');
        else res.send('{"CODE":"FALSE","MSG":"ALIMENT NOT IN DB
            "}');
      }
    });
  }catch(err){
    res.status(500);
    res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
  }
};
//
//Codigo a ejecutar cuando se realiza un put sobre el id de un
    recurso
//
exports.resourceIdPut = function(req, res){
  try{
    var ID = req.params.id;
    var table = replaceAll(req.url,"/","").replace(req.params.id
        .replace(" ","%20"),"");
    var item = req.body;
    var KEY={};
    if(table == "user") KEY.user = ({'S': ID});
    if(table == "allergen") KEY.allergen = ({'S': ID});
    if(table == "aliment") KEY.barcode = ({'N': ID});
    if(table == "ingredient") KEY.ingredient = ({'S': ID});
    item.microtime = ({'Value': {'N': new Date().getTime().
        toString()}, 'Action': 'PUT'});
    token = req.headers.access_token;
    dd.scan({
      'TableName':table,
      'ScanFilter':{ 'token':{'ComparisonOperator':'EQ','
          AttributeValueList':[{'S':token}]}}
    }, function(err, data) {
      if (err){
        res.status(500);
        console.log(err, err.stack);
        res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
      }
      else{
        if(table == "user") {
```

```
            if (ID != data.Items[0].user.S){
              res.status(401);
              res.send('{"CODE":"FALSE","MSG":"INCORRECT RIGHTS
                  "}');
              return false;
            }
          }
          dd.updateItem({
            'TableName': table,
            'Key': KEY,
            'AttributeUpdates' : item
          },function(err, data) {
            if (err){
              console.log(err, err.stack);
              res.status(500);
              res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
            }
            else {
              if (verbose){
                console.log("<< PUT ID - 1006");
                console.log(KEY);
              }
              res.send('{"CODE":"1006","MSG":"SAVE OR EDIT CORRECT
                  "}');
            }
          });
        }
      });
    }catch(err){
      res.status(500);
      res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
    }
};
//
//Codigo a ejecutar cuando se realiza un delete sobre el id de
   un recurso
//
exports.resourceIdDel = function(req, res){
  try{
    var ID = req.params.id;
    var table = replaceAll(req.url,"/","").replace(req.params.id
        .replace(" ","%20"),"");
    var item = {};
    if(table == "user") item.user = ({'S': ID});
    if(table == "allergen") item.allergen = ({'S': ID });
```

```
    if(table == "aliment") item.barcode = ({'N': ID });
    if(table == "ingredient") item.ingredient = ({'S': ID });
    token = req.headers.access_token;
    dd.scan({
      'TableName':'user',
      'ScanFilter':{ 'token':{ 'AttributeValueList':[{'S':token
         }],'ComparisonOperator':'EQ'}},
    }, function(err, data) {
      if (err){
        console.log(err, err.stack);
        res.status(500);
        res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
      }
      else{
        if(table == "user") {
          item.user = ({'S': data.Items[0].user.S});
          if (item.user.S !=  ID){
            return false;
          }
        }
        dd.deleteItem({
          'TableName': table,
          'Key': item
        }, function(err, data) {
          if (err){
            console.log(err, err.stack);
            res.status(500);
            res.send('{"CODE":"FALSE","MSG":"DB ERROR"}');
          }
          else {
            if (verbose){
              console.log("<< DEL ID - 1007");
              console.log(item);
            }
            res.send('{"CODE":"1007","MSG":"DELETE CORRECT"}');
          }
        });
      }
    });
  }catch(err){
    res.status(500);
    res.send('{"CODE":"FALSE","MSG":"'+err+'"}');
  }
};
//
```

```
//Codigo auxiliar para remplazar todos los caracteres de una
    cadena
//
function replaceAll( text, busca, reemplaza ){
  while (text.toString().indexOf(busca) != -1)
    text = text.toString().replace(busca,reemplaza);
  return text;
}
//
//Codigo auxiliar para encriptar
//
function crypt(key, pass) {
     var crypto = require('crypto');
     return crypto.createHmac('sha1', key).update(pass).digest('
         hex');
}
```