BUAP

Araujo García José Mauricio

Dra. Hilda Castillo Zacatelco

Sistemas Operativos II

Proyecto Final

Introducción

Uno de los mayores retos que presentan las aplicaciones actuales es el de escalar constantemente para cubrir el total de sus necesidades a medida que estas aumentan, sin perder la disponibilidad del servicio, de manera transparente para los usuarios y agregando la menor complejidad posible a la infraestructura que soporta la aplicación. Este reto ha tratado de resolverse de diferentes maneras desde la incepción de las aplicaciones por computadora, pero este esfuerzo por encontrar la solución óptima se ha hecho mayor en los últimos 10 años debido a la popularización del Internet y de los dispositivos móviles inteligentes.

Los sistemas distribuidos se presentan como una solución obvia y casi natural al problema anteriormente expuesto. Es por este motivo que se han propuesto una multitud de componentes, protocolos, sistemas operativos, bases de datos, etc. a través de los años, que han convergido en lo que es ahora conocido como el Cloud Computing.

En el presente documento se realizará el análisis de un problema particular de la vida real y se expondrá una solución utilizando el paradigma distribuido.

Problemática

Antes de presentar el problema, es importante familiarizarse con el concepto de un instrumento psicológico, que es definido por los psicólogos como:

"Una medida objetiva y estandarizada de una muestra de conducta; las pruebas, tests o instrumentos de medición psicológicos son herramientas que recogen muestras de conducta producidas por los sujetos en respuesta a unos estímulos que le son presentados. Estas respuestas son puntuadas o valoradas según unos criterios, ofreciendo información del lugar que ocupa el sujeto dentro de un grupo de referencia normativo, medir las diferencias entre individuos y entre las reacciones de la misma persona en circunstancias distintas."

Teniendo en cuenta esta definición, la problemática a resolver es la de construir un sistema escalable, transparente y tolerante a fallas para la administración y aplicación de instrumentos psicológicos a pacientes de un hospital.

Análisis

Para cumplir los requisitos de escalabilidad, transparencia y tolerancia a fallas es necesario pensar con un paradigma distribuido en cuestión a la infraestructura, tecnología de *back-end* y base de datos que se utilizará para soportar la aplicación. Cada una de estas partes tiene que adaptarse al paradigma para que el sistema funcione en su totalidad sin caer en una estructura centralizada tradicional. El sistema se separará en dos capas, la primera capa será la base de datos y encima de esta, existirá la capa de la aplicación.

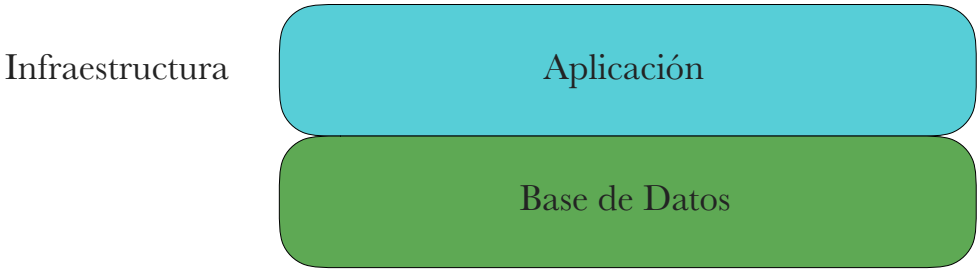

Fig. 1.1. Diagrama de arquitectura de la aplicación.

Base de Datos

La primera consideración será en la primera capa, donde existirá la base de datos. Se utilizará la MongoDB, ya que su estructura NoSQL es mucho más flexible, lo que resultará útil para una aplicación distribuida. Esta base de datos puede configurarse para replicarse a través de diferentes servidores para formar lo que se conoce cómo *Replica Set.* Así, la base de datos proporciona alta disponibilidad y redundancia de la información.

Al tener redundancia, la base de datos se vuelve tolerante a fallas y es resistente a la pérdida de algún servidor. Además de esto, la alta disponibilidad significa que la aplicación puede distribuirse a través de múltiples servidores y mantener la velocidad y consistencia en las operaciones.

En MongoDB, el *replica set* consiste de varios procesos que mantienen la misma información. Solamente uno de estos nodos es nominado como nodo primario y el resto de ellos se denominan

secundarios. El nodo primario es el que confirma las operaciones de escritura y guarda los cambios. Los nodos secundarios reciben este cambio y lo replican en su propia información para que sea consistente con la del nodo primario. Si el nodo primario no se encuentra disponible, se realiza una elección entre los nodos secundarios para elegir un nuevo primario. El diagrama siguiente detalla esta relación.

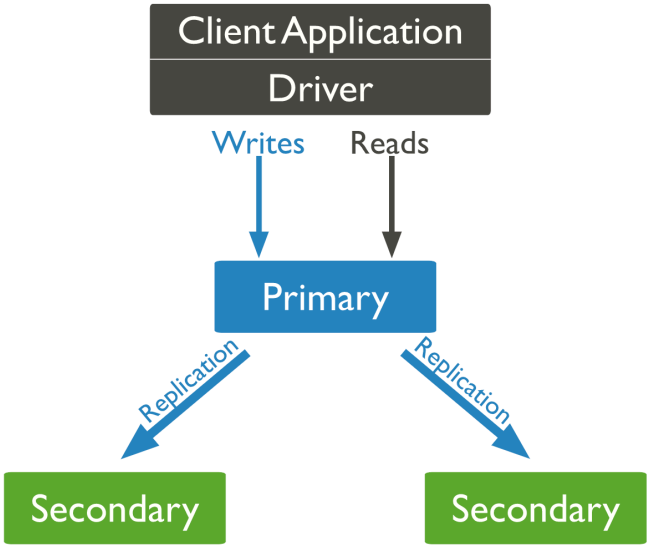

Fig. 1.2. Diagrama de arquitectura de la base de datos.

Si el nodo primario no se comunica con los demás en cierto tiempo, se realiza una elección para elegir un nuevo nodo primario. Durante el tiempo en el que se elige un nuevo nodo primario, las operaciones de escritura no son procesadas, pero las de lectura sí.

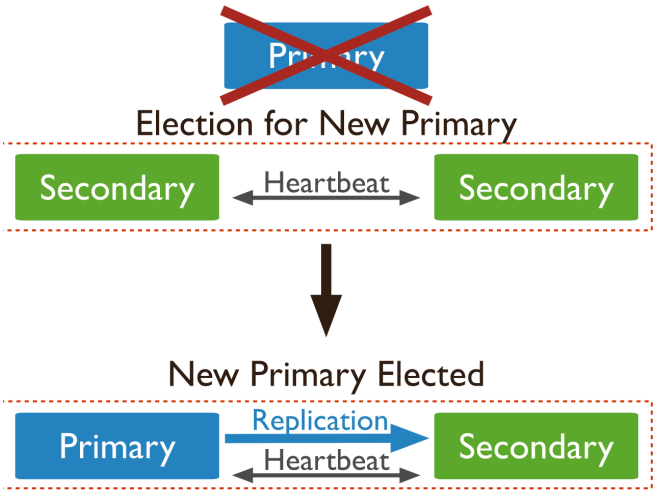

Fig. 1.3. Diagrama de tolerancia a fallas en la base de datos.

Infraestructura

La capa de infraestructura donde reside la aplicación se encontrará en un clúster de Kubernetes. Un clúster de Kubernetes es un conjunto de computadoras con alta disponibilidad interconectadas entre sí y que actúan como una sola unidad. De esta manera la aplicación no está atada a una computadora individual. Como consecuencia de este modelo, la aplicación tiene que ser desplegada en una manera que la separe de cualquier *host*; para lograr esto se tiene que desplegar en un contenedor. Así, la aplicación es independiente del entorno en el que se encuentra, no esta instalada directamente en ningún servidor, simplemente reside dentro de un contenedor que a su vez se encuentra en un servidor. Esto facilita el paradigma distribuido en el sistema, pero el manejo y orquestación de todos los servicios tiene que ser automatizado por Kubernetes. La estructura se aprecia en el siguiente diagrama:

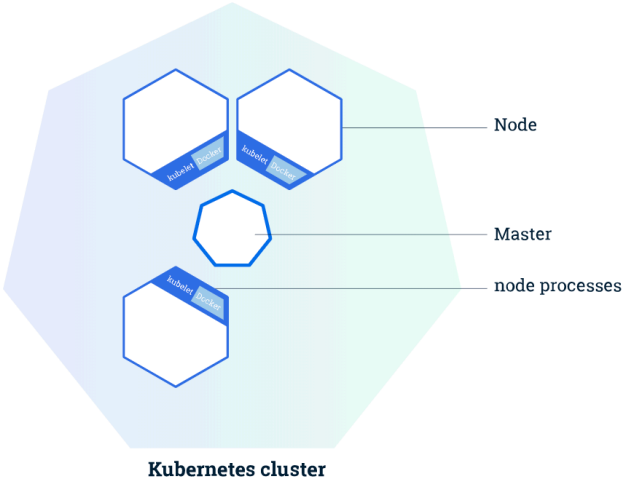

Fig. 1.4. Diagrama de un clúster en Kubernetes.

El clúster tiene un maestro, que es el encargado de cualquier operación dentro de él. Un nodo es cualquier computadora o máquina virtual que tiene la función de trabajador/esclavo. El nodo maestro indica que contenedores tienen que levantarse en cada nodo y se encarga de comunicarlos.

Habiendo detallado la arquitectura de la aplicación, se detallará la implementación actual y los pasos que se siguieron para realizarla.

Aplicación

La estructura de la parte de aplicación es sencillo. Se siguió el paradigma REST, o Representational State Transfer para comunicar al cliente y al servidor. Un sistema REST se caracteriza por su falta de estado, ya que separa completamente al cliente y al servidor desde la implementación. Esta arquitectura separa completamente el código del cliente y del servidor, por lo que ni siquiera tienen que estar escritos en el mismo lenguaje. Además, cada uno de estos componentes se hace modular, escalaban y mantenible. Si el código del cliente cambia, no afecta al servidor y vice-versa, siempre y cuando se mantenga la estructura de la comunicación entre ellos.

En esta arquitectura, el cliente manda solicitudes para crear, modificar, leer y borrar información, y el servidor le contesta con respuestas a las solicitudes hechas. La manera de comunicarse es mediante verbos HTTP (comúnmente GET, POST, PUT, DELETE) que llaman a rutas específicas del servidor (aplicación/usuarios/1029).

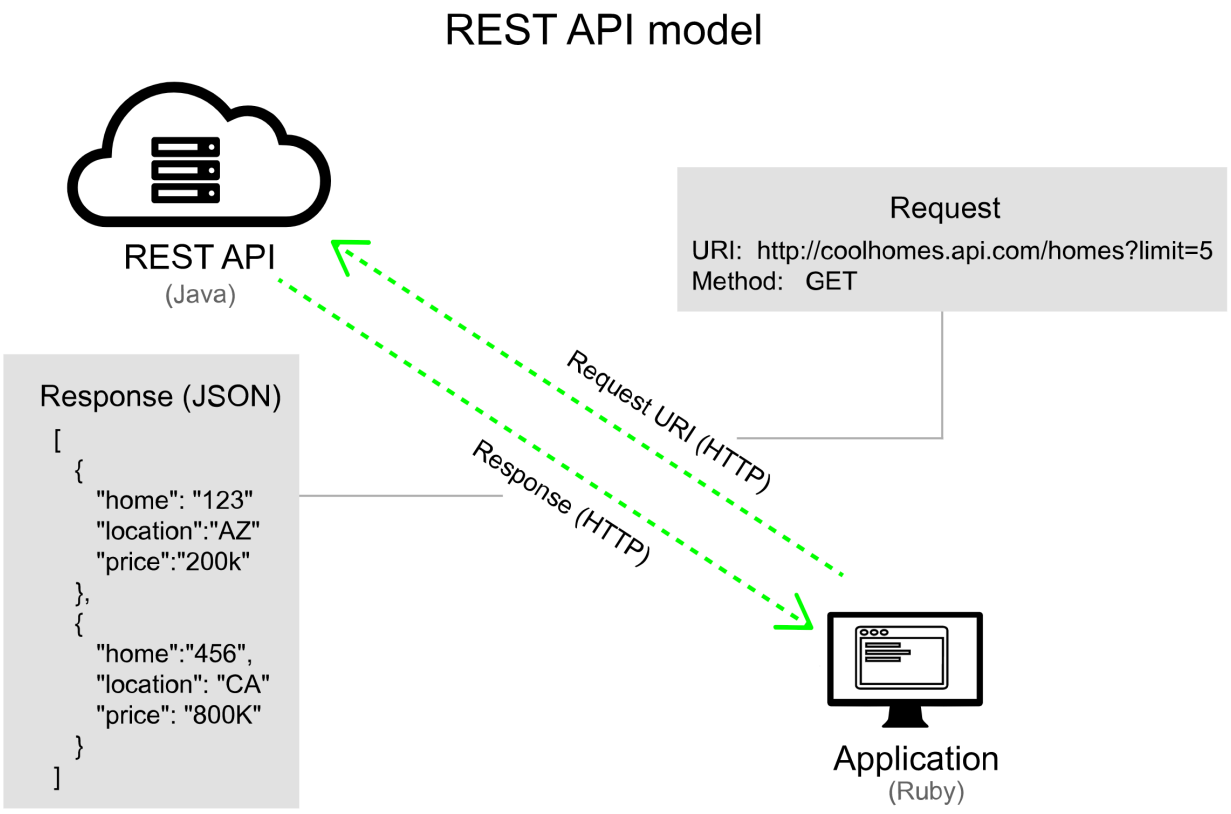

Fig. 1.5. Diagrama de un REST API.

Desarrollo

Lo siguiente es exponer la solución dada. Se decidió desarrollar una aplicación web ya que este tipo de aplicaciones le son familiares a la mayoría de personas de casi todas las edades. El primer paso del desarrollo fue desarrollar una interfaz para la aplicación donde se pudieran crear y aplicar los instrumentos, así como administrar a los usuarios. Esta interfaz fue hecha con la librería Vue.js, los resultados se presentan a continuación.

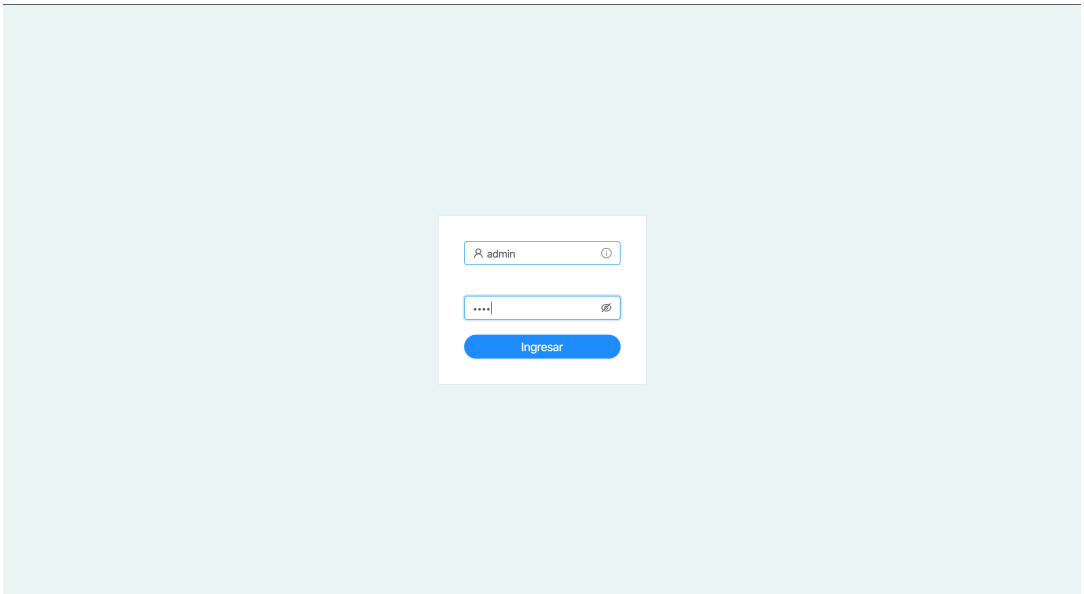

Fig. 2.1. Pantalla para autenticar al usuario.

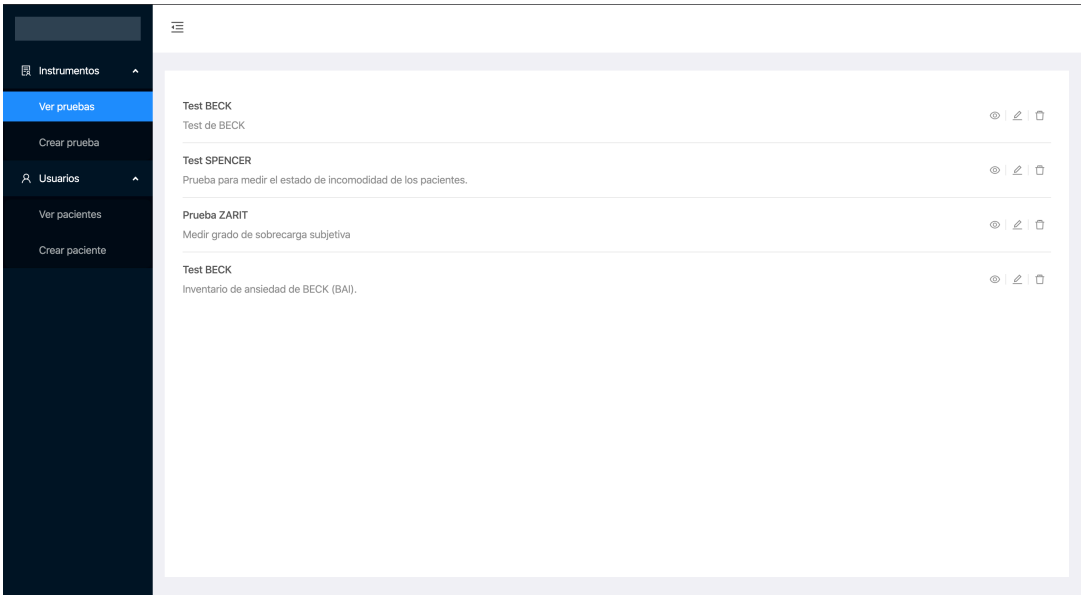

Fig. 2.2. Dashboard administrativo.

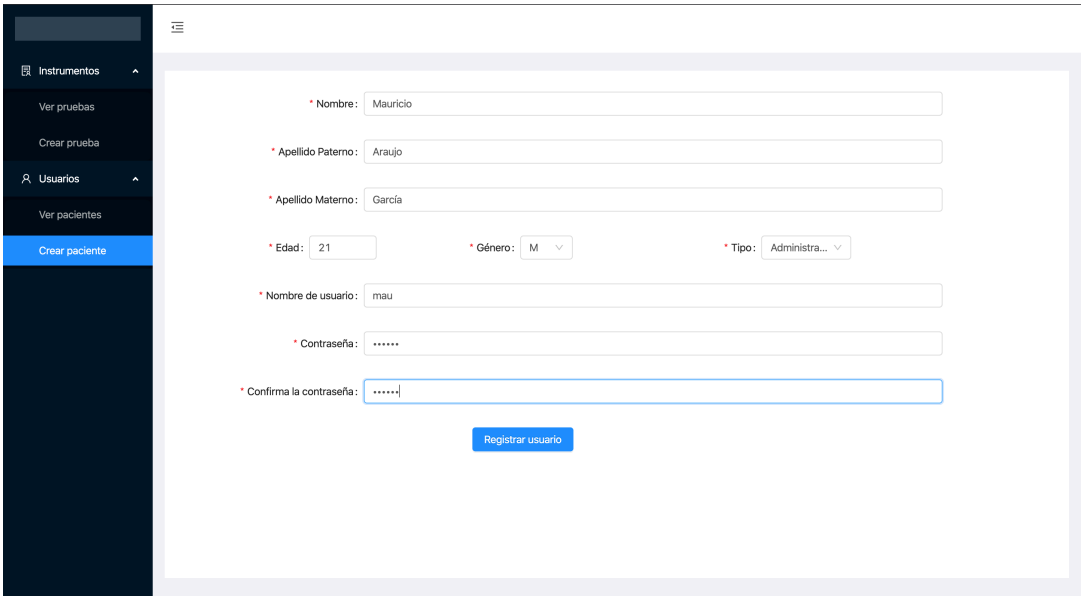

Fig. 2.3. Formulario de creación de usuario.

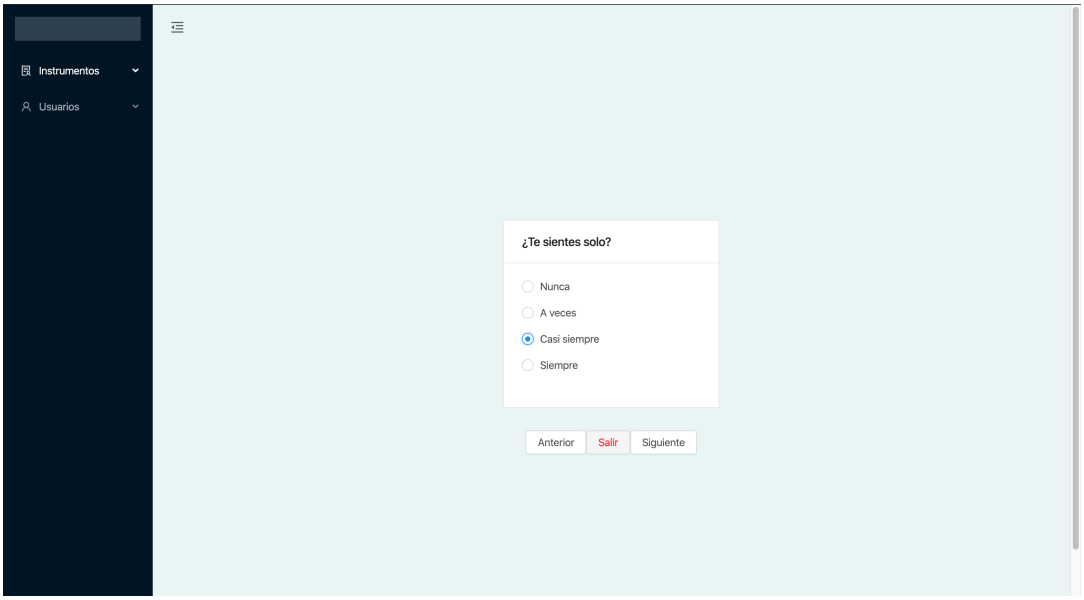

Fig. 2.4. Pantalla de aplicación de instrumento.

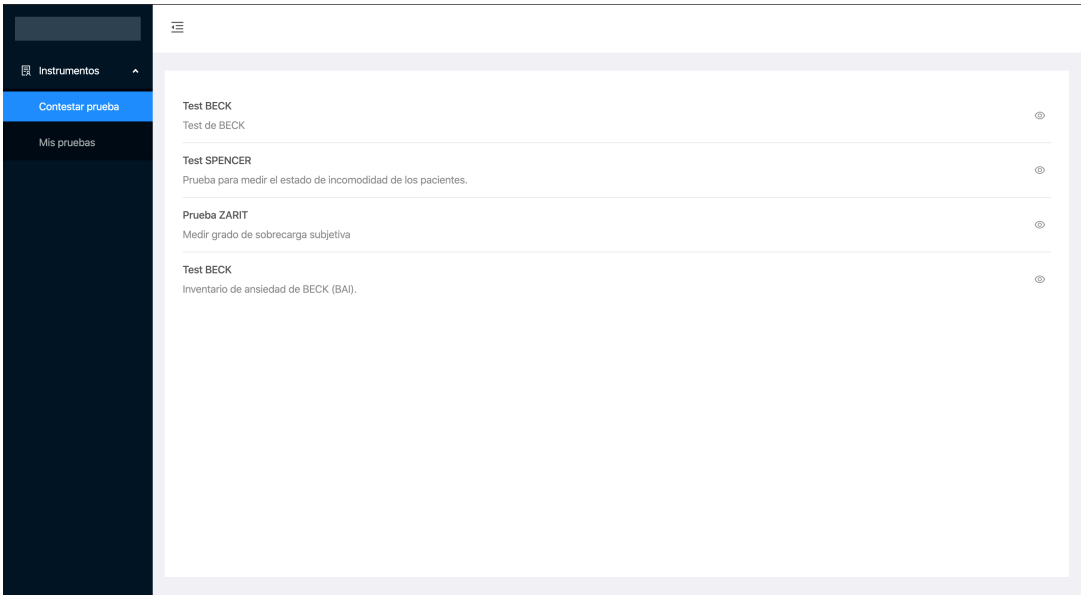

Fig. 2.5. Dashboard de paciente.

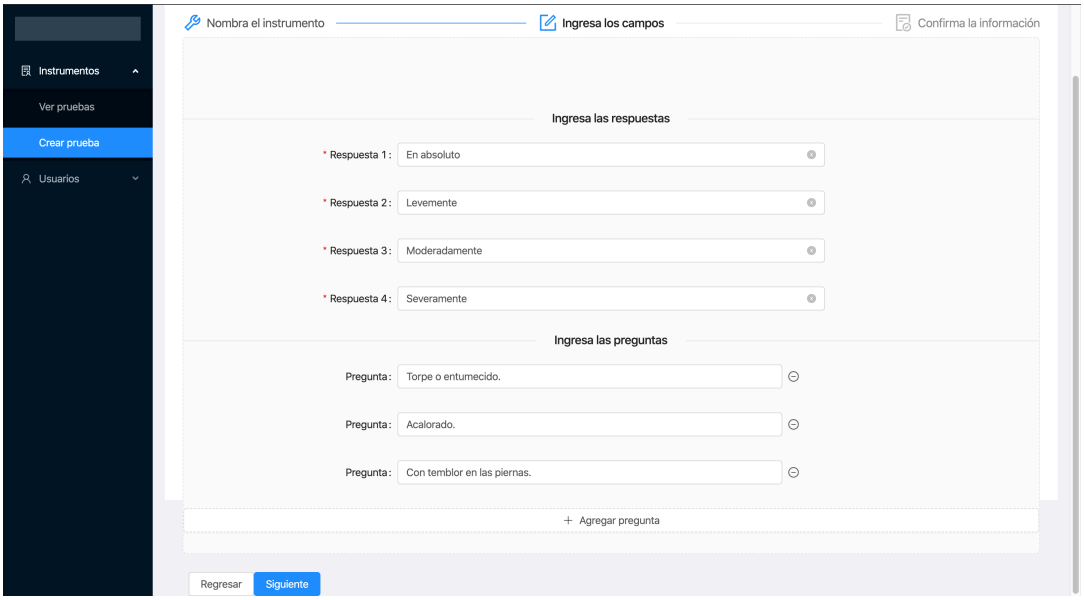

Fig. 2.6. Formulario de creación de un instrumento.

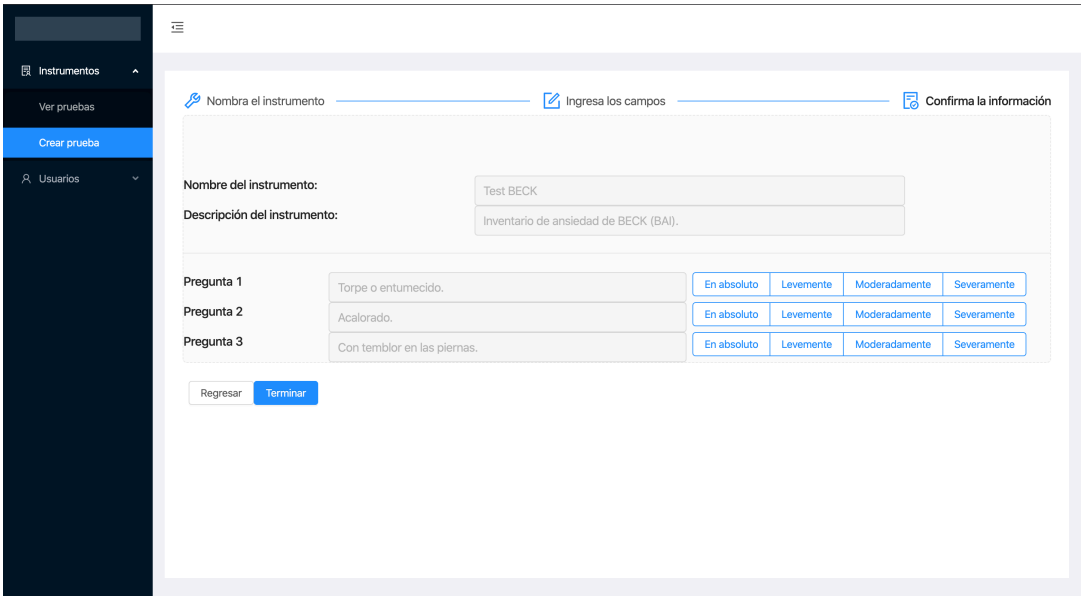

Fig. 2.7. Pantalla de confirmación de un nuevo instrumento.

Para soportar y servir los llamados de la aplicación, se implementó un servidor web en el lenguaje Go con el router HTTP gorilla/mux y el driver de mongo-go-driver. Este servidor se encarga de recibir y responder a todas las solicitudes del cliente, y de interactuar con la base de datos.

Teniendo la aplicación funcional, se empaquetó la aplicación en un contenedor de Docker mediante un DockerFile, el cual compila la aplicación y el servidor, instala todo lo necesario para que funcionen e inicia el scheduler supervisord, que se encarga de manejar estos dos procesos dentro de la aplicación. El contenedor tiene la siguiente estructura.

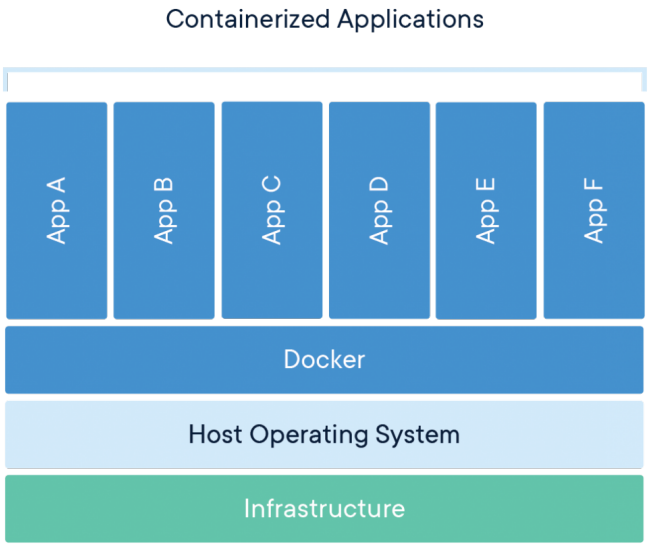

Fig. 2.8. Estructura de un contenedor de Docker.

```
FROM node:lts-alpine AS node-build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY ./ .
RUN npm run build

FROM golang:alpine AS go-build
RUN mkdir /go/src/app
COPY . /go/src/app
WORKDIR /go/src/app
RUN go build -o server .

RUN apk add nginx
RUN mkdir /app
RUN mkdir -p /run/nginx
COPY --from=node-build /app/dist /app
COPY nginx.conf /etc/nginx/nginx.conf

EXPOSE 3000
EXPOSE 8080

RUN apk add supervisor
RUN mkdir -p /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord"]
```

Fig. 2.9. DockerFile para configurar el contenedor.

La aplicación y el servidor se pueden ver como cualquiera de las aplicaciones App A-F en la Fig. 2.5. Se creó una imagen de este contenedor que fue almacenado y publicado en Docker Hub, para facilidad de uso dentro del clúster.

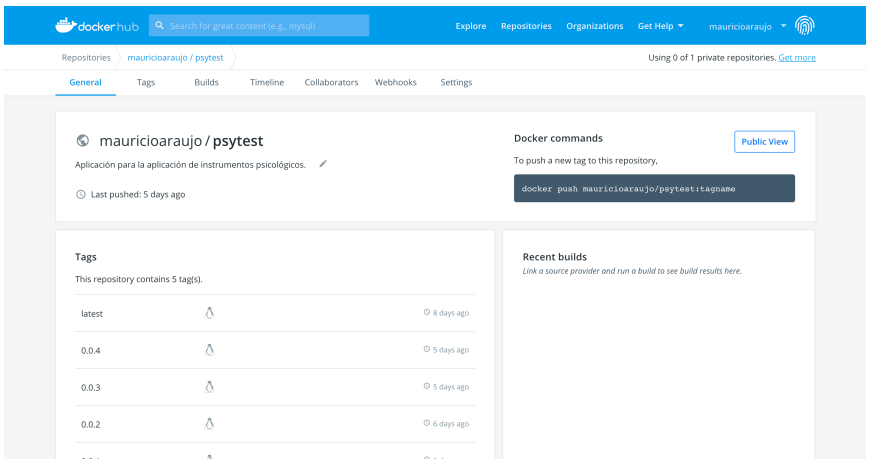

Fig. 2.10. Repositorio dónde se encuentra la imagen del contenedor.

Finalmente, teniendo todos los componentes necesarios, se levantó el clúster de Kubernetes con un único plano de control en tres computadoras diferentes, todas con el sistema operativo de CentOS. Para levantar un clúster, lo primero es crearlo utilizando la herramienta *kubeadm* e iniciando el servicio *kubelet* en cada uno de los nodos. Una vez creado, es necesario instalar una red definida por software (SDN, Software Defined Network) para poder comunicar a los nodos dentro del clúster. El SDN elegido fue Project Calico por su licencia abierta y escalabilidad.

Se probó la comunicación de los nodos, y cuando ésta fue exitosa, se procedió a desplegar el contenedor mediante un Replica Set de Kubernetes. Se creó el despliegue mediante un archivo de configuración YAML, en el que se especificó el número de replicas deseadas (en este caso 3, una por cada nodo). Al crear el despliegue, se levantan múltiples pods (igual al número de replicas especificadas) y Kubernetes se encarga de repartirlos entre los nodos.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: psytest
  labels:
    app: psytest
spec:
  selector:
    matchLabels:
      app: psytest
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: psytest
    spec:
      terminationGracePeriodSeconds: 30
      containers:
      - name: psytest
        image: mauricioaraujo/psytest:0.0.4
        imagePullPolicy: "Always"
        ports:
          - containerPort: 8080
      hostNetwork: true
```

Fig. 2.11. Archivo de configuración para el despliegue.

```
# Configure the MTU to use
veth_mtu: "1440"

# The CNI network configuration to install on each node.  The special
# values in this config will be automatically populated.
cni_network_config: |-
  {
    "name": "k8s-pod-network",
    "cniVersion": "0.3.1",
    "plugins": [
      {
        "type": "calico",
        "log_level": "info",
        "datastore_type": "kubernetes",
        "nodename": "__KUBERNETES_NODE_MAME__",
        "mtu": __CNI_MTU__,
        "ipam": {
            "type": "calico-ipam"
        },
        "policy": {
            "type": "k8s"
        },
        "kubernetes": {
            "kubeconfig": "__KUBECONFIG_FILEPATH__"
        }
      },
      {
        "type": "portmap",
        "snat": true,
        "capabilities": {"portMappings": true}
      }
    ]
  }
```

Fig. 2.12. Fragmento de configuración para el SDN Calico.

Después de confirmar que los pods estén funcionando dentro de cada nodo, se expuso el despliegue mediante un servicio, el cual regresa la dirección IP del clúster para poder acceder a la aplicación. La aplicación dentro del clúster tiene una estructura como lo muestra el siguiente diagrama.

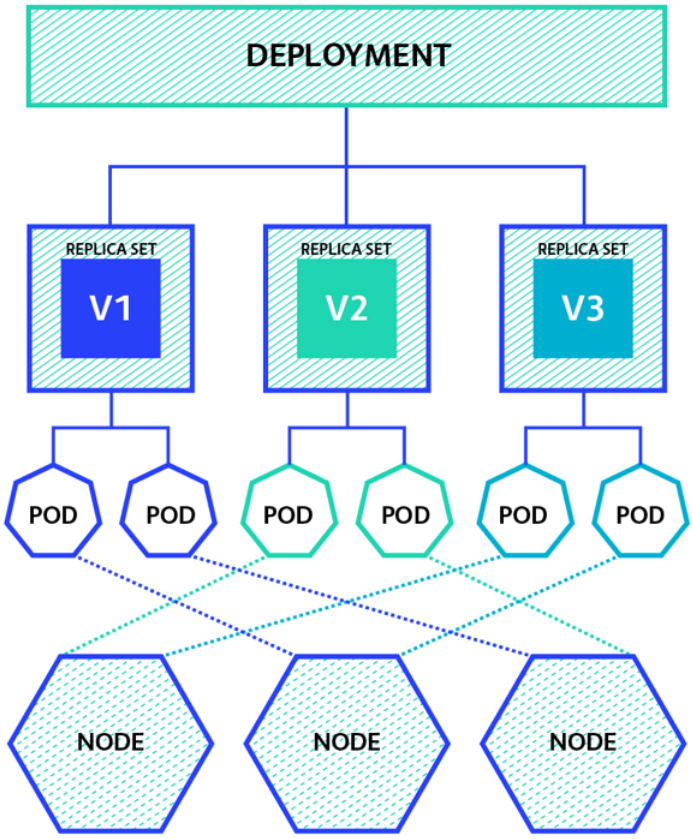

Fig. 2.13. Estructura del despliegue.

Conclusiones

La solución propuesta cumple con los requisitos del problema, y lo hace de una manera eficaz y rápida. Pero esta solución es una prueba de concepto solamente, no es apropiada para un despliegue en producción, ya que no se consideraron muchas configuraciones extras que no son completamente necesarias en un ambiente de desarrollo. Sin embargo, se prueba que una aplicación distribuida transparente, tolerante a fallos, escalable y además amigable para los usuarios es enteramente posible y no está fuera de las posibilidades de ningún estudiante ni desarrollador. Esta tecnología fue desarrollada por las empresas líderes en la industria para solucionar los problemas que el cómputo en la nube presenta y que ahora son herramientas de código abierto accesibles para cualquiera. Es por esto que en el futuro, muy probablemente se verán aplicaciones que utilicen una arquitectura distribuida de este tipo para hacerle frente al creciente número de usuarios.

Referencias

1. **Araujo, M. (2019, Noviembre 26). mauricioaraujo / psytest. Recuperado Diciembre 2, 2019, de https://hub.docker.com/repository/docker/mauricioaraujo/psytest/general.**

2. **Creating a single control-plane cluster with kubeadm. (2019, June 12). Recuperado Diciembre 2, 2019, de https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/.**

3. **Docker Documentation. (n.d.). Recuperado Diciembre 2, 2019, de https://docs.docker.com/.**

4. **Pedersen, B. E. (2018, Mayo 5). Replica Set. Recuperado Diciembre 2, 2019, de https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.**

5. **Perry, S. (2018, Mayo 15). Using Minikube to Create a Cluster. Recuperado Diciembre 2, 2019, de https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/.**

6. **Quickstart for Calico on Kubernetes. (n.d.). Recuperado Diciembre 2, 2019, de https://docs.projectcalico.org/v3.10/introduction/.**

7. **Replication. (2008). Recuperado Diciembre 2, 2019, de https://docs.mongodb.com/manual/replication/.**

8. **Silverlock, M. (2016, Julio 18). gorilla/mux. Recuperado Diciembre 2, 2019, de https://github.com/gorilla/mux.**

9. **Supervisor: A Process Control System. (2004). Recuperado Diciembre 2, 2019, de http://supervisord.org/.**

10. **Trejo, A. (2013, Agosto 25). Clasificación de los instrumentos psicológicos. Recuperado Diciembre 2, 2019, de https://www.cognicionpsicologica.com/clasificacion-de-los-instrumentos-psicologicos/.**

11. **What is REST. (2017). Recuperado Diciembre 2, 2019, de https://restfulapi.net/.**

12. **Wilson, C. (2017, Mayo 26). MongoDB Go Driver. Recuperado Diciembre 2, 2019, de https://github.com/mongodb/mongo-go-driver.**