Escuela Superior de Cómputo
Evaluación a Título de Suficiencia Especial

Sistemas Distribuidos

Prototipo de sistema de venta de boletos para eventos utilizando microservicios sobre Kubernetes en la nube

Resumen

Se desarrollará un prototipo de sistema de venta de boletos para eventos desplegado en la nube, consistente en un front-end HTML-Javascript y un back-end implementado como microservicios Java, accediendo a una instancia de MySQL administrada (PaaS). Cada microservicio ejecutará en Kubernetes.

El servidor web será Nginx y ejecutará en Kubernetes. Los archivos del front-end estarán en un almacenamiento compartido. El servidor web accederá a los archivos del front-end utilizando Persistent Volume (PV) y Persistent Volume Claim (PVC).

El API Gateway será Nginx, el cual recibirá peticiones HTTP y ejecutará en Kubernetes. El API Gateway se expondrá al navegador mediante un servicio de Kubernetes de tipo LoadBalancer.

El API Gateway enrutará las peticiones HTTP a los pods que ejecutarán los microservicios y al servidor web, por tanto, los pods se expondrán al API Gateway utilizando un servicio de Kubernetes de tipo ClusterIP.

No se requiere escalado automático de pods, por tanto, las replicas de los pods se definirán directamente en los manifest files de tipo Deployment.

Requerimientos no funcionales

1. El back-end consistirá de cuatro microservicios Java ejecutando en Kubernetes:

1.1 **Gestión de usuarios**. Microservicio para gestionar los usuarios.
1.2 **Gestión de lugares**. Microservicio para gestionar los lugares donde se llevaran a cabo los eventos.
1.3 **Gestión de eventos**. Microservicio para gestionar los eventos.
1.4 **Gestión de compras**. Microservicio para gestionar la compra de boletos.

2. Se implementará un servidor web mediante Nginx ejecutando en Kubernetes.

3. Se deberá utilizar Kubernetes administrado en la nube (PaaS).

4. Se deberá utilizar MySQL administrado en la nube (PaaS).

5. Los archivos del front-end (.html, .js, .jpeg, .css, etc.) **se deberán** colocar en un almacenamiento compartido (Azure Files, Amazon EFS o Google Filestore). Los archivos del front-end no deberán colocarse en el file system del contenedor que ejecuta el servidor web (Nginx).

6. El front-end se desarrollará en HTML-Javascript. Se podrá utilizar cualquier framework (p.e. React, Angular, Bootstrap, etc.).

7. El back-end consistirá en servicios web Java ejecutando en Kubernetes. Se podrá utilizar cualquier framework para desarrollar los microservicios (p.e. Spring Boot, Azure Functions, JDK, etc.).Todas las peticiones del back-end (GET, POST, PUT y DELETE) regresarán JSON. Las peticiones POST y PUT recibirán los parámetros como JSON en el “request body ”.

8. Cada microservicio tendrá su propia base de datos y no deberá acceder a la base de datos de otro microservicio. Suponiendo que “12345678” es el número de boleta del alumno o alumna:

    8.1 La base de datos para la **gestión de usuarios** se llamará “usuarios_12345678” e incluirá la siguiente tabla:

Tabla: usuarios			
Columna	Tipo	Nulo	Default
id_usuario	int	no	
email	varchar(100)	no	
password	varchar(64)	no	
token	varchar(40)	si	
nombre	varchar(100)	no	
apellidos	varchar(100)	no	
rol	varchar(20)	no	“cliente”

El id_usuario será llave primaria auto-incrementada.

El usuario con el rol de “administrador”, será creado directamente en la base de datos.

Se deberá crear un índice único para el campo “email”.

La columna “password” contiene el hash (sha-256) de la contraseña capturada, en formato hexadecimal. Debido a que sha-256 produce un número de 32 bytes, el password se almacena como 64 caracteres hexadecimales.

8.2 La base de datos para la **gestión de eventos** se llamará “eventos_12345678” e incluirá las siguientes tablas:

Tabla: eventos		
Columna	Tipo	Nulo
id_evento	int	no
nombre	varchar(256)	no
descripcion	varchar(256)	no
id_lugar	int	no
fecha_hora_evento	datetime	no

El id_evento será llave primaria auto-incrementada.

Tabla: fotos_eventos		
Columna	Tipo	Nulo
id_foto	int	no
foto	longblob	no
id_evento	int	no

El id_foto será llave primaria auto-incrementada.

El campo id_evento deberá ser una llave foránea que referencie la llave primaria de la tabla “eventos”.

8.3 La base de datos para la **gestión de lugares** se llamará “lugares_12345678” e incluirá las siguiente tablas:

Tabla: lugares		
Columna	Tipo	Nulo
id_lugar	int	no
nombre	varchar(100)	no
direccion	varchar(256)	no

El id_lugar será llave primaria auto-incrementada.

Tabla: asientos		
Columna	Tipo	Nulo
id_asiento	int	no
id_lugar	int	no
descripcion	varchar(50)	no

El id_asiento será llave primaria auto-incrementada.

El campo id_lugar deberá ser una llave foránea que referencie la llave primaria de la tabla “lugares”.

8.4 La base de datos para la **gestión de compras** se llamará “compras_12345678” e incluirá las siguientes tablas:

Tabla: asientos_evento			
Columna	Tipo	Nulo	Default
id_evento	int	no	
id_asiento	int	no	
precio_boleto	decimal(10,2)	no	
vendido	char(1)	no	N

Se deberá crear una llave primaria compuesta con los campos (id_evento,id_asiento).

El campo “vendido” indica si el asiento se ha vendido (“S”) o no (“N”).

Tabla: carrito_compra		
Columna	Tipo	Nulo
id_evento	int	no
id_asiento	int	no
id_usuario	int	no
precio_boleto	decimal(10,2)	no

Se deberá crear una llave foránea compuesta con los campos (id_evento,id_asiento), que referencie la llave primaria de la tabla “asientos_evento”.

9. **Para cada microservicio** y para el servidor web, se deberá escribir: 1) un archivo Dockerfile para crear la imagen, 2) un archivo manifest file de tipo Deployment para el despliegue de los pods y 2) un archivo manifest file de tipo ClusterIP para el despliegue del servicio que expone el microservicio al API Gateway.

10. Para que el servidor web (Nginx) tenga acceso a los archivos de la aplicación web (front-end), **se deberá** crear un servicio Persistent Volume (PV) y un servicio Persistent Volume Claim (PVC). El PVC deberá definir storageClassName: "" (lo cual indica que el almacenamiento compartido no lo creará automáticamente Kubernetes), por tanto, el almacenamiento compartido **deberá ser creado** por el alumno o la alumna. Los archivos del front-end (html, js, css, jpeg, png, etc.) deberán cargarse en el almacenamiento compartido.

11. El API Gateway será Nginx, el cual deberá ejecutar en Kubernetes y deberá exponer un servicio de Kubernetes de tipo Loadbalancer.

12. El despliegue de los Deployements y Services de Kubernetes se deberá hacer mediante el comando: **kubectl apply -f <manifestfile>**

13. El sistema mostrará las siguientes pantallas:
    - 13.1 **Login**. Pantalla que permite ingresar el email del usuario y la contraseña. Así mismo, en esta pantalla se incluirá una opción para que se pueda registrar un nuevo usuario.
    - 13.2 **Alta de usuario**. Pantalla para se registre el usuario.
    - 13.3 **Menú principal**. Pantalla que muestra al rol “administrador” un menú con las siguientes opciones: “Captura de lugar”, “Captura de evento” y “Compra de boleto”. Al rol “cliente” solo se mostrará la opción “Compra de boleto”.
    - 13.4 **Captura de lugar.** Pantalla para dar de alta un lugar. Esta pantalla solo la podrá acceder el usuario con rol de “administrador”.

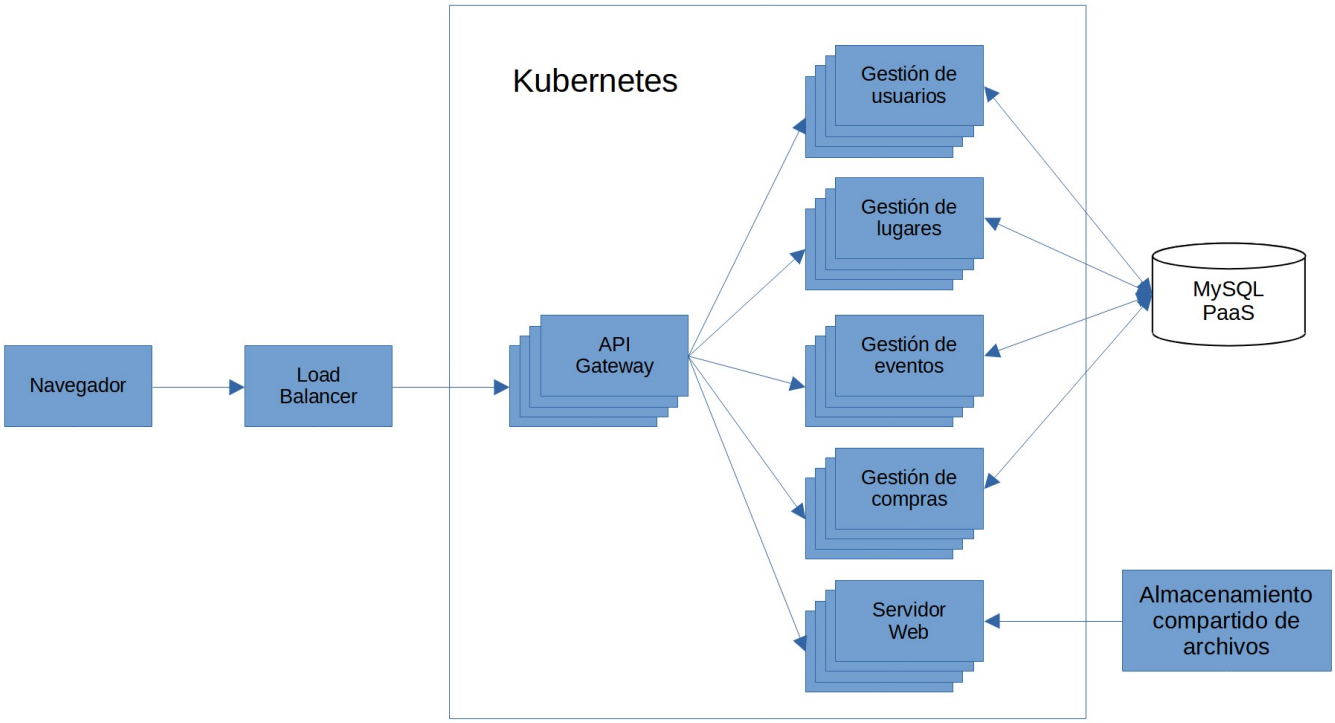
13.5 **Captura de evento**. Pantalla para dar de alta un evento y los asientos para el evento. Esta pantalla solo la podrá acceder el usuario con rol “administrador”.
13.6 **Compra de boleto**. Pantalla para buscar eventos y para comprar boletos.
13.7 **Seleccionar asiento**. Pantalla para seleccionar el asiento para el evento. Esta pantalla muestra los asientos disponibles y los asientos vendidos.
13.8 **Carrito de compra**. Pantalla que muestra los eventos en el carrito de compra.

14. Se podrá utilizar Microsoft Azure, AWS o Google Cloud. La siguiente tabla muestra cómo se llaman los diferentes servicios a implementar, de acuerdo a la plataforma a utilizar:

Servicio	Azure	AWS	Google Cloud
Kubernetes administrado	AKS (Azure Kubernetes Service)	EKS (Elastic Kubernetes Service)	GKE (Google Kubernetes Engine)
Servicio interno entre pods	Serrvicio ClusterIP	Serrvicio ClusterIP	Serrvicio ClusterIP
Exposición externa del API Gateway	Servicio LoadBalancer	Servicio LoadBalancer	Servicio LoadBalancer
Balanceador creado automáticamente	Azure Load Balancer	AWS ELB / NLB	Google Cloud Load Balancer
Almacenamiento compartido de archivos	Azure Files	Amazon EFS	Filestore
Persistent Volume	PV (Azure Files CSI)	PV (EFS CSI)	PV (Filestore CSI)
Persistent Volume Claim	PVC	PVC	PVC
MySQL administrado	Azure Database for MySQL	Amazon RDS for MySQL	Cloud SQL for MySQL
Registro de imágenes de contenedor	Azure Container Registry (ACR)	Amazon ECR	Artifact Registry
Escalado de los microservicios	Replicas en manifest file	Replicas en manifest file	Replicas en manifest file

Arquitectura del sistema

La arquitectura del sistema es la siguiente (notar que podrán existir varias réplicas de los pods que ejecutan el API Gateway, los microservicios y el servidor web):



Requerimientos funcionales del back-end

1. El microservicio "Gestión de usuarios" deberá incluir las siguientes funciones:

- **usuarios/alta_usuario**. Permite al cliente registrarse en el sistema. Recibe los siguientes parámetros: email, password, nombre y apellidos. Notar que el rol default es “cliente”. Si no hay error regresa el código HTTP 200 y el JSON {“mensaje”:”OK”} de otra manera regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. La petición HTTP deberá ser POST.

- **usuarios/login**. Autentica un usuario. Recibe los siguientes parámetros en la URL: email y password (sha-256 de la contraseña capturada). Si el email y el hash de la contraseña se encuentran en la tabla de "usuarios", genera un token aleatorio de 40 caracteres, actualiza el token en la tabla "usuarios" y regresa el código HTTP 200 y el JSON {“id_usuario”: <id_usuario>, ”token”: ”<token>”,”rol”:<”administrador” o “cliente”>}. Si el email y el hash de la contraseña no se encuentran en la tabla "usuarios", regresa el código HTTP 400 y el JSON {“mensaje”: "Acceso denegado"}. La petición HTTP deberá ser GET.

- **usuarios/verifica_acceso**. Verifica que el usuario se haya autenticado (logueado). La función recibe los siguientes parámetros en la URL: id_usuario, token y rol. El parámetro rol podrá ser “administrador”, “cliente” o “” (este último indica cualquier rol).
    - Si rol es “”:
        - Si id_usuario y token existen en la tabla "usuarios", la función regresará el código HTTP 200 y el JSON {“acceso”:true}.
        - Si id_usuario y token no existen en la tabla “usuarios”, entonces la función regresará el código HTTP 200 y el JSON {“acceso”:false}.
    - Si rol no es “”:
        - Si id_usuario, token y rol existen en la tabla "usuarios", la función regresará el código HTTP 200 y el JSON {“acceso”:true}.
        - Si id_usuario, token y rol no existen en la tabla “usuarios”, entonces la función regresará el código HTTP 200 y el JSON {“acceso”:false}.

  Si hay error regresa el código HTTP 400 y el JSON {“mensaje”: "<mensaje de error"}. La petición HTTP deberá ser GET.

2. El microservicio "Gestión de lugares" deberá incluir las siguientes funciones:

- **lugares/alta_lugar.** Da de alta un lugar en la tabla "lugares". La función recibirá los siguientes parámetros: nombre, direccion, id_usuario y token. Para ejecutar la función,

se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “administrador”. Si no hay error, regresa el código HTTP 200 y el JSON {“mensaje”:“OK”} de otra manera regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. La petición HTTP deberá ser POST.

- **lugares/alta_asiento**. Da de alta un asiento en la tabla “asientos”. La función recibirá los siguientes parámetros: id_lugar, descripcion, id_usuario y token. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “administrador”. Si no hay error, regresa el código HTTP 200 y el JSON {“mensaje”:“OK”} de otra manera regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. La petición HTTP deberá ser POST.

- **lugares/obtiene_lugares**. Regresa un arreglo JSON con id_lugar, nombre y dirección de todos los lugares en la tabla de “lugares”. La función recibirá en la URL como parámetros el id_usuario y token. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “”. La petición HTTP deberá ser GET.

- **lugares/obtiene_asientos**. Obtiene de la tabla “asientos” un arreglo JSON con todos los asientos de un lugar. Regresa los campos id_asiento y descripcion. La función recibirá en la URL como parámetros el id_lugar, id_usuario y token. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “administrador”. La petición HTTP deberá ser GET.

3. El microservicio "Gestión de eventos" deberá incluir las siguientes funciones:

- **eventos/alta_evento.** Da de alta un evento en la tabla "eventos". La función recibirá los siguientes parámetros: nombre, descripcion, id_lugar, fecha_hora_evento, id_usuario y token. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “administrador”. Si no hay error, la función regresa el código HTTP 200 y el JSON {“mensaje”:“OK”} de otra manera regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. El INSERT a la tabla “eventos” y el INSERT a la tabla “fotos_eventos” deberá hacerse **dentro de una transacción**. La petición HTTP deberá ser POST. Para crear boletos para el evento, se deberá invocar la función **compras/alta_asiento_evento**, ya que la tabla “asientos_evento” se encuentra en la base de datos del microservicio "Gestión de compras".

- **eventos/consulta_eventos.** Busca una palabra (p.e. “concierto”) en los campos “nombre” y “descripcion” de la tabla “eventos”. Si no hay error, regresa el código HTTP 200 y un arreglo JSON con los datos de los eventos (id_evento, fotografía en base 64,

nombre, descripción, nombre del lugar, dirección del lugar y fecha_hora_evento). Si hay error regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. La búsqueda se deberá realizar utilizando una instrucción SELECT con LIKE. La función recibirá los siguientes parámetros en la URL: palabra (la palabra que se buscará en los campos “nombre” y “descripcion”), id_usuario y token. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “”. La petición HTTP deberá ser GET.

4. El microservicio "Gestión de compras" deberá incluir las siguientes funciones:

- **compras/alta_asiento_evento.** Da de alta un asiento para un evento en la tabla “asientos_evento”. La función recibirá los siguientes parámetros: id_evento, id_asiento, precio_boleto, id_usuario y token. Notar que el campo “vendido” por default es “N”. Si no hay error regresa {“mensaje”:”OK”} de otra manera regresa {“mensaje”:”<mensaje de error>”}. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “administrador”. La petición HTTP deberá ser POST.

- **compras/obtiene_asientos**. Obtiene de la tabla “asientos_evento” un arreglo JSON con todos los asientos para un evento. Regresa los campos id_asiento, precio_boleto y vendido. La función recibirá en la URL como parámetros el id_evento, id_usuario y token. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “”. La petición HTTP deberá ser GET.

- **compras/compra_boleto.** Realiza la compra de un boleto. La función recibirá los siguientes parámetros en la URL: id_evento, id_asiento, precio_boleto, id_usuario y token. Si no hay error, regresa el código HTTP 200 y el JSON {“mensaje”:”OK”}, de lo contrario, regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “”. La petición HTTP deberá ser PUT.
    - Si el asiento ya está vendido, se deberá regresar un código HTTP 400 y el JSON {“mensaje”:”El asiento ya fue vendido"}.
    - Se deberá insertar en la tabla "carrito_compra" el id_usuario, id_evento, id_asiento y precio_boleto, así mismo, se actualizará el campo “vendido” a “S” para el asiento en la tabla “asientos_evento”. El INSERT a la tabla "carrito_compra" y el UPDATE a la tabla "asientos_evento" se deberán realizar **dentro de una transacción**.

- **compras/elimina_asiento_carrito_compra.** Elimina un asiento para un evento de la tabla "carrito_compra". La función recibirá los siguientes parámetros en la URL:

id_evento, id_asiento, id_usuario y token. Si no hay error regresa el código HTTP 200 y el JSON {“mensaje”:”OK”} de otra manera regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “”. La petición HTTP deberá ser DELETE.

- La función deberá eliminar el asiento de la tabla “carrito_compra” y actualizar el campo “vendido” a “N” en la tabla “asientos_evento”.
- La actualización (UPDATE) de la tabla "asientos_evento" y el borrado (DELETE) del asiento de la tabla "carrito_compra", deberán realizarse **dentro de una transacción**.

- **compras/elimina_carrito_compra.** Borra todos los registros del carrito_compra del usuario. La función recibirá los siguientes parámetros en la URL: id_usuario y token. Si no hay error regresa el código HTTP 200 y el JSON {“mensaje”:”OK”} de otra manera regresa el código HTTP 400 y el JSON {“mensaje”:”<mensaje de error>”}. Para ejecutar la función, se deberá verificar el acceso utilizando la función usuarios/verifica_acceso con rol “”. La petición HTTP deberá ser DELETE.
  - La función deberá eliminar cada asiento del usuario de la tabla “carrito_compra”, y actualizar el campo “vendido” del asiento a “N” en la tabla “asientos_evento”.
  - Las actualizaciones (UPDATE) a la tabla "asientos_evento" y el borrado (DELETE) de la tabla "carrito_compra" deberán realizarse **dentro de una transacción**.

Requerimientos funcionales del front-end

1. La pantalla “Login” es la pantalla inicial. Esta pantalla permite capturar el email y la contraseña del usuario, entonces invoca la función “login” del microservicio “Gestión de usuarios”. Se deberá enviar al back-end el hash (SHA-256) de la contraseña (no se deberá enviar al back-end la contraseña capturada).

Si el email y la contraseña son correctos, se guarda en el front.end el id_usuario y el token y se despliega la pantalla “Menú principal”. Si el email o la contraseña no son correctos, se despliega el mensaje “Acceso denegado” (ver la función usuarios/login del back-end).

2. La pantalla “Login” incluye una opción para registrar un nuevo usuario. Cuando se selecciona esta opción, se despliega la pantalla “Alta de usuario”.

3. La pantalla “Alta de usuario” permite capturar los datos del usuario. Se deberá enviar al back-end el hash (SHA-256) de la contraseña (no se deberá enviar al back-end la contraseña capturada) (ver la función usuarios/alta_usuario del back-end).

4. Al seleccionar la opción “Captura de lugar” se deberá desplegar la pantalla “Captura de lugar”, la cual permitirá capturar el nombre de un lugar, la dirección y los asientos del lugar (ver la función lugares/alta_lugar del back-end).

La pantalla “Captura de lugar” permitirá capturar los asientos para el lugar (ver la función lugares/alta_asiento del back-end).

5. Al seleccionar la opción "Captura de evento" se deberá desplegar la pantalla "Captura de evento", la cual permitirá capturar el nombre del evento, la descripción del evento, la fecha, la hora, el precio del boleto, la fotografía del evento (ver la función eventos/alta_evento del back-end). Se deberá seleccionar el lugar de una lista de lugares (ver la función lugares/obtiene_lugares del back-end).

La pantalla “Captura de evento” permitirá capturar los asientos para el evento. Ver la función compras/alta_asiento_evento del back-end. Se deberá seleccionar el asiento de una lista de asientos (ver la función lugares/obtiene_asientos).

6. Al seleccionar la opción "Compra de boleto" se deberá desplegar la pantalla "Compra de boleto". En esta pantalla el usuario podrá ingresar una palabra (p.e. “concierto”) que se buscará en los campos "nombre" y "descripción" de la tabla "eventos" (ver la función eventos/consulta_eventos del back-end). El resultado de la búsqueda se guardará en un arreglo en el front-end para su uso posterior (ver requerimiento funcional 10 del front-end).

7. Para **cada evento** resultado de la búsqueda, se deberá desplegar en la pantalla “Compra de boleto” una pequeña foto del evento, el nombre, la descripción, el lugar, la dirección del lugar, la fecha y la hora del evento, el precio del boleto, un botón “Seleccionar asiento” y un botón de "Compra".

8. Cuando el usuario presione el botón “Seleccionar asiento” se mostrará la ventana “Seleccionar asiento” la cual mostrará los asientos disponibles y los asientos vendidos para el evento; en esta pantalla el usuario podrá seleccionar el asiento (ver la función compras/obtiene_asientos del back-end).

9. Cuando el usuario presione el botón de "Compra", se realizará la compra (ver función compras/compra_boleto del back-end).

10. La pantalla de "Compra de boleto" deberá disponer de un botón "Carrito de compra" el cual deberá desplegar una pantalla "Carrito de compra" con los eventos del usuario en la tabla "carrito_compra", incluyendo una pequeña imagen del evento, nombre del evento, el nombre del lugar, la descripción del asiento y el precio del boleto. Así mismo, en la pantalla "Carrito de compra" se deberá desplegar el total de la compra. El precio del boleto se obtendrá del resultado de la búsqueda de eventos (ver el requerimiento funcional 6 del front-end).

11. Para cada evento en la pantalla "Carrito de compra" se deberá incluir un botón "Eliminar evento" el cual permitirá eliminar el asiento del carrito de compra (ver la función compras/elimina_asiento_carrito_compra del back-end).

12. La pantalla "Carrito de compra" deberá tener un botón "Eliminar carrito" el cual permitirá borrar el carrito de compra (ver la función compras/elimina_carrito_compra del back-end).

13. La pantalla "Carrito de compra" deberá tener un botón "Seguir comprando" el cual deberá permitir regresar a la pantalla "Compra de boleto".

Evaluación del proyecto

° Los alumnos y alumnas deberán presentar su proyecto al profesor evaluador en el laboratorio en el horario que corresponda, matutino o vespertino.

° Para la evaluación del proyecto, se utilizará una rúbrica la cual permitirá verificar que se hayan cumplido los requerimientos funcionales y no funcionales descritos en las especificaciones del proyecto.

° Para presentar su proyecto, los alumnos y las alumnas podrán utilizar su propio equipo de cómputo o una computadora del laboratorio.

° Al finalizar la evaluación del proyecto, los alumnos y alumnas deberán enviar los siguientes archivos al correo electrónico capineda@ipn.mx:

1. Código fuente del front-end (solo archivos con formato texto, es decir, .html, .css, .js, etc.).
2. Código fuente del back-end (solo archivos con formato texto, es decir, .java).
3. Archivo de configuración de Nginx como servidor web.
4. Archivo de configuración de Nginx como API Gateway.
5. Dockerfiles de los microservicios, dockerfile del servidor web y dockerfile del APIGateway.
6. Manifest files de los despliegues (Deployment) de los microservicios, manifest fiile del servidor web y manifest file del APIGateway.
7. Manifest files de los servicios ClusterIP de los microservicios y manifest file del servidor web.
8. Manifest file del servicio LoadBalancer,
9. Manifest file del servicio PV y manifest file del servicio PVC.
10. Scripts de las bases de los microservicios.
11. Un archivo PDF con los diagramas E-R de las bases de datos de los microservicios (obtenidos mediante MySQL Workbench).