«Talento Tech»

Testing QA

Clase 10

Buenos Aires Ciudad

Clase N°10 | Pruebas de API 2

Temario:

- Pruebas de servicios
    - ¿Qué son los microservicios?
- Arquitectura Monolítica vs. Microservicios
- Tipos de prueba API
- ¿Por qué probamos APIs?
- Funciones de APIs
- Implementación de APIs
- Herramientas de APIs
- SOAP & REST
    - ¿Qué es SOAP?
    - ¿Qué es REST?
    - SOAP vs REST
- Herramientas para API Testing

Objetivos de la clase

Hoy en día todos los sistemas se interconectan. Una de las formas más comunes y eficientes de lograrlo es mediante el uso de **servicios web**, que permiten a los sistemas intercambiar datos sin compartir directamente sus bases de datos.

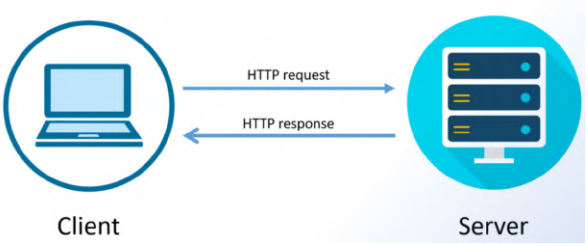
En esta clase veremos qué son los servicios, qué función cumplen en la arquitectura del software moderno y cómo pueden ser probados desde el rol de QA. También aprenderemos las diferencias entre los estilos **SOAP** y **REST**, los tipos de APIs y las herramientas más utilizadas para su testing.

¿Qué son los servicios?

Un servicio es un conjunto de funciones que permite a diferentes aplicaciones comunicarse entre sí a través de internet o redes internas. Estos servicios permiten que los sistemas intercambien datos y ejecuten operaciones sin necesidad de compartir directamente sus bases de datos.

Esta tecnología se basa en dos características fundamentales:

1. **Multiplaforma**: el cliente y el servidor no necesitan tener las mismas configuraciones para comunicarse entre ellos.



2. **Compartición**: en la mayoría de los casos, un servicio web no está disponible solo para una única integración sino para múltiples a través de Internet.

Cuando se utiliza un web service, un cliente envía una solicitud a un servidor, lo que provoca una acción. El servidor, a su vez, envía una respuesta al cliente.

🗨️En la clase anterior vimos **JSON y XML**, que son formatos utilizados para intercambiar datos en servicios. Ahora, vamos a aprender cómo estos datos se transmiten a través de **APIs**.

- **Ejemplo 1:**
  Cuando una aplicación de reservas de hotel consulta disponibilidad en una base de datos central, lo hace a través de un **servicio** que expone los datos necesarios sin comprometer la seguridad de la información interna.

Ejemplo 2:
Una app de reservas hace una solicitud GET a:

```
GET https://api.hoteles.com/habitaciones/disponibles
```

El servicio responde con:

```
{
   "habitaciones":[
      {
         "numero":101,
         "tipo":"Suite",
         "disponible":true
      }
   ]
}
```

Esto permite mostrar en la app solo las habitaciones disponibles, sin necesidad de acceder directamente a la base de datos del hotel.

¿Qué son los microservicios?

Los microservicios son una forma moderna de construir aplicaciones dividiéndolas en pequeños servicios independientes, cada uno con una responsabilidad específica.

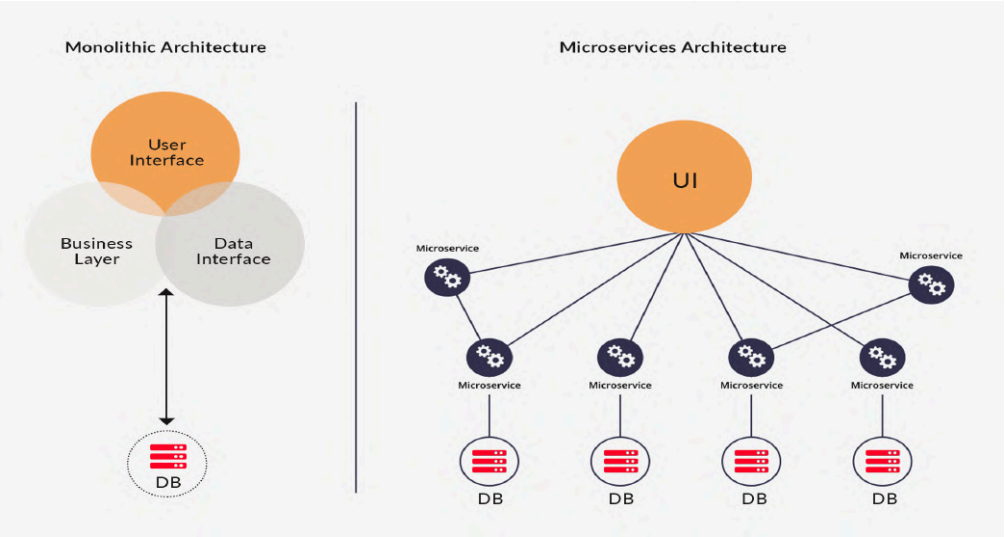
Una de sus principales ventajas es que permiten que cada servicio esté desarrollado en un lenguaje diferente (por ejemplo: uno en Python, otro en Java y otro en C#), siempre que se comuniquen usando protocolos comunes como REST y formatos como JSON.

Esto mejora la flexibilidad del equipo y permite escalar solo los servicios que lo necesiten, sin afectar al resto del sistema. Además, facilita el mantenimiento, la implementación continua y el aislamiento de errores.

Ejemplo: Talento Lab puede tener microservicios separados para autenticación, carga de CVs, postulaciones y recomendaciones.

Arquitectura Monolítica vs. Microservicios

Característica	Monolítica	Microservicios
Escalabilidad	Escala todo el sistema	Escala por módulo
Despliegue	Un único bloque	Independiente por servicio
Mantenimiento	Puede volverse complejo	Más mantenible a largo plazo



La imagen compara dos arquitecturas: **monolítica** (izquierda) y **de microservicios** (derecha).

En la arquitectura **monolítica**, todos los componentes del sistema —interfaz de usuario, lógica de negocio y acceso a datos— están integrados en un solo bloque que se conecta con una única base de datos. Esto hace que cualquier cambio o fallo pueda afectar a toda la aplicación.

En cambio, en la arquitectura de **microservicios**, la interfaz de usuario se comunica con múltiples servicios independientes. Cada microservicio gestiona su propia lógica y base de datos, lo que permite que puedan desarrollarse, actualizarse y escalarse por separado. Además, estos servicios pueden estar escritos en distintos lenguajes y comunicarse entre sí mediante APIs.

Esta separación brinda mayor flexibilidad, escalabilidad y facilidad para aplicar cambios sin afectar al sistema completo.

¿Por qué probamos APIs?

Las pruebas de APIs en QA son fundamentales dentro del contexto de los servicios web, ya que garantizan que la comunicación entre sistemas funcione correctamente y que los datos se transmitan de manera precisa y segura. Dado que los servicios web permiten la integración entre múltiples plataformas a través de formatos de intercambio de datos como JSON y XML, es crucial probar que las APIs procesen correctamente las solicitudes (requests) y devuelvan respuestas (responses) adecuadas. Por ejemplo, en una aplicación de reservas de hotel, un servicio web expone información sobre disponibilidad y precios sin comprometer la base de datos interna. A través de pruebas de API, los testers pueden verificar que al enviar una solicitud con un ID de reserva, la API responde con los datos correctos, asegurando que la integración entre el backend y otros sistemas se realice sin errores y cumpliendo con los requisitos de funcionalidad, seguridad y rendimiento.

Importancia de probar APIs en QA

Las pruebas de APIs permiten detectar errores antes de que una aplicación se integre completamente con un frontend. Son esenciales para asegurar que:

- La API **responde correctamente** según las solicitudes.
- Se **envían y reciben datos correctamente** en JSON o XML.
- Se respetan los estándares de seguridad y rendimiento.

Ejemplo práctico
En una API de reservas de hotel:

- Una prueba funcional verificará si al enviar un **ID de reserva**, la API devuelve los detalles correctos.
- Una prueba de carga verificará si la API **maneja múltiples solicitudes sin fallar**.

Tipos de APIs

Existen diferentes tipos de APIs dependiendo de su acceso y uso:

Tipo	Descripción	Ejemplo
APIs Públicas	Abiertas al público para uso general.	API de Google Maps
APIs Privadas	Solo accesibles dentro de una organización.	API interna de un banco
APIs de Socios	Compartidas con terceros autorizados.	API de pagos de PayPal
APIs Compuestas	Combinan múltiples servicios en una única API.	API de comercio electrónico que usa pagos + envíos

¿Cómo se estructura una API REST?

Para que una API REST funcione correctamente, debe seguir una estructura organizada que permita la comunicación entre cliente y servidor. Esta estructura se compone de cuatro elementos clave:

1. Endpoint (URL de acceso)

Un **endpoint** es la dirección (URL) a la que se envían las solicitudes para interactuar con un recurso específico en la API. Representa el punto de acceso a una funcionalidad determinada.

Ejemplo: En una API de reservas de hotel, el endpoint para obtener las habitaciones disponibles podría ser:

```
https://api.hotelreservas.com/habitaciones/disponibles
```

Importancia: Permite acceder a los recursos de manera estructurada y predecible.

2. Método HTTP (GET, POST, PUT, DELETE)

Las APIs REST utilizan diferentes **métodos HTTP** para definir el tipo de operación que se realizará sobre el recurso.

Método	Descripción	Ejemplo en API de hotel
GET	Solicita datos de un recurso.	Obtener la lista de habitaciones disponibles.
POST	Crea un nuevo recurso.	Realizar una nueva reserva de habitación.
PUT	Actualiza un recurso existente.	Modificar los datos de una reserva existente.
DELETE	Elimina un recurso.	Cancelar una reserva.

Ejemplo de uso (solicitud GET)
Para obtener información sobre una reserva en la API:

```
GET https://api.hotelreservas.com/reservas/12345
```

Importancia: Define la acción a ejecutar y evita modificar datos accidentalmente al utilizar el método incorrecto.

3. Encabezados HTTP (Autenticación, tipo de contenido)

Los **encabezados HTTP (Headers)** proporcionan información adicional sobre la solicitud o la respuesta, como la autenticación o el formato de los datos.

Encabezados comunes:

- `Authorization`: Usado para autenticar la solicitud (ejemplo: tokens de acceso).
- `Content-Type`: Indica el formato de los datos enviados (ejemplo: `application/json`).
- `Accept`: Especifica qué tipo de respuesta espera el cliente.

Ejemplo de encabezados en una solicitud:

```
GET /reservas/12345 HTTP/1.1
Host: api.hotelreservas.com
Authorization: Bearer abcdef123456
Content-Type: application/json
Accept: application/json
```

Importancia: Garantiza seguridad (mediante autenticación) y compatibilidad entre cliente y servidor.

4. Cuerpo de la solicitud (JSON o XML con los datos)

El **cuerpo (Body)** de una solicitud contiene los datos necesarios para realizar ciertas operaciones en la API, especialmente en métodos como `POST` y `PUT`.

Ejemplo de cuerpo en JSON para crear una reserva:

```
{
  "cliente": {
    "nombre": "Juan Pérez",
    "telefono": "+54 11 1234-5678",
    "email": "juanperez@email.com"
  },
  "habitacion": {
    "numero": 305,
    "tipo": "Suite",
    "precio_por_noche": 120.50
  },
  "fecha_entrada": "2024-04-10",
  "fecha_salida": "2024-04-15"
}
```

Importancia: Permite enviar datos estructurados de manera eficiente y legible.

SOAP (Simple Object Access Protocol)

Es un protocolo estructurado basado en XML, que define estrictas reglas de comunicación.

Características:

- Usa **XML** como formato de datos.
- Opera sobre **HTTP, SMTP, TCP, entre otros**.
- Permite mayor seguridad y transacciones complejas.

Ejemplo de una solicitud SOAP

```
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header/>
   <soap:Body>
      <ObtenerReserva>
         <IdReserva>12345</IdReserva>
      </ObtenerReserva>
   </soap:Body>
</soap:Envelope>
```

Ejemplo de solicitud SOAP para consultar una reserva

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header/>
   <soap:Body>
      <ConsultarReserva>
         <IdReserva>12345</IdReserva>
      </ConsultarReserva>
   </soap:Body>
</soap:Envelope>
```

Ejemplo de respuesta SOAP (XML)

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Body>
      <ConsultarReservaResponse>
         <Reserva>
            <IdReserva>12345</IdReserva>
            <Cliente>Juan Pérez</Cliente>
            <Estado>Confirmada</Estado>
         </Reserva>
      </ConsultarReservaResponse>
   </soap:Body>
</soap:Envelope>
```

Ventajas de XML en SOAP:

- Estructura más robusta para validaciones.
- Permite definir **esquemas XML** (XSD) para garantizar integridad de datos.
- Se usa en sistemas empresariales que requieren mayor seguridad.

Uso en QA: Podemos validar la estructura XML con herramientas como **XMLValidator o XMLint** (que vimos la clase pasada) y asegurarnos de que cumple con los esquemas esperados.

REST (Representational State Transfer)

Es un estilo de arquitectura más flexible y liviano que SOAP, utilizando principalmente **JSON** para intercambiar datos.

Características:

- Utiliza **HTTP** como base de comunicación.
- Usa **JSON** o XML para estructurar respuestas.
- Es más rápido y fácil de implementar que SOAP.

Ejemplo de una solicitud REST (JSON)

```
{
  "idReserva": "12345"
}
```

En la clase anterior aprendimos sobre **estructuras JSON y XML**. Ahora, veremos cómo estos formatos son utilizados en **SOAP y REST** para comunicar datos.

JSON en APIs REST

REST es un estilo arquitectónico que permite la comunicación entre aplicaciones usando **HTTP**. La mayoría de las APIs REST utilizan JSON para estructurar sus respuestas y solicitudes.

Ejemplo de una solicitud REST con JSON

Solicitud HTTP (POST) para crear una reserva de hotel

```
POST /reservas HTTP/1.1
Host: api.hoteles.com
Content-Type: application/json
Authorization: Bearer abc123
```

Cuerpo de la solicitud (JSON)

```
{
  "cliente": "Juan Pérez",
  "habitacion": 305,
  "fecha_entrada": "2024-04-10",
  "fecha_salida": "2024-04-15"
}
```

Ventajas de JSON en APIs REST:

- Menos pesado que XML (ocupa menos espacio en la red).
- Fácil de leer y escribir.
- Compatible con múltiples lenguajes de programación.

Ejemplo de respuesta JSON de la API

```
{
  "reserva_id": "12345",
  "estado": "Confirmada"
}
```

Uso en QA: Podemos validar esta respuesta en Postman asegurándonos de que la API devuelve los datos correctos.

SOAP vs REST: Comparación Final

Característica	SOAP	REST
Formato de datos	Solo XML	JSON y XML
Protocolos soportados	HTTP, SMTP, TCP	Solo HTTP
Velocidad	Más lento	Más rápido
Facilidad de uso	Más complejo	Más simple
Seguridad	Mejor seguridad integrada	Seguridad basada en estándares externos

Herramientas para probar APIs

Herramienta	Descripción
Postman	Permite probar, automatizar y documentar APIs.
SoapUI	Herramienta para pruebas de servicios SOAP y REST.
cURL	Línea de comandos para hacer peticiones HTTP.
Newman	Permite ejecutar pruebas Postman desde la terminal.

¡Trabajando en Talento Lab!

Silvia y Matías te convocan nuevamente al equipo de QA. En esta etapa del proyecto, Talento Lab está incorporando **microservicios** para separar responsabilidades clave: gestión de usuarios, carga de CVs y publicación de ofertas laborales.

Silvia te explica que el equipo backend definió una arquitectura REST para facilitar las integraciones, pero también hay módulos heredados que siguen usando SOAP.

“Vamos a necesitar validar ambos tipos de servicios. Y como QA, tenemos que asegurarnos de que todo funcione correctamente, sin importar si usan XML o JSON.”

Matías te pide que investigues las diferencias entre SOAP y REST, y prepares un pequeño informe comparativo para presentarlo en la próxima daily. También te desafía a analizar ejemplos reales de respuestas XML y JSON, identificando su estructura.

“No te preocupes si no podés hacer pruebas aún. Esta clase es para entender la base. La semana que viene empezamos con Postman.”

Ejercicio práctico

Analizar y comparar dos tipos de respuestas de servicios: una en formato XML (SOAP) y otra en JSON (REST).

Instrucciones:
Examina detenidamente los ejemplos de respuestas SOAP y REST proporcionados

1. Identifica las diferencias en estructura, formato y legibilidad
2. Considera qué formato preferirías validar como tester y por qué
3. Reflexiona sobre cómo estas diferencias afectan tu enfoque de pruebas

Ejemplo SOAP:

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
     <Reserva>
        <Id>12345</Id>
        <Estado>Confirmada</Estado>
     </Reserva>
  </soap:Body>
</soap:Envelope>
```

Ejemplo REST (respuesta JSON):

```
{
   "id": "12345",
   "estado": "Confirmada"
 }
```

Preguntas para reflexionar

- ¿Por qué es importante probar APIs antes de integrar el frontend?
- ¿En qué casos puede fallar una API sin que lo note el usuario final?
- ¿Qué beneficios aporta una arquitectura basada en microservicios?
- ¿Qué diferencias funcionales hay entre SOAP y REST?
- ¿Por qué es útil conocer el tipo de API con la que trabajaremos?

Próximos pasos

En la próxima clase vamos a dar el primer paso hacia la automatización de pruebas de APIs, utilizando la herramienta **Postman**. Aprenderás cómo enviar solicitudes, validar respuestas y simular distintos escenarios.

Recordá traer instalada la aplicación, porque haremos los primeros ejercicios prácticos en clase.


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad