Sistemas Distribuidos

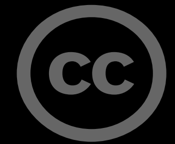
Bloque I

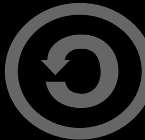
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.1:

Introducción a los Sistemas Distribuidos

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Índice

- ¿Qué es un Sistema distribuido?
- Características
- Ventajas
- Desafíos
- Arquitecturas
- Comunicación
- Tolerancia a fallos, recuperación y seguridad
- Aplicaciones actuales

¿Qué es un Sistema Distribuido?

- **Definición básica:** Conjunto de computadoras independientes que se presentan al usuario como una única computadora coherente.

- **Objetivo principal:** Conectar recursos y usuarios de manera transparente.

Características

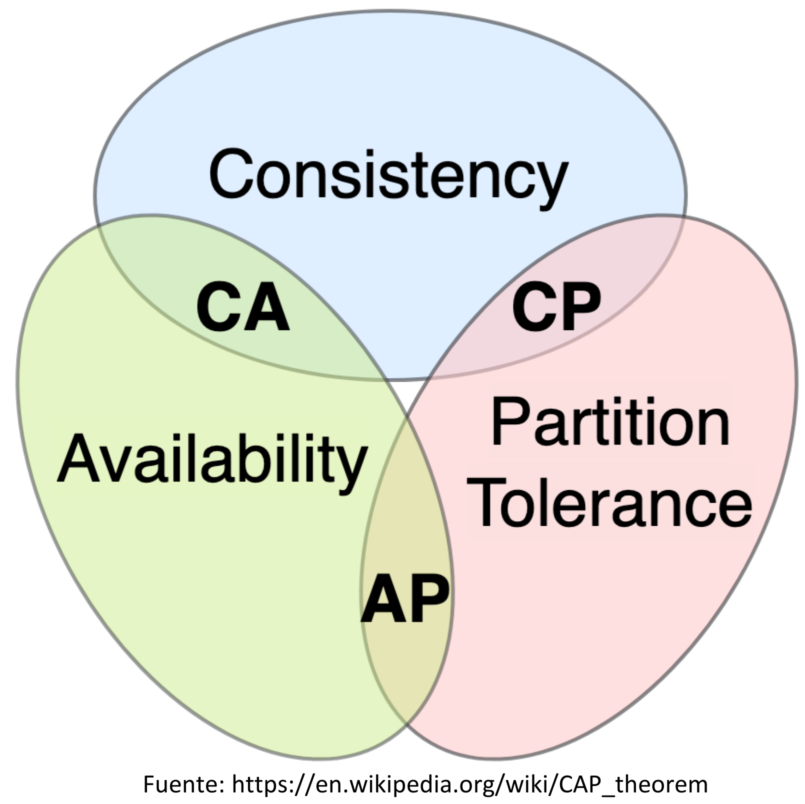
- **Concurrencia**
  - Varios procesos pueden correr simultáneamente en diferentes nodos.
  - Es necesario coordinar acciones y evitar conflictos.
- **Escalabilidad**
  - Capacidad para crecer y gestionar un aumento en usuarios y tareas.
  - Puede ser vertical (añadir más recursos a un nodo) o horizontal (añadir más nodos).
- **Fallos independientes**
  - Un fallo en un nodo no debería causar la caída del sistema completo.
  - La detección y recuperación son esenciales.
- **Heterogeneidad**
  - Diversidad de hardware, sistemas operativos, redes y aplicaciones.
  - El middleware ayuda a unificar esta diversidad.

Ventajas

- Compartir recursos
- Tolerancia a fallos y alta disponibilidad
- Escalabilidad
- Flexibilidad y adaptabilidad
- Reducción de costes y aumento de rendimiento

Desafíos

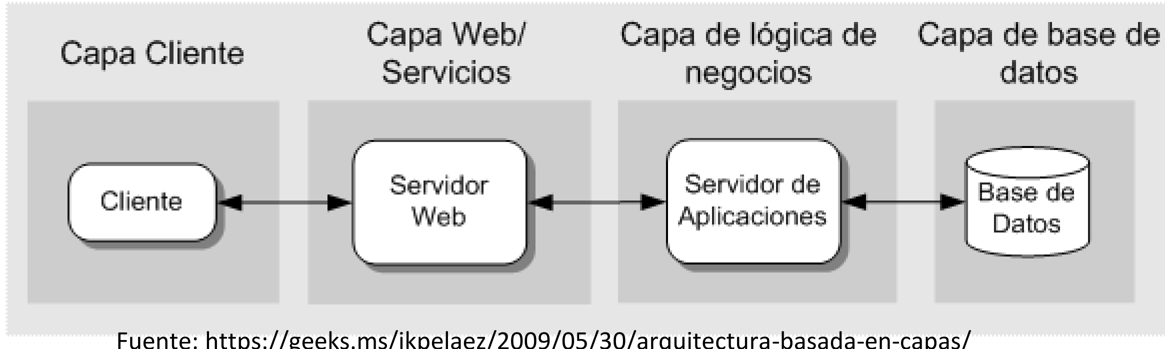
- Coordinación y comunicación entre nodos
- Gestión de fallos y recuperación
- Seguridad y privacidad
- Consistencia y replicación
- Diseño y desarrollo



Fuente: https://en.wikipedia.org/wiki/CAP_theorem

Arquitecturas

- **Modelo Cliente-Servidor**
  - Un servidor provee recursos o servicios, y los clientes consumen esos servicios.
  - Variantes:
    - Grueso delgado (Dependiendo de dónde resida la lógica de la aplicación).
    - P2P (Cada nodo puede ser tanto cliente como servidor).
- **Arquitectura en Capas**
  - Capas separan las responsabilidades y simplifican la gestión y el desarrollo.
  - OSI Model con sus 7 capas desde física hasta aplicación.



Fuente: https://geeks.ms/jkpelaez/2009/05/30/arquitectura-basada-en-capas/

- **Arquitectura Basada en Objetos**
  - **Objetos distribuidos**: Entidades que encapsulan datos y métodos y pueden ser invocados remotamente.
  - **CORBA** (Arquitectura de Objetos de Broker Común) es un estándar que permite la comunicación entre objetos en diferentes sistemas.
  - **Java RMI**: Permite a un objeto invocar métodos de otro objeto en una máquina distinta.

Comunicación

- **Protocolos y Mensajes**:
  - HTTP, FTP, SMTP son ejemplos de protocolos.
  - Un protocolo define las reglas para el intercambio de mensajes.
- **Invocación Remota**:
  - **RPC (Remote Procedure Call)**: Permite que programas llamen a procedimientos en otra máquina.
  - **RMI (Remote Method Invocation)**: Similar al RPC pero orientado a objetos.
- **Middleware**: Software que proporciona servicios para ayudar a las aplicaciones distribuidas a comunicarse y gestionar datos.
  - Ejemplo: **Message Queue** (cola de mensajes) para la comunicación asíncrona entre aplicaciones.

Tolerancia a fallos, recuperación y seguridad

- **Tipos de fallos**:
  - Fallo de nodo, fallo de comunicación, fallo de software, etc.
- **Técnicas de recuperación**:
  - **Check-pointing**: Guardar el estado de una aplicación en un punto específico.
  - **Logging**: Registrar las operaciones para poder "reproducirlas" en caso de fallo.
- **Técnicas de cifrado:**
  - Simétrico (mismo cifrado y descifrado de clave) y Asimétrico (clave pública y privada).
- **Certificados:**
  - Proporcionados por una Autoridad de Certificación, garantizan la identidad de un nodo.

Aplicaciones actuales – Cloud Computing

- **Definición:** Entrega de servicios informáticos a través de Internet.
- **Tipos:** IaaS, PaaS, SaaS.
- **Proveedores líderes:** Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure.
- **Aplicaciones:** Almacenamiento en la nube, bases de datos en la nube, Inteligencia Artificial, Machine Learning.

Aplicaciones actuales – BBDD

- **Definición:** Sistemas que almacenan datos en múltiples ubicaciones, pero son tratados como una única entidad.
- **Ventajas:** Tolerancia a fallos, alta disponibilidad, escalabilidad.
- **Ejemplos:** Cassandra, MongoDB, CockroachDB.
- **Aplicaciones:** Comercio electrónico, servicios financieros, plataformas sociales.

Aplicaciones actuales – Blockchain

- **Definición:** Registro distribuido y cifrado de transacciones.
- **Características:** Transparente, inmutable, seguro.
- **Ejemplos:** Bitcoin, Ethereum.
- **Aplicaciones:** Monedas digitales, contratos inteligentes, registros de tierras.

Aplicaciones actuales – Sistemas de archivos

- **Definición:** Almacenamiento de datos que permite el acceso concurrente desde múltiples nodos.
- **Ventajas:** Tolerancia a fallos, escalabilidad, eficiencia.
- **Ejemplos:** Hadoop Distributed FileSystem (HDFS), Google File System (GFS).
- **Aplicaciones:** Análisis de big data, almacenamiento de backup, streaming de contenidos.

Aplicaciones actuales – Otras

- **IoT (Internet de las cosas):** Dispositivos conectados que recopilan y comparten datos.
- **Microservicios:** Aplicaciones descompuestas en servicios más pequeños e independientes.
- **Computación Edge:** Procesamiento de datos cerca de la fuente de datos, como dispositivos IoT.
- **Redes Sociales:** Servicios como Facebook y Twitter que operan en plataformas distribuidas globales.

Sistemas Distribuidos

Bloque I

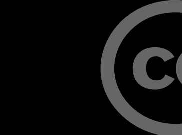
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

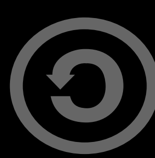
Tema 1.1:

Introducción a los Sistemas Distribuidos

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
"Atribución-CompartirIgual 4.0 Internacional" de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

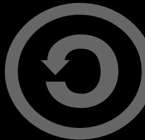
Bloque I

Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.2:

Introducción a la web

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **Historia temprana**
- Nacimiento de HTML, CSS y JavaScript
- Evolución de la web (1990-2000)
- La era de la web 2.0 (2000-2010)
- La web moderna (2010-actualidad)
- Desafíos

Historia temprana

- **1989: Propuesta inicial**: Sir Tim Berners-Lee, un científico británico en CERN, propone un sistema para automatizar el intercambio y la consulta de documentos entre científicos.
- **1990: Nacimiento de la WWW**: Se introduce el término "World Wide Web". Berners-Lee crea el primer navegador y servidor web en una computadora NeXT.

Historia temprana

- **1991: Primer sitio web y lanzamiento al público**: El CERN publica información sobre el proyecto World Wide Web, y el primer sitio web es lanzado, explicando qué es la WWW y cómo navegarla.
- **1992: Expansión y primeros navegadores**: Se desarrollan y lanzan navegadores como ViolaWWW y MidasWWW. La idea de la web comienza a ganar tracción.

Nacimiento de HTML, CSS y JavaScript

- **HTML (1990)**:
  - Lenguaje de marcado para estructurar contenido en la web.
- **CSS (1996)**:
  - Introducido para separar el diseño y la presentación de la estructura del documento.
- **JavaScript (1995)**:
  - A pesar de las críticas iniciales, rápidamente se convirtió en un pilar de la web, permitiendo la creación de contenido dinámico.

Evolución de la web (1990-2000)

- **1993: Mosaic**: Lanzado por el National Center for Supercomputing Applications (NCSA), Mosaic es reconocido por popularizar la web gracias a su interfaz gráfica amigable.
- **1994: Netscape Navigator**: Surgido de los creadores de Mosaic, Netscape Navigator se convierte en el navegador líder y es el precursor del nacimiento de JavaScript.
- **1995: Microsoft lanza Internet Explorer**: Esto marca el comienzo de la "guerra de navegadores". Microsoft y Netscape luchan por la supremacía, lo que lleva a rápidas innovaciones (y desviaciones de estándares).

Evolución de la web (1990-2000)

- **1996-1999: Estándares y consolidación**: La W3C (World Wide Web Consortium) trabaja en estandarizar HTML, CSS y otros aspectos clave de la web. Internet Explorer se establece como el navegador dominante.
- **1999: Web 2.0**: Aunque el término se populariza más tarde, los conceptos centrales de la Web 2.0, como la participación del usuario y las aplicaciones web, comienzan a emerger en este período.

La era de la web 2.0 (2000-2010)

- **Concepto de Web 2.0**: Evolución de sitios estáticos a aplicaciones web interactivas.
- **Redes Sociales**: MySpace, luego Facebook y Twitter. Cambian la forma en que nos conectamos y compartimos.
- **AJAX**: Técnica que permite actualizar páginas web sin recargarlas. Gmail es un ejemplo pionero de esta tecnología.
- **Inicio de la Movilidad**: Con el lanzamiento del iPhone en 2007, comienza la necesidad de sitios web optimizados para dispositivos móviles.

La web moderna I (2010-presente)

- **Diseño responsive**: CSS3 y frameworks como Bootstrap permiten que los sitios web se adapten a cualquier tamaño de pantalla.

- **Single Page Applications (SPA)**: Frameworks de JavaScript como React, Angular y Vue permiten la creación de aplicaciones web rápidas y fluidas que cargan una sola vez y luego actualizan dinámicamente el contenido.

La web moderna II (2010-presente)

- **Web APIs y Web Components**: Permiten la creación de aplicaciones más modularizadas y conectadas.

- **Progressive Web Apps (PWAs)**: Combinan lo mejor de la web y las aplicaciones móviles. Permiten funcionalidades offline, notificaciones y un rendimiento superior.

- **HTTP/2**: Nuevo protocolo para la web que mejora la velocidad y la eficiencia de las conexiones.

Desafíos

- **Seguridad**: A medida que la web se vuelve más poderosa, también se vuelve un objetivo mayor para los ciberdelincuentes. SSL/TLS, CORS y otros mecanismos de seguridad se vuelven esenciales.

- **Privacidad**: Con el auge de las redes sociales y las aplicaciones en línea, la privacidad de los datos es una preocupación. Regulaciones como el GDPR se implementan.

Desafíos

- **Optimización y rendimiento**: La necesidad de cargar rápidamente y ofrecer experiencias fluidas, especialmente en áreas con conexiones lentas o en dispositivos menos potentes.
- **Desinformación**: La web como plataforma para noticias falsas y polarización. Necesidad de herramientas y educación para discernir información.

Sistemas Distribuidos

Bloque I

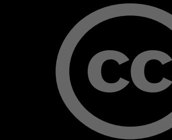
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

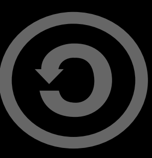
Tema 1.2:

Introducción a la web

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
"Atribución-CompartirIgual 4.0 Internacional" de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

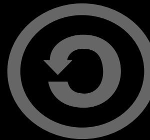
Bloque I

Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.3:

HTML

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **¿Qué es HTML?**
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- Etiquetas comunes
- Atributos
- Tablas
- iframe y entidades
- Bloques, comentario y salto de línea
- Formularios

¿Qué es HTML?

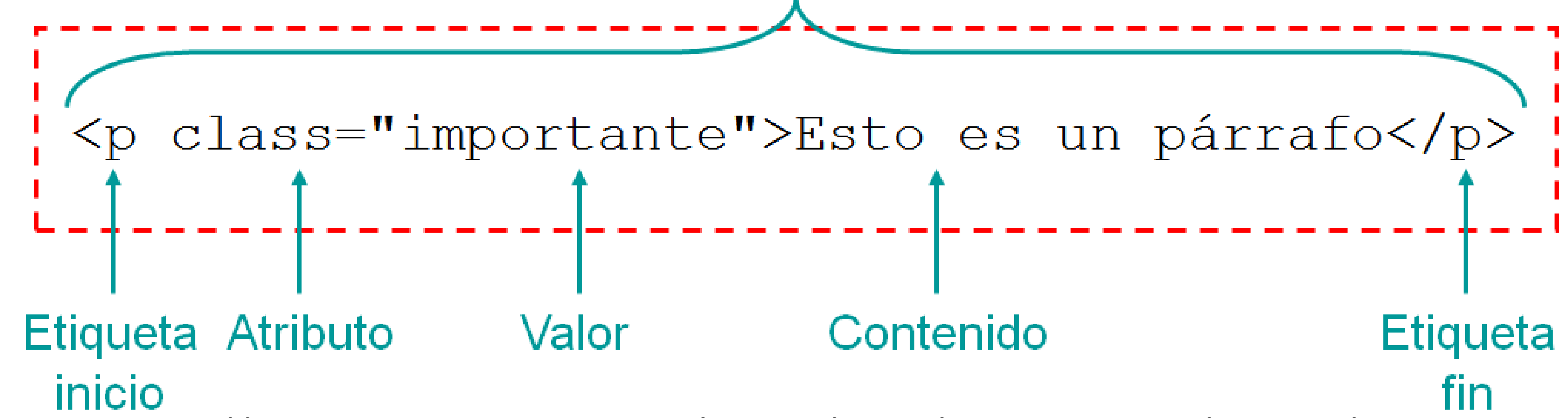
- **HyperText Markup Language**: Lenguaje de marcado utilizado para estructurar contenido en la web.
- **Lenguaje de marcado**: Utiliza "etiquetas" para determinar cómo se debe presentar el contenido.
- Todos los sitios web utilizan HTML para presentar su contenido.

Índice

- ¿Qué es HTML?
- **Componentes básicos de HTML**
- Estructura básica de un documento HTML
- Etiquetas comunes
- Atributos
- Tablas
- iframe y entidades
- Bloques, comentario y salto de línea
- Formularios

Componentes básicos de HTML

- Etiquetas (tags): Indican el inicio y el final de un elemento (p.ej., <p></p>).
- Atributos: Propiedades adicionales de un elemento (p.ej., href en <a>).
- Contenido: Información entre las etiquetas (ej., texto en <p>)



Fuente: http://desarrolloweb.dlsi.ua.es/cursos/2011/html5-css3-es/images/html-sintaxis.png

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- **Estructura básica de un documento HTML**
- Etiquetas comunes
- Atributos
- Tablas
- iframe y entidades
- Bloques, comentario y salto de línea
- Formularios

Estructura básica de un documento HTML

```
<!DOCTYPE html>
<html>
    <head>
        <title>Page Title</title>
    </head>
    <body>
    <h1>This is a Heading</h1>
    <p>This is a paragraph.</p>
    </body>
</html>
```

Estructura básica de un documento HTML

- <!DOCTYPE html>: Declaración del tipo de documento.
- <html>: Raíz del documento.
- <head> y <body>: Secciones principales; <head> contiene metainformación y <body> el contenido visible.
- Elementos básicos: Como <title>, <meta>, <h1>, <p>.

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- **Etiquetas comunes**
- Atributos
- Tablas
- iframe y entidades
- Bloques, comentario y salto de línea
- Formularios

Etiquetas comunes

- **Encabezados** (<h1>, <h2>,... <h6>):

<h1> es el más importante y <h6> el menos importante.

- **Párrafos** (<p>): Texto regular en el contenido.
- **Enlaces** (<a>): Conectan a otras páginas o recursos.
- **Imágenes** (<img>): Insertan imágenes en el documento.
- **Listas**: No ordenadas (<ul>) con <li> y ordenadas (<ol>) con <li>.

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- Etiquetas comunes
- **Atributos**
- Tablas
- iframe y entidades
- Bloques, comentario y salto de línea
- Formularios

Atributos

- Propiedades que definen características adicionales para un elemento. Sintaxis:
  - Se coloca siempre dentro de la etiqueta de inicio.
  - Formato: nombre="valor".
- Atributo class: Permite asignar una o más clases a un elemento. Útil para estilizar o seleccionar elementos con CSS o JavaScript.
  - Ejemplo: <div class="contenedor principal"></div>.
- Atributo id: Asigna un identificador único a un elemento. Útil para referenciar elementos específicamente.
  - Ejemplo: <div id="encabezado"></div>.

Atributos

- Atributo src: Especifica la fuente o dirección de recursos como imágenes, videos o scripts.
  - Ejemplo: <img src="imagen.jpg">.
- Atributo href: Indica el destino o URL de un enlace.
  - Ejemplo: Ejemplo.
- Atributo alt: Proporciona una descripción textual para elementos como imágenes, útil para accesibilidad.
  - Ejemplo: <img src="imagen.jpg" alt="Descripción de la imagen">.
- Atributos width y height: Determinan el ancho y alto de elementos como imágenes o iframes.
  - Ejemplo: <img src="imagen.jpg" width="300" height="200">.

Ejercicios I

- Genera los 6 encabezados con distinto nivel con tus profes favoritos
- Añade una descripción a cada uno de ellos
- Añade un enlace a su perfil en la URJC (por ejemplo: https://gestion2.urjc.es/pdi/ver/nicolas.rodriguez)

- Genera una lista ordenada con tus 3 Pokémon favoritos.
- Añade una foto de uno de ellos. La foto tiene que llevar a su ficha en la Pokédex

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- Etiquetas comunes
- Atributos
- **Tablas**
- iframe y entidades
- Bloques, comentario y salto de línea
- Formularios

Tablas

- Sirve para la representación de datos en formato tabla.
- Elementos esenciales: <table>, <tr>, <td>, <th>.
- Además de <thead>, <tbody>, <tfoot>
- Atributos adicionales: rowspan y colspan para combinar celdas.

Tablas

```
<table>
  <tr>
    <th>Encabezado 1</th>
    <th>Encabezado 2</th>
    </tr>
  <tr>
    <td>Dato 1</td>
    <td>Dato 2</td>
  </tr>
</table>
```

Encabezado 1	Encabezado 2
Dato 1	Dato 2

Ejercicios II

- Genera una lista ordenada con tus 3 Pokémon favoritos.
- Añade una foto de uno de ellos. La foto tiene que llevar a su ficha en la Pokédex
- Todo lo anterior en formato tabla

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- Etiquetas comunes
- Atributos
- Tablas
- **iframe y entidades**
- Bloques, comentario y salto de línea
- Formularios

iframe y entidades

- <iframe>: Permite incrustar contenido de otra página, como mapas o videos.

- Uso principal: Representar caracteres que tienen significado en HTML.
- Ejemplos comunes: < | > | &.
- Importancia: Evitar errores de interpretación en el navegador.

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- Etiquetas comunes
- Atributos
- Tablas
- iframe y entidades
- **Bloques, comentario y salto de línea**
- Formularios

Bloques, comentario y salto de línea

- <div> para bloques y <span> para contenido en línea.
- Versatilidad: Ambas etiquetas se usan comúnmente para aplicar estilos y estructurar contenido.
- <!-- Esto es un comentario -->.
-
 para saltos de línea y <hr> para separación temática.

Índice

- ¿Qué es HTML?
- Componentes básicos de HTML
- Estructura básica de un documento HTML
- Etiquetas comunes
- Atributos
- Tablas
- iframe y entidades
- Bloques, comentario y salto de línea
- **Formularios**

Formularios

- Los formularios en HTML permiten recopilar información del usuario. Se componen de diferentes elementos para permitir la entrada de datos, como campos de texto, áreas de texto, casillas de verificación, botones de opción, listas desplegables y botones, entre otros.
- <form>: Es el contenedor principal para todos los elementos del formulario. Sus atributos más comunes son:
  - action: URL a la que se envían los datos del formulario cuando el usuario lo envía.
  - method: Método HTTP utilizado para enviar los datos, comúnmente "GET" (datos adjuntos a la URL) o "POST" (datos en el cuerpo de la solicitud).

Formularios

- <input>: Elemento versátil que puede tener diferentes "tipos" (type), como:
  - text: Para entrada de texto.
  - password: Campo de contraseña.
  - radio: Botón de opción.
  - checkbox: Casilla de verificación.
  - submit: Botón para enviar el formulario.
  - file: Para cargar archivos.
  - Entre otros

Formularios

- <textarea>: Para entrada de texto multilínea.
- <select> y <option>: Para listas desplegables. <select> es el contenedor y <option> representa cada opción en la lista.
- <button>: Representa un botón y puede tener diferentes roles según el atributo type.
- <label>: Proporciona una descripción o etiqueta para un elemento de entrada.

Ejercicio III

- Genera un formulario con los siguientes campos:
  - Nombre
  - Contraseña
  - Pronombre preferido (desplegable)
  - Intereses (desplegable)
  - Botón “Registrar”
- Genera un salto de línea
- Añade un iframe con algún vídeo de YouTube

Herramientas de desarrollo

- Inspeccionar elementos: Ver y modificar el HTML en tiempo real.
- Editar HTML: Cambiar el código directamente desde el navegador.
- Visualizar estilos y scripts: Acceder a estilos CSS asociados y scripts JavaScript.

Integración con CSS y JavaScript

- Estilización: Hacer que el HTML se vea atractivo y se comporte de manera adaptativa.
- Buena práctica: Preferir estilos externos para separar contenido y presentación.
- Interactividad: JavaScript permite que el contenido HTML responda a las acciones del usuario.
- Eventos: Acciones como onclick, onload que pueden disparar funciones JavaScript.
- Manipulación del DOM: Cambiar, agregar o eliminar contenido HTML en tiempo real.

Sistemas Distribuidos

Bloque I

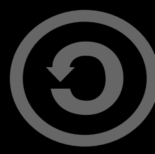
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.3:

HTML

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
“Atribución-CompartirIgual 4.0 Internacional” de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Sistemas Distribuidos

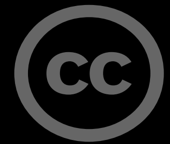
Bloque I

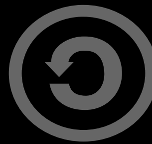
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.4:

CSS

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Índice

- **¿Qué es CSS?**
- ¿Cómo se incluye CSS?
- Estilo de texto
- Posicionamiento
- Modelo de caja
- Frameworks

¿Qué es CSS?

- CSS (Cascading Style Sheets) es un lenguaje utilizado para describir la apariencia y formato de un documento escrito en HTML.

Mi Página de Perfil

Sobre Mí

Texto sobre ti, tus intereses, formación, etc.

Habilidades

- HTML
- CSS

Experiencia

Logo Empresa

Descripción de la experiencia laboral.

Derechos reservados - Tu Nombre

Índice

- ¿Qué es CSS?
- **¿Cómo se incluye CSS?**
- Estilo de texto
- Posicionamiento
- Modelo de caja
- Frameworks

¿Cómo se incluye CSS?

- Inline: Directamente en un elemento HTML usando el atributo style.

```
<section id="about">
    <h2 style='color:red' >Sobre Mí</h2>
    <p>Texto sobre ti, tus intereses, formación, etc.</p>
</section>
```

- Internal: Dentro de la etiqueta <style> en el <head> del documento HTML.

```
<style>
    h2{
        color:red;
    }
</style>
```

- External: Enlazando a una hoja de estilos externa mediante la etiqueta <link>.

```
<link rel="stylesheet" href="style.css">
```

Mi Página de Perfil

Sobre Mí

Texto sobre ti, tus intereses, formación, etc.

Habilidades

- HTML
- CSS

Experiencia

Logo Empresa

Descripción de la experiencia laboral.

Derechos reservados - Tu Nombre

¿Cómo se incluye CSS?

- ID: Estiliza un elemento con un ID específico (ej: #miID).
- Clase: Estiliza todos los elementos con una clase específica (ej: .miClase).

Mi Página de Perfil

Sobre Mí

Texto sobre ti, tus intereses, formación, etc.

Habilidades

- HTML
- CSS

Experiencia

Logo Empresa

Descripción de la experiencia laboral.

Derechos reservados - Tu Nombre

Mi Página de Perfil

Sobre Mí

Texto sobre ti, tus intereses, formación, etc.

Habilidades

- HTML
- CSS

Experiencia

Logo Empresa

Descripción de la experiencia laboral.

Derechos reservados - Tu Nombre

Índice

- ¿Qué es CSS?
- ¿Cómo se incluye CSS?
- **Estilo de texto**
- Posicionamiento
- Modelo de caja
- Frameworks

Estilo del texto

- CSS permite un control detallado sobre el estilo del texto en las páginas web. Estas propiedades afectan cómo se muestra el texto, mejorando la legibilidad y el atractivo visual.
- font-size:
  - Define el tamaño del texto.
  - Ejemplo: font-size: 16px; hace que el texto tenga un tamaño de 16 píxeles.
- font-family:
  - Establece la tipografía del texto.
  - Se pueden especificar varias fuentes, y el navegador utilizará la primera que esté disponible.
  - Ejemplo: font-family: Arial, sans-serif;.

Estilo del texto

- text-align:
  - Alinea el texto dentro de su contenedor.
  - Valores comunes incluyen left, right, center, y justify.
  - Ejemplo: text-align: center; alinea el texto al centro.
- color:
  - Cambia el color del texto.
  - Puede especificarse con valores HEX, RGB, nombres de colores, etc.
  - Ejemplo: color: #000000; para texto en color negro.
- line-height:
  - Ajusta la altura de línea del texto, afectando el espaciado vertical entre líneas.
  - Puede mejorar la legibilidad, especialmente en bloques de texto.
  - Ejemplo: line-height: 1.5; para 1.5 veces el tamaño de la fuente.

Índice

- ¿Qué es CSS?
- ¿Cómo se incluye CSS?
- Estilo de texto
- **Posicionamiento**
- Modelo de caja
- Frameworks

Posicionamiento

- CSS ofrece múltiples propiedades para controlar cómo se colocan y organizan los elementos en una página web. Position (define cómo un elemento es posicionado en el documento):
  - Static (predeterminado): El elemento se coloca según el flujo normal del documento.
  - relative: El elemento se posiciona relativo a su posición original.
  - absolute: El elemento se posiciona relativo a su contenedor más cercano posicionado (no static).
  - fixed: El elemento se posiciona relativo a la ventana del navegador, permaneciendo fijo durante el desplazamiento.
  - sticky: Una mezcla entre relative y fixed; el elemento se "pega" durante el scroll en un punto específico.

Posicionamiento

- top, right, bottom, left:
  - Usados para posicionar el elemento cuando se utiliza position: relative, absolute, fixed o sticky.
  - Especifican la distancia desde los bordes del contenedor de posicionamiento.
- float y clear:
  - float: Permite que los elementos floten a la derecha o izquierda, con el contenido fluyendo a su alrededor.
  - clear: Controla el flujo de elementos flotantes, impidiendo que elementos floten a uno o ambos lados.

Índice

- ¿Qué es CSS?
- ¿Cómo se incluye CSS?
- Estilo de texto
- Posicionamiento
- **Modelo de caja**
- Frameworks

Modelo de caja

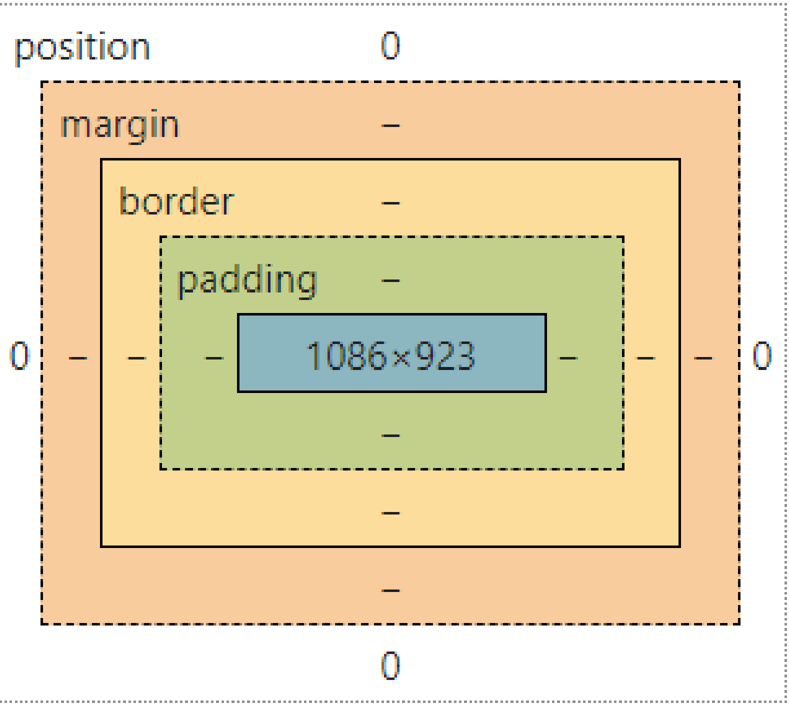
- En CSS, cada elemento se representa como una caja rectangular.
- Content (Contenido): Es el área donde se muestra el contenido del elemento, como texto o imágenes. Su tamaño se puede ajustar mediante las propiedades width y height.
- Padding: Espacio entre el contenido y el borde. El padding aumenta el tamaño total del elemento, pero no el del contenido.

Modelo de caja

- Border (Borde): Rodea el padding y el contenido. Su grosor también añade al tamaño total del elemento.
- Margin (Margen): Espacio exterior alrededor del borde. El margin no afecta el tamaño del elemento en sí, pero sí influye en el espacio que ocupa en la página.
- Esencial para el diseño y la disposición de los elementos en una página web. Afecta al posicionamiento y al espaciado entre elementos.

Modelo de caja

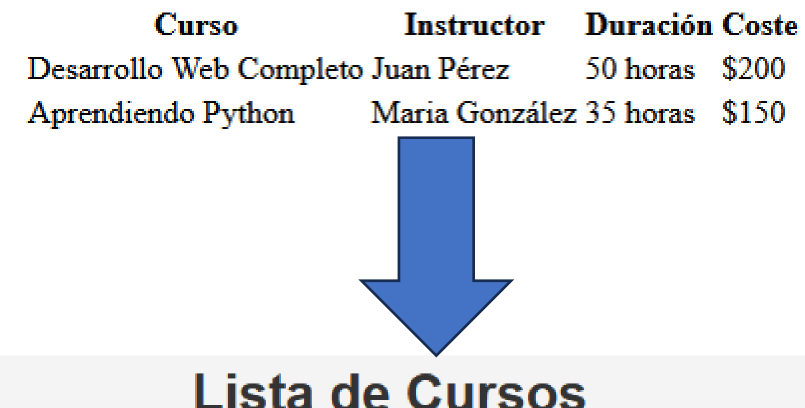
- Chrome – F12 -Computed



Ejercicio

- Pasar de un diseño a otro

Lista de Cursos



Curso	Instructor	Duración	Coste
Desarrollo Web Completo	Juan Pérez	50 horas	$200
Aprendiendo Python	Maria González	35 horas	$150

Lista de Cursos

Curso	Instructor	Duración	Coste
Desarrollo Web Completo	Juan Pérez	50 horas	$200
Aprendiendo Python	Maria González	35 horas	$150

Índice

- ¿Qué es CSS?
- ¿Cómo se incluye CSS?
- Estilo de texto
- Posicionamiento
- Modelo de caja
- **Frameworks**

Frameworks

- Los frameworks CSS son bibliotecas pre-escritas de CSS que ayudan a los desarrolladores a construir páginas web de manera eficiente y coherente. Ofrecen un conjunto de estilos predefinidos, componentes y estructuras de diseño que se pueden utilizar para desarrollar rápidamente interfaces de usuario atractivas y responsive.
- Bootstrap:
  - Uno de los frameworks más populares y ampliamente utilizados.
  - Basado en un sistema de grid, incluye componentes para navegación, formularios, botones, y mucho más.
  - Fácil de personalizar y altamente responsive.

Frameworks

- Foundation:
  - Conocido por ser altamente personalizable.
  - Ofrece un enfoque más profesional y complejo que Bootstrap.
  - Incluye un sistema de grid avanzado y componentes móviles-first.
- Tailwind CSS:
  - Ofrece un enfoque más bajo nivel y requiere una curva de aprendizaje más pronunciada.
  - Altamente personalizable y flexible.
- Materialize:
  - Basado en Material Design de Google.
  - Ofrece una amplia gama de componentes con animaciones y transiciones elegantes.
  - Fácil de usar y con un diseño estético atractivo.

Sistemas Distribuidos

Bloque I

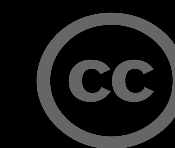
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

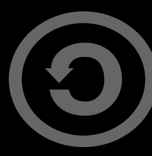
Tema 1.4:

CSS

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

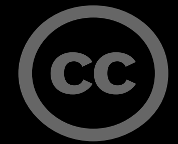
Bloque I

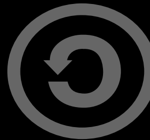
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.5:

JavaScript

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Índice

- **Definición y origen**
- Características principales
- Control de flujo
- Tipos de datos
- Operadores
- Funciones
- Objetos
- DOM y BOM
- Eventos

Definición y origen

- JavaScript es un lenguaje de programación interpretado, de alto nivel, que se utiliza principalmente para scripts en páginas web.
- Fue creado por Brendan Eich en 1995 y originalmente se llamaba "Mocha", luego "LiveScript", y finalmente "JavaScript".
- A pesar de su nombre, JavaScript no está relacionado con Java y fue desarrollado independientemente.

Índice

- Definición y origen
- **Características principales**
- Control de flujo
- Tipos de datos
- Operadores
- Funciones
- Objetos
- DOM y BOM
- Eventos

Características principales

- Es un lenguaje de programación dinámico que soporta estilos de programación orientada a objetos, imperativa y funcional.
- Se ejecuta principalmente en el navegador, aunque también se puede usar en el servidor (por ejemplo, Node.js).
- Es conocido por su capacidad para agregar interactividad a las páginas web, mejorar la experiencia del usuario y crear aplicaciones web modernas.

Características principales

- JavaScript es uno de los tres lenguajes fundamentales en el desarrollo web, junto con HTML y CSS.
- Ha jugado un papel crucial en la evolución de la web, desde páginas estáticas hasta aplicaciones web dinámicas y complejas.
- Su popularidad y adopción han crecido enormemente, y ahora es una herramienta indispensable para cualquier desarrollador web.

Características principales

- Un script JavaScript típico se compone de declaraciones que se ejecutan de arriba hacia abajo.
- Cada declaración suele terminar con un punto y coma (;), aunque este es opcional debido al "Automatic Semicolon Insertion" en JavaScript.
- Declaración de variables:
  - Palabras clave var, let y const para la declaración de variables.
  - var para variables con scope de función
  - let para variables con scope de bloque
  - const para valores constantes.

```
let mensaje = "Hola, mundo!";
console.log(mensaje);
```

Índice

- Definición y origen
- Características principales
- **Control de flujo**
- Tipos de datos
- Operadores
- Funciones
- Objetos
- DOM y BOM
- Eventos

Control de flujo

```
let edad = 20;
if (edad < 18) {
	console.log("Eres menor de edad.");
} else if (edad < 60) {
	console.log("Eres adulto.");
} else {
	console.log("Eres adulto mayor.");
```

```
let nombres = ["Ana", "Luis", "Carlos", "Marta"];
for (let i = 0; i < nombres.length; i++) {
	console.log(nombres[i]);
}
```

```
let numero = 8;
while (numero > 1) {
	console.log(numero);
	numero = numero / 2;
}
```

```
let dia = "Martes";
switch (dia) {
	case "Lunes": console.log("Inicia la semana de trabajo.");
	break;
case "Martes":
case "Miércoles":
case "Jueves":
	console.log("En medio de la semana laboral.");
	break;
case "Viernes":
	console.log("Casi es fin de semana!");
	break;
case "Sábado":
case "Domingo":
	console.log("Es fin de semana, a disfrutar!");
	break;
default: console.log("No es un día válido.");
```

Índice

- Definición y origen
- Características principales
- Control de flujo
- **Tipos de datos**
- Operadores
- Funciones
- Objetos
- DOM y BOM
- Eventos

Tipos de datos

- Primitivos:
  - Number: Representa tanto enteros como flotantes. Ejemplo: let edad = 25;.
  - String: Cadenas de caracteres, ya sea en comillas simples o dobles. Ejemplo: let nombre = "Alice";.
  - Boolean: Representa valores de verdad (true o false). Ejemplo: let estaActivo = true;.
  - Undefined: Indica una variable que no ha sido asignada. Ejemplo: let resultado;.
  - Null: Representa una ausencia intencional de cualquier valor de objeto. Ejemplo: let respuesta = null;.
- Complejos:
  - Object: Colecciones de propiedades.
    - Ejemplo: let persona = {nombre: "Alice", edad: 25};.
  - Array: Lista ordenada de datos.
    - Ejemplo: let numeros = [1, 2, 3, 4, 5];.

Índice

- Definición y origen
- Características principales
- Control de flujo
- Tipos de datos
- **Operadores**
- Funciones
- Objetos
- DOM y BOM
- Eventos

Operadores

- Aritméticos: Usados con valores numéricos para realizar operaciones matemáticas comunes, como +, -, *, /, y % (módulo).
- Asignación: Asignan un valor a una variable. El más simple es =, pero también incluyen operadores compuestos como +=, -=, *=, y /=.
- Comparación: Comparan dos valores y retornan un booleano. Incluyen == (igualdad), === (igualdad estricta), != (desigualdad), !== (desigualdad estricta), >, <, >=, <=.
- Lógicos: Utilizados para determinar la lógica entre variables o valores. Incluyen && (y), || (o), ! (no).
- String: Concatenación (+)

Operadores

- La diferencia entre igualdad (==) e igualdad estricta (===) en JavaScript es una distinción importante y es fundamental :
- Igualdad (==)
  - Descripción: La igualdad, o "igualdad abstracta", compara dos valores por su igualdad después de convertir ambos valores a un tipo común.
  - Conversión de Tipo: Si los dos valores tienen tipos diferentes, JavaScript intenta convertirlos a un tipo común antes de hacer la comparación, lo que se conoce como "coerción de tipo". Ejemplo:
    - 0 == '0' evalúa como true porque el string '0' se convierte en el número 0 antes de la comparación.
    - null == undefined también es true ya que ambos se consideran ausencia de valor.
- Igualdad Estricta (===)
  - Descripción: La igualdad estricta, o "igualdad de tipo", compara tanto el valor como el tipo de los dos operandos, sin realizar la conversión de tipo.
  - Sin Conversión de Tipo: Si los dos valores tienen tipos diferentes, la comparación es automáticamente false. Ejemplo:
    - 0 === '0' es false porque aunque el valor numérico es el mismo, los tipos son diferentes (número vs. string).
    - null === undefined es false ya que null y undefined son tipos diferentes.

Índice

- Definición y origen
- Características principales
- Control de flujo
- Tipos de datos
- Operadores
- **Funciones**
- Objetos
- DOM y BOM
- Eventos

Funciones

- Las funciones son fundamentales en JavaScript para organizar y reutilizar código.
- Existen las funciones anónimas, que no tienen nombre y generalmente se usan como argumento de otras funciones.
- Las arrow functions (introducidas en ES6) tienen una sintaxis más corta para escribir funciones.

```
function saludo(nombre) {
        return `Hola, ${nombre}!`; }
console.log(saludo("Alice"));
```

```
document.getElementById('miBoton’).
addEventListener('click', function() {
 alert('¡El botón fue pulsado!'); });
```

```
const suma = (a, b) => a + b;
 console.log(suma(5, 3));
```

Funciones

- Existen también las closures functions, como funciones que recuerdan el entorno en el que fueron creadas, lo que permite técnicas avanzadas como encapsulación y fabricación de funciones.

```
function crearSaludo(saludo) {
		return function(nombre) {
				return `${saludo}, ${nombre}!`;
				}
		}
const saludoPersonal = crearSaludo("Hola");
console.log(saludoPersonal("Alice"));
```

Índice

- Definición y origen
- Características principales
- Control de flujo
- Tipos de datos
- Operadores
- Funciones
- **Objetos**
- DOM y BOM
- Eventos

Objetos

- Un objeto es una colección de propiedades, y una propiedad es una asociación entre un nombre (o clave) y un valor.
- El valor de una propiedad puede ser una función, en cuyo caso la propiedad es conocida como un método.

```
let persona = {
        nombre: "Alice",
        edad: 25,
        saludar: function() {
                console.log(`Hola, mi nombre es ${this.nombre}`);
        }
 };
persona.saludar();
```

Listas

- Colección “ordenada” de valores, que pueden ser de cualquier tipo, incluidos otras listas y objetos.
- Métodos comunes para manipulación de lista: push, pop, shift, unshift, map, filter, reduce.

```
let frutas = ['manzana', 'banana', 'cereza’];
frutas.push('naranja’);
console.log(frutas);
// ['manzana', 'banana', 'cereza', 'naranja']
```

```
let frutas = ["manzana", "banana", "cereza"];
console.log(frutas[1]);
// Accede al segundo elemento: banana
```

```
let numeros = [2, 3, 4]; numeros.unshift(1);
console.log(numeros);
// [1, 2, 3, 4]
```

```
let numeros = [1, 2, 3, 4, 5];
let cuadrados = numeros.map(num => num * num);
console.log(cuadrados);
// [1, 4, 9, 16, 25]
```

Índice

- Definición y origen
- Características principales
- Control de flujo
- Tipos de datos
- Operadores
- Funciones
- Objetos
- **DOM y BOM**
- Eventos

DOM y BOM

- El DOM es una interfaz de programación que representa los documentos HTML y XML, como una estructura de árbol, donde cada nodo es un objeto que representa una parte del documento.
- JavaScript puede utilizar el DOM para manipular el contenido, la estructura y el estilo de los documentos web.
- El BOM es una representación del navegador que incluye objetos para trabajar con todo lo que no forma parte del documento HTML en sí, como window, navigator, screen, location, history.
- El BOM permite interactuar con el navegador, incluyendo manipulación de ventanas, detección de características del navegador y más.

DOM y BOM

- Ejemplo 1: Cambiando el contenido de un elemento

```
// Cambiando el contenido de un párrafo
document.getElementById('miParrafo').textContent = 'Texto actualizado!';
```

- Ejemplo 2: Añadiendo un nuevo elemento

```
// Creando un nuevo elemento y añadiéndolo al DOM
let nuevoDiv = document.createElement('div');
nuevoDiv.textContent = '¡Soy un nuevo div!';
document.body.appendChild(nuevoDiv);
```

- Ejemplo 3: Modificando estilos

```
// Cambiando el color de fondo de un elemento
document.getElementById('miDiv').style.backgroundColor = 'lightblue';
```

DOM y BOM

- Ejemplo 4: Añadiendo y quitando clases CSS

```
// Añadiendo una clase CSS a un elemento
document.getElementById('miElemento').classList.add('mi-clase-css’);
// Quitando una clase CSS
document.getElementById('miElemento').classList.remove('otra-clase-css');
```

- Ejemplo 5: Manejador de eventos

```
// Añadiendo un evento click a un botón
let miBoton = document.getElementById('miBoton');
miBoton.addEventListener('click', function() {
		alert('¡Botón pulsado!’);
});
```

- Ejemplo 6: Obteniendo el valor de un campo de formulario

```
// Obteniendo el valor de un campo de texto
let valor = document.getElementById('miCampoTexto').value;
console.log(valor);
```

Índice

- Definición y origen
- Características principales
- Control de flujo
- Tipos de datos
- Operadores
- Funciones
- Objetos
- DOM y BOM
- **Eventos**

Eventos

- Los eventos en JavaScript son acciones o sucesos que ocurren en la página web, como clics, pulsaciones de teclas, movimientos del mouse, etc.
- JavaScript permite responder a estos eventos mediante el uso de funciones.
- Evento1: primer argumento es el tipo de evento y el segundo es la función que se ejecuta cuando ocurre el evento.

```
document.getElementById('miBoton').addEventListener('click', function() {
console.log('Botón pulsado');
```

- Evento2: Prevención del comportamiento predeterminado de eventos usando event.preventDefault().

```
document.getElementById('miEnlace').addEventListener('click',
function(event) { event.preventDefault();
```

Eventos

- Ejemplo 1: Cambiar el estilo de un elemento en un evento

```
document.getElementById('miBoton').addEventListener('mouseover', function() {
        this.style.backgroundColor = 'lightblue'; });
document.getElementById('miBoton').addEventListener('mouseout', function() {
this.style.backgroundColor = ''; });
```

- Ejemplo 2: Validar un formulario antes de enviar

```
document.getElementById('miFormulario').addEventListener('submit',
function(event) {
        let campo = document.getElementById('miCampoTexto').value;
        if (campo.length < 3) {
                alert('El texto ingresado es demasiado corto.');
                event.preventDefault();
        }
});
```

Ejercicio 1

- Desarrollar una página web que permita al usuario añadir elementos a una lista y verlos reflejados en tiempo real en la página. Se pide:
  - Estructura HTML:
    - Crea un archivo HTML con un formulario que incluya un campo de texto (input) y un botón (button) para añadir elementos a la lista.
    - Incluye un elemento ul vacío donde se añadirán los nuevos elementos de la lista (li).
  - JavaScript para manejar eventos:
    - Script que "espere" el evento 'click' en el botón de añadir.
    - Cuando se haga clic en el botón, el script deberá leer el valor del campo de texto y añadirlo como un nuevo elemento li dentro del ul.
  - Validación:
    - Asegúrate de que el campo de texto no esté vacío antes de añadir un elemento a la lista. Si está vacío, muestra una alerta o mensaje de error al usuario.
    - Limpiar el campo de texto, después de añadir un elemento a la lista, limpia el campo de texto para que esté listo para la siguiente entrada.

Ejercicio 2

- Desarrollar una página web que permita al usuario cambiar el tema de color de la página (por ejemplo, de claro a oscuro) utilizando JavaScript y el DOM. Se pide:
  - Estructura HTML:
    - Crea un archivo HTML con un botón o interruptor (switch) que el usuario pueda hacer clic para cambiar el tema de color de la página.
    - Incluye varios elementos en la página, como un encabezado, un párrafo y un área de contenido, para que el cambio de tema sea evidente.
  - CSS para tema:
    - Define dos temas de color en tu archivo CSS: uno claro y otro oscuro. Puedes hacerlo definiendo clases que cambien propiedades como background-color y color.
  - JavaScript para cambiar temas:
    - Escribe un script que escuche el evento 'click' en el botón de cambio de tema.
    - Cuando se haga clic en el botón, el script deberá alternar entre las clases de tema claro y oscuro en los elementos relevantes de la página.

Sistemas Distribuidos

Bloque I

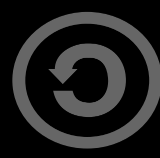
Introducción a los Sistemas Distribuidos: caracterización y arquitecturas.

Tema 1.5:

JavaScript

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Sistemas Distribuidos

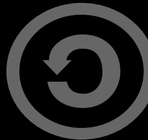
Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.1:

Spring Framework

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **Definición y origen**
- Características principales
- Ecosistema
- MVC
- Maven
- Configuración de un proyecto básico
- Ejemplo

Definición y origen

- Spring Boot es un framework que se utiliza para desarrollar aplicaciones basadas en Java.
- Funciona como una extensión del ya conocido Spring Framework, pero con un enfoque en simplificar el proceso de configuración y despliegue de aplicaciones.
- Es ideal para construir tanto aplicaciones web como servicios de backend.

spring BOOT

Índice

- Definición y origen
- **Características principales**
- Ecosistema
- MVC
- Maven
- Configuración de un proyecto básico
- Ejemplo

Características principales

- Reducción de Configuración:
  - Uno de los principales atractivos de Spring Boot es su capacidad para minimizar la configuración requerida para arrancar una aplicación Spring.
- Enfoque en la Convención:
  - Spring Boot sigue el principio de "convenio sobre configuración", lo que significa que intenta adivinar la configuración que necesitas en función de las bibliotecas que tienes en tu classpath.
- Relación con Spring:
  - Ofrece una configuración predeterminada que ayuda a reducir el tiempo de arranque y desarrollo.
- Ventaja adicional:
  - Capacidad de crear aplicaciones independientes.

Índice

- Definición y origen
- Características principales
- **Ecosistema**
- MVC
- Maven
- Configuración de un proyecto básico
- Ejemplo

Ecosistema

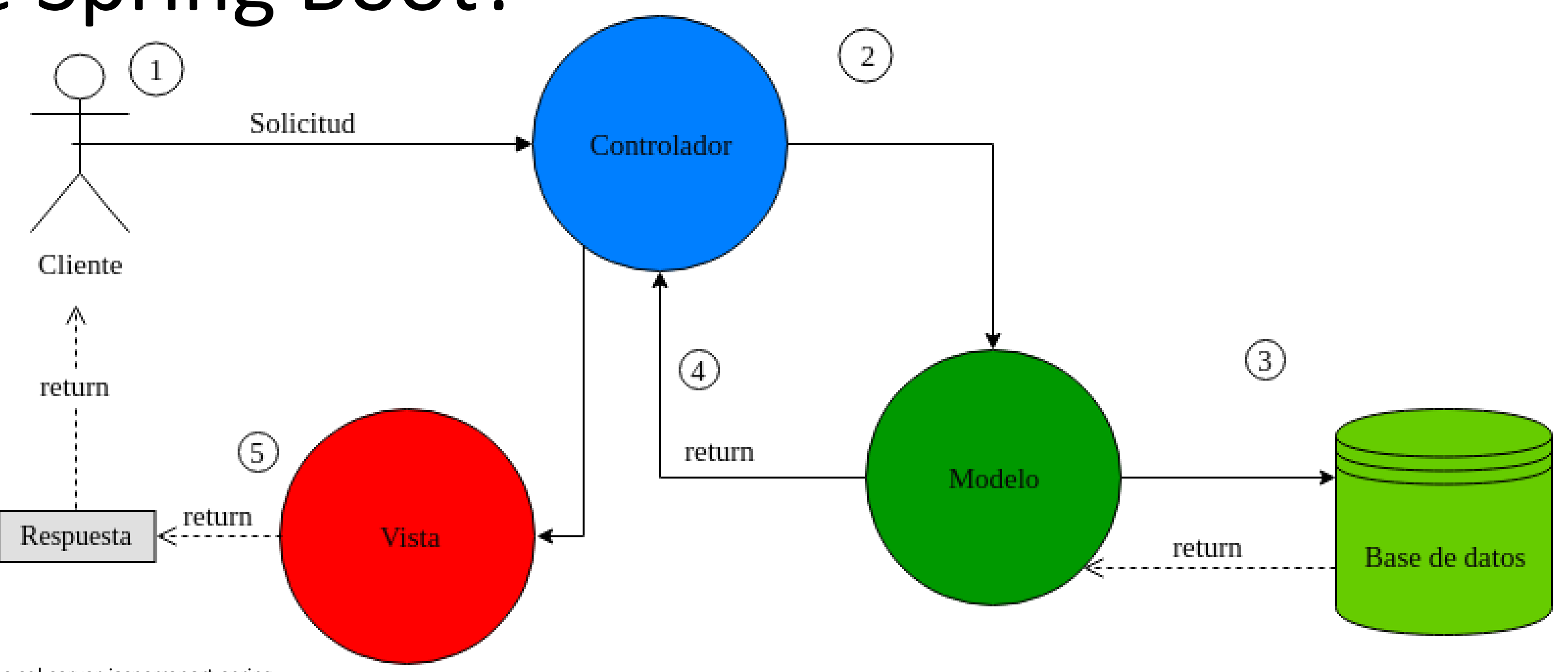
- Spring Boot es un miembro del ecosistema de Spring, que incluye proyectos como Spring Framework, Spring Data, Spring Security, entre otros.
- Se usa Spring MVC para desarrollar aplicaciones web, y Spring Boot simplifica significativamente la configuración requerida.
- Despliegue Independiente y microservicios: facilita la creación de aplicaciones independientes y su relevancia en la construcción de arquitecturas de microservicios.

Índice

- Definición y origen
- Características principales
- Ecosistema
- **MVC**
- Maven
- Configuración de un proyecto básico
- Ejemplo

MVC

- El modelo Modelo-Vista-Controlador (MVC) es un patrón de arquitectura de software ampliamente utilizado en el desarrollo de aplicaciones web.
- ¿Cómo funciona cada componente de este patrón en el contexto de Spring Boot?



Fuente: https://github.com/DavidBrionesFF/planilla-empleados-sql-server-jasperreport-spring

MVC

- El Modelo en una aplicación Spring Boot representa la capa de datos y la lógica de negocio. Consiste en objetos que llevan datos y la lógica relacionada con estos datos. Estos objetos suelen ser POJOs (Plain Old Java Objects) que representan entidades, es decir, reflejan las tablas en una base de datos.
- En Spring Boot, el modelo se gestiona típicamente mediante JPA (Java Persistence API) para mapear estos objetos a registros de base de datos. Spring Data JPA se utiliza a menudo para simplificar las operaciones de base de datos, proporcionando una interfaz de repositorio que maneja tareas comunes (CRUD)

MVC

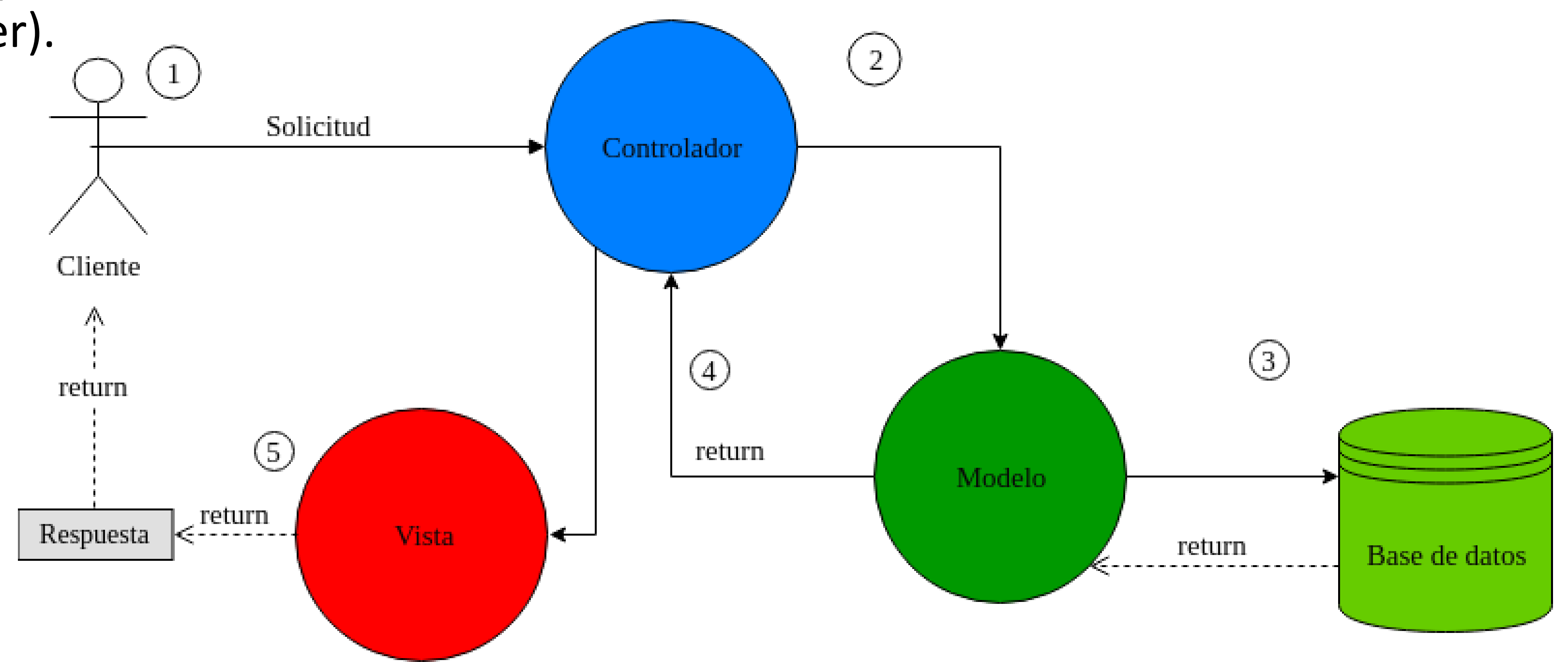
- La Vista es responsable de presentar los datos al usuario. En el contexto de una aplicación web, esto implica generar HTML, CSS y JavaScript para mostrar en el navegador del usuario.
- Spring Boot utiliza plantillas para generar la vista. Estas plantillas pueden ser Thymeleaf, Mustache, JSP, o cualquier otra tecnología de plantillas compatible. Estas plantillas definen cómo se presenta la información del modelo al usuario, y Spring Boot las procesa para generar el HTML final que se envía al navegador.

MVC

- El Controlador actúa como intermediario entre la Vista y el Modelo. Maneja las solicitudes entrantes del usuario (generalmente solicitudes HTTP), interactúa con el modelo para procesar datos o realizar operaciones de negocio, y luego elige una vista para presentar la salida.
- En Spring Boot, los controladores se implementan como clases anotadas con @Controller o @RestController. Estas clases utilizan anotaciones como @RequestMapping o variantes (@GetMapping, @PostMapping, etc.) para mapear diferentes acciones a métodos específicos. En el caso de @RestController, se utiliza principalmente para servicios API donde la respuesta es generalmente JSON o XML, en lugar de una vista HTML.

MVC

- Flujo de trabajo de Solicitud/Respuesta:
  - Cuando un usuario realiza una solicitud (por ejemplo, al cargar una página o enviar un formulario), esta solicitud es manejada por un controlador.
  - El controlador procesa la solicitud, interactúa con el modelo si es necesario (por ejemplo, para recuperar datos o actualizar la base de datos), y luego devuelve una respuesta.
  - Esta respuesta puede ser una vista renderizada (en caso de @Controller) o datos (en caso de @RestController).



Fuente: https://github.com/DavidBrionesFF/planilla-empleados-sql-server-jasperreport-spring

Índice

- Definición y origen
- Características principales
- Ecosistema
- MVC
- **Maven**
- Configuración de un proyecto básico
- Ejemplo

Maven

- Maven es una herramienta de gestión de proyectos y comprensión de software utilizada para manejar dependencias, compilar el proyecto y gestionar el ciclo de vida del software. En proyectos Spring Boot, Maven es comúnmente utilizado para configurar dependencias, plugins y otras configuraciones del proyecto.
- El archivo pom.xml en un proyecto Spring Boot es el corazón de la configuración de Maven. En él se definen las dependencias del proyecto, la versión de Spring Boot utilizada, plugins y cualquier otra configuración específica de Maven necesaria para el proyecto.

Configuración de un proyecto básico

- Spring Initializr (disponible en start.spring.io) es una herramienta en línea que facilita la generación de un proyecto Spring Boot. Se puede seleccionar la versión de Spring Boot, las dependencias iniciales (como Web, JPA, Thymeleaf, etc.), y otras propiedades del proyecto como el nombre, descripción, y tipo de empaquetado (JAR o WAR).
- Una vez configuradas las opciones, Spring Initializr genera un proyecto con la estructura básica y un archivo de configuración y permite descargarlo como un archivo ZIP.

Índice

- Definición y origen
- Características principales
- Ecosistema
- MVC
- Maven
- **Configuración de un proyecto básico**
- Ejemplo

Configuración de un proyecto básico

- Directorio src/main/java: Contiene el código fuente de la aplicación, incluyendo la clase principal que inicia la aplicación Spring Boot.
- Directorio src/main/resources: Alberga recursos como archivos de propiedades (application.properties o application.yml) para la configuración de la aplicación, así como plantillas de vistas y archivos estáticos (CSS, JavaScript, imágenes).
- Directorio src/test/java: Destinado para el código de pruebas, donde se pueden escribir pruebas unitarias y de integración para la aplicación.

Configuración de un proyecto básico

- Archivo application.properties: Describe cómo estos archivos se utilizan para configurar diferentes aspectos de la aplicación, como configuraciones de base de datos, parámetros de servidor, y otras propiedades personalizadas.
- La clase principal de Spring Boot, se marca generalmente con las anotaciones @SpringBootApplication, que actúa como punto de entrada para la ejecución de la aplicación. Esta anotación incluye @Configuration, @EnableAutoConfiguration, y @ComponentScan.

Índice

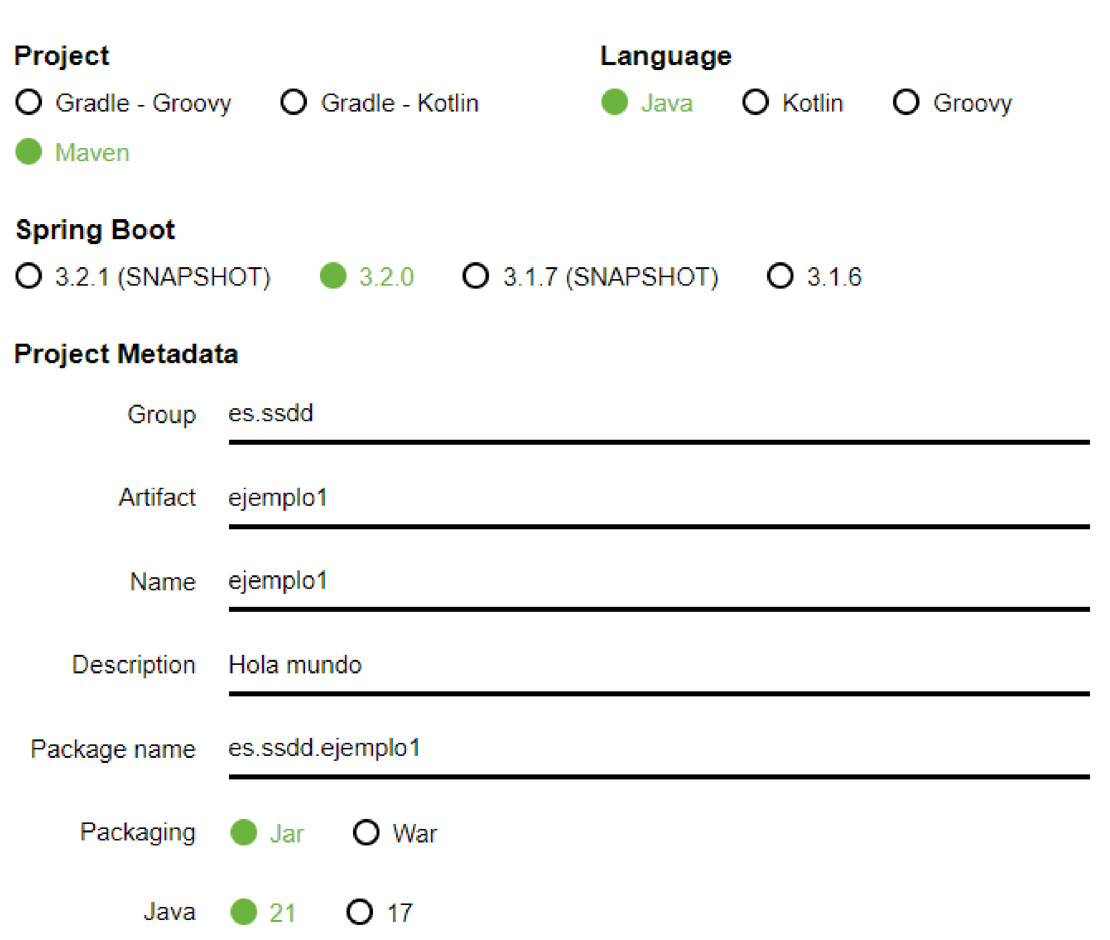
- Definición y origen
- Características principales
- Ecosistema
- MVC
- Maven
- Configuración de un proyecto básico
- **Ejemplo**

Ejemplo

```
mi-aplicacion/
│
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── ejemplo/
│   │   │           └── MiAplicacion.java
│   │   │           └── controlador/
│   │   │               └── SaludoControlador.java
│   │   └── resources/
│   │       ├── templates/
│   │       │   └── saludo.mustache
│   │       └── application.properties
│   └── test/
│       └── java/
└── pom.xml
```

- Descripción de la Estructura:
  - src/main/java/: Este directorio contiene el código fuente de la aplicación.
  - com/ejemplo/MiAplicacion.java: La clase principal que arranca la aplicación Spring Boot.
  - com/ejemplo/controlador/SaludoControlador.java: Un controlador que maneja las solicitudes web y utiliza una vista Mustache para la respuesta.
  - src/main/resources/: Alberga los recursos de la aplicación.
  - templates/: Contiene las plantillas Mustache para las vistas.
  - saludo.mustache: La plantilla Mustache para la página de saludo.
  - application.properties: Archivo de configuración para propiedade de Spring Boot.
  - src/test/java/: Directorio para el código de prueba de la aplicación.
  - pom.xml: Archivo de configuración de Maven que incluye dependencias (como Spring Boot Starter Web y Spring Boot Starter Mustache) y configuración del proyecto.

Ejemplo



Dependencies **ADD DEPENDENCIES...** CTRL + B

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

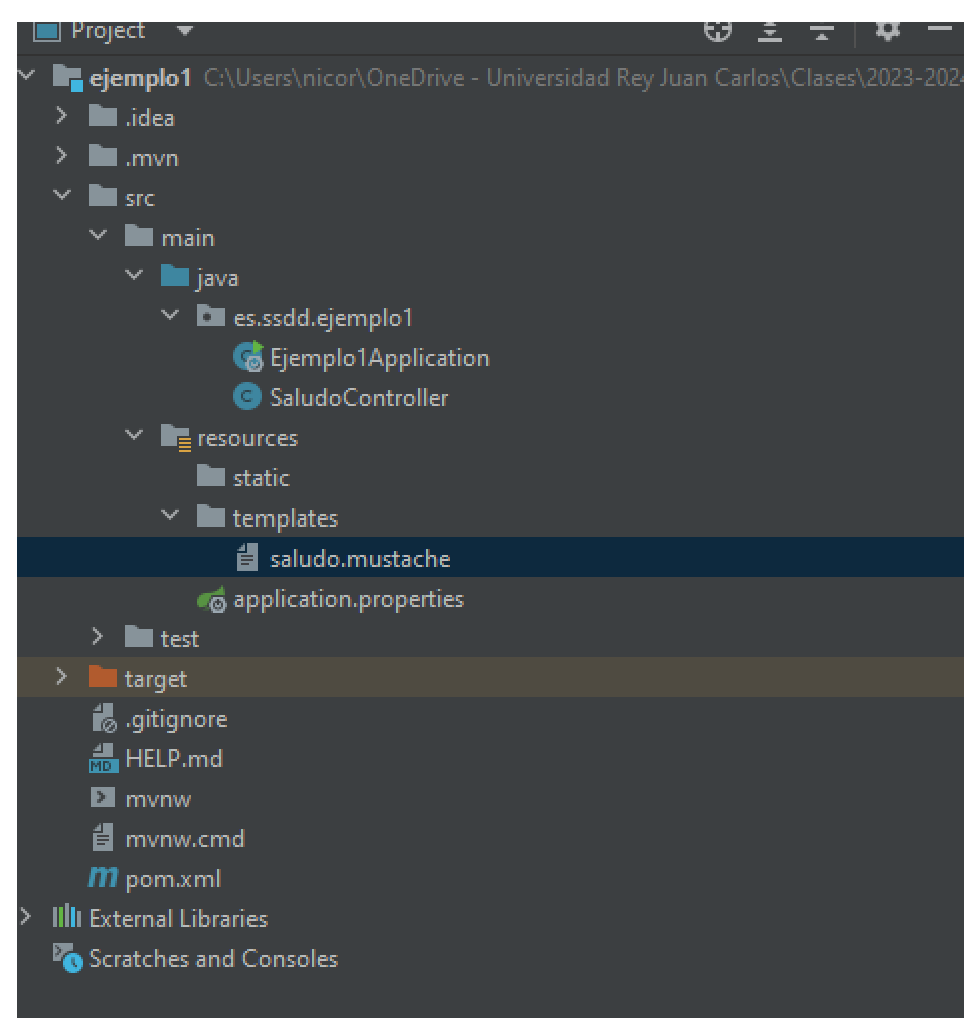
Mustache TEMPLATE ENGINES

Logic-less templates for both web and standalone environments. There are no if statements, else clauses, or for loops. Instead there are only tags.

https://start.spring.io/

Ejemplo

- Descomprimir el zip
- IntelliJ/Open/{ruta}
- Esperar la carga completa del proyecto



Ejemplo

SaludoControlador.java

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
@Controller
public class SaludoController {
   @GetMapping("/saludo")
   public String saludo(Model model) {
      model.addAttribute("mensaje", "Hola, Mundo!");
      return "saludo";
   }
}
```

- “mensaje” es una variable de Mustache
- A la izquierda, nombre de la variable
- A la derecha, contenido de la variable a mostrar por la plantilla
- “saludo” nombre de la vista

- ¡¡Correspondencias!!
  - Nombre de la vista – Nombre fichero
  - Nombre variable - Variable

http://localhost:8080/saludo

saludo.mustache

```
<!DOCTYPE html>
<html>
        <head> <title>Saludo</title>
        </head>
        <body>
                <h1>{{mensaje}}</h1>
        </body>
</html>
```

Sistemas Distribuidos

Bloque II

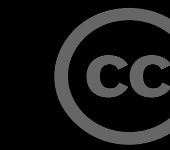
Desarrollo de sistemas distribuidos y aplicaciones web.

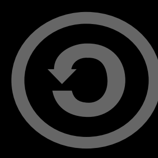
Tema 2.1:

Spring Framework

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

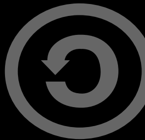
Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.2:

Spring Web y Mustache

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **Formulario web y Mustache**
- Condicionales y bucles
- Formulario con validación
- Operaciones CRUD sobre entidad

Formulario web y Mustache

- Página de perfil de usuario básica
- Crea una plantilla Mustache que muestre información básica de un usuario (como nombre, correo electrónico y una breve biografía).
- Pasa un objeto Usuario desde el controlador de Spring Boot a la vista Mustache y muestra sus propiedades.

Formulario web y Mustache

1. Configuración del proyecto:
   - Crea un nuevo proyecto Spring Boot con las dependencias de Spring Web y Mustache.
2. Creación del Modelo de usuario:
   - Define una clase Usuario con atributos como nombre, correo electrónico y biografía.
3. Controlador para mostrar el perfil:
   - Crea un controlador que pase un objeto Usuario a la vista.
4. Vista Mustache para el perfil de usuario:
   - Diseña una plantilla Mustache para mostrar la información del usuario.

Formulario web y Mustache

Usuario.java

```
public class Usuario {
    private String nombre;
    private String correo;
    private String biografia;
// Constructor, getters y setters
}
```

Los nombres es los atributos en la clase Usuario y los campos a los que accedemos desde Mustache tienen que tener el mismo nombre.

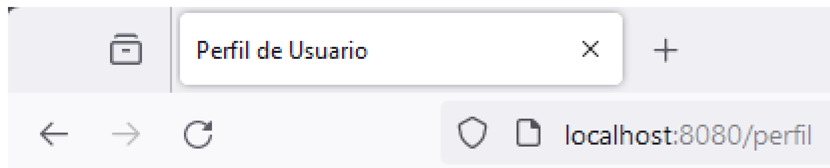
src/main/resources/templates/perfil.mustache

```
<!DOCTYPE html>
<html>
  <head> <title>Perfil de Usuario</title> </head>
  <body>
    <h1>Perfil de {{usuario.nombre}}</h1>
      <p>Correo: {{usuario.correo}}</p>
    <p>Biografía: {{usuario.biografia}}</p>
  </body>
</html>
```

Formulario web y Mustache

UsuarioControlador.java

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class UsuarioControlador {
   @GetMapping("/perfil")
   public String perfil(Model model) {
      Usuario usuario = new Usuario("JD", "juan@example.com", "DevOps.");
      model.addAttribute("usuario", usuario);
      return "perfil";
   }
}
```



Una vez lanzada la aplicación, si accedemos al localhost:8080/perfil, deberíamos ver algo así:

Perfil de JD

Correo: juan@example.com

Biografía: DevOps.

Índice

- Formulario web y Mustache
- **Condicionales y bucles**
- Formulario con validación
- Operaciones CRUD sobre entidad

Condicionales y bucles

- <u>Listado de Productos con Condicionales y Bucles:</u>

- Utiliza Mustache para mostrar una lista de productos, cada uno con nombre, precio y disponibilidad (sí/no).
- Implementa condicionales en Mustache para mostrar un mensaje especial para productos que no estén disponibles.

Condicionales y bucles

1. Configuración del proyecto:
    - Crear un proyecto Spring Boot con las dependencias de Spring Web y Mustache.
2. Creación del Modelo de producto:
    - Definir una clase Producto con atributos como id, nombre, precio y un booleano para la disponibilidad.
3. Controlador para listar productos:
    - Crear un controlador con un método que devuelva una lista de productos, incluyendo algunos productos no disponibles.
4. Vista Mustache para mostrar productos:
    - Diseñar una plantilla Mustache para mostrar la lista de productos y usar condicionales para mostrar mensajes para productos no disponibles.

Condicionales y bucles

Producto.java

```
public class Producto {
    private Long id;
    private String nombre;
    private Double precio;
    private boolean disponible;

    //Getters, setters and constructive
}
```

ProductoControlador.java

```
@Controller
public class ProductoControlador {
    @GetMapping("/productos")
    public String listarProductos(Model model) {
        List<Producto> productos = Arrays.asList(
                new Producto(1L, "Producto 1", 10.0, true),
                new Producto(2L, "Producto 2", 15.0, false));
        model.addAttribute("productos", productos);
        return "productos";
    }
}
```

En este caso, en lugar de pasar un único elemento, se pasa una lista de ellos, en este caso de Producto

Condicionales y bucles

src/main/resources/templates/productos.mustache

```
<!DOCTYPE html>
<html>
<head> <title>Listado de Productos</title>
</head>
<body>
<h1>Productos</h1>
<ul>
  {{#productos}}
    <li>
      {{nombre}} - ${{precio}}
      {{#disponible}}
        (Disponible)
      {{/disponible}}
      {{^disponible}}
        (No disponible)
      {{/disponible}}
    </li>
  {{/productos}}
</ul>
</body>
</html>
```

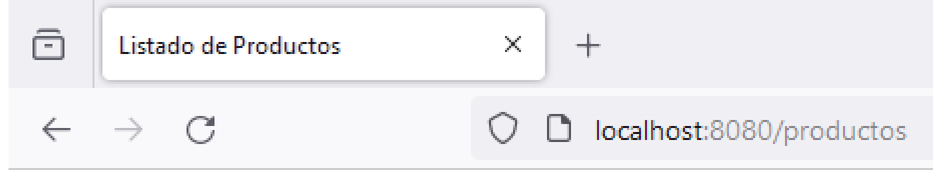
{{#productos}} {{/productos}} es una región de Mustache que funciona como un bucle. En este caso, nos mostrará todos los elementos en productos.

{{#disponible}} {{/disponible}} muestra (Disponible) si el valor es True.

{{^disponible}} es justo al revés

https://mustache.github.io/mustache.5.html

Salida:



Productos

- Producto 1 - $10.0 (Disponible)
- Producto 2 - $15.0 (No disponible)

Índice

- Formulario web y Mustache
- Condicionales y bucles
- **Formulario con validación**
- Operaciones CRUD sobre entidad

Formulario con validación

- <u>Formulario de Contacto con Validación de Lado del Cliente:</u>
- Crea una plantilla Mustache para un formulario de contacto con campos como nombre, correo electrónico y mensaje.
- Utiliza Mustache para generar mensajes de error o confirmación después de enviar el formulario, basándote en la respuesta del servidor.

Formulario con validación

1. Configuración del Proyecto:
   - Crear un proyecto Spring Boot con las dependencias de Spring Web y Mustache.
2. Creación de un Modelo de Mensaje:
   - Definir una clase Mensaje para representar los datos del formulario de contacto.
3. Controlador para el Formulario de Contacto:
   - Crear un controlador para manejar las solicitudes GET y POST del formulario de contacto.
4. Vista Mustache para el Formulario de Contacto:
   - Diseñar una plantilla Mustache para el formulario de contacto y agregar validación de lado del cliente.

Formulario con validación

ContactoControlador.java

```
@Controller
public class ContactoControlador {
   @GetMapping("/contacto")
   public String mostrarFormulario(Model model) {
      model.addAttribute("mensaje", new Mensaje());
      return "contacto";
   }
   @PostMapping("/contacto")
   public String procesarFormulario(Mensaje mensaje, Model model) {
      System.out.println("El mensaje de:" + mensaje.getCorreo() + " se ha mandado");
      model.addAttribute("exito", true);
      return "contacto"; }
}
```

Formulario con validación

src/main/resources/templates/contacto.mustache

```
<!DOCTYPE html>
<html>
<head>
   <title>Formulario de Contacto</title>
   <script>
      function validarFormulario() {
         var nombre = document.forms["formularioContacto"]["nombre"].value;
         var correo = document.forms["formularioContacto"]["correo"].value;
         var contenido = document.forms["formularioContacto"]["contenido"].value;
         if (nombre === "" || correo === "" || contenido === "") {
            alert("Todos los campos son obligatorios.");
            return false;
         }
         return true;
      }
   </script>
</head>
```

Mensaje.java

```
public class Mensaje {
   private String nombre;
   private String correo;
   private String contenido;
// Constructor, getters y setters
}
```

Formulario con validación

src/main/resources/templates/contacto.mustache

```
<body>
<h1>Formulario de Contacto</h1>
{{#exito}}
   <p>Mensaje enviado con éxito.</p>
{{/exito}}
<form name="formularioContacto" action="/contacto" method="post" onsubmit="return validarFormulario()">
   <label for="nombre">Nombre:</label>
   <input type="text" id="nombre" name="nombre"><br><br>
   <label for="correo">Correo electrónico:</label>
   <input type="email" id="correo" name="correo"><br><br>
   <label for="contenido">Mensaje:</label>
   <textarea id="contenido" name="contenido"></textarea><br><br>
   <input type="submit" value="Enviar">
</form>
</body>
</html>
```

Nota

- @PathVariable es una anotación utilizada en Spring.
- Sirve para extraer valores de las variables de ruta (path variables) de las URLs en las solicitudes HTTP
- Cuando defines una URL en tu controlador, puedes especificar partes de esta URL como variables de ruta. Por ejemplo, en una URL como /productos/{id}, {id} es una variable de ruta. Puedes utilizar @PathVariable para capturar el valor de {id} y usarlo en tu método del controlador.

```
@GetMapping("/productos/{id}")
public String getProducto(@PathVariable Long id, Model model) {
// Aquí, el valor de {id} en la URL se pasa al parámetro 'id' del método.
// Puedes usar 'id' para, por ejemplo, buscar un producto en una base de datos. }
```

Índice

- Formulario web y Mustache
- Condicionales y bucles
- Formulario con validación
- **Operaciones CRUD sobre entidad**

Operaciones CRUD sobre entidad

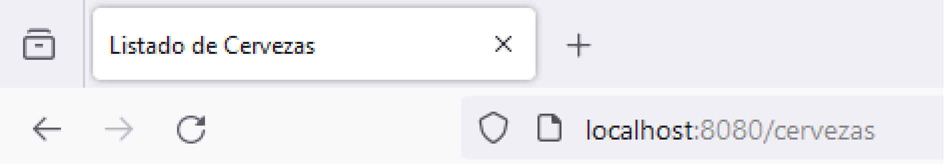
- Almacenamiento en Memoria:
  - Utilizar un mapa para almacenar las Cerveza.
- Agregar Cerveza:
  - Crear un formulario para añadir nuevas Cerveza.
  - Implementar un método en el controlador para procesar este formulario.
- Eliminar Cerveza:
  - Añadir un botón o enlace en la vista para eliminar Cerveza.
  - Implementar un método en el controlador para manejar la eliminación.
- Modificar Cerveza:
  - Crear un formulario para modificar Cerveza existentes.
  - Implementar un método en el controlador para actualizar una Cerveza.
- Mostrar todas las Cerveza
  - Mostrar un listado con todas los Cerveza (solo nombre)
  - Mostrar todas las características del Cerveza al hacer click sobre una

Operaciones CRUD sobre entidad

1. Configuración del Proyecto:
   - Crear un proyecto Spring Boot con las dependencias de Spring Web y Mustache.
2. Creación del Modelo de Cerveza:
   - Definir una clase Cerveza con atributos como id, nombre, tipo y alcohol.
3. Controlador para Operaciones CRUD de Cerveza:
   - Crear un controlador que maneje las operaciones CRUD: listar, agregar, modificar y eliminar Cerveza.
4. Vistas Mustache para Cerveza:
   - Diseñar plantillas Mustache para mostrar la lista de Cerveza y formularios para agregar y modificar

Operaciones CRUD sobre entidad

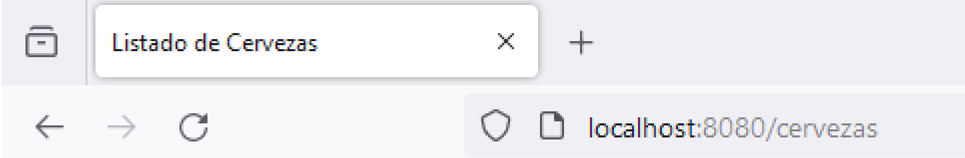
Listado de cervezas (vacío)



Cervezas Disponibles

Agregar Cerveza

Listado de cerveza

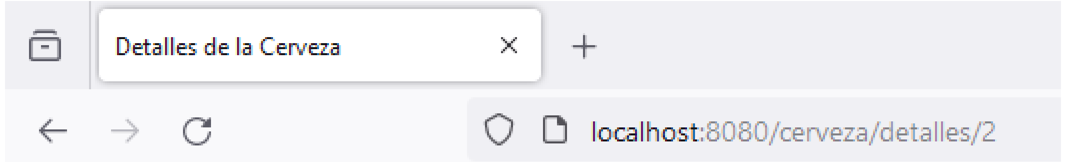


Cervezas Disponibles

- Mahou Session IPA
- Alhambra 1925

Agregar Cerveza

Detalles de la cerveza



Detalles de la Cerveza

Nombre: Alhambra 1925

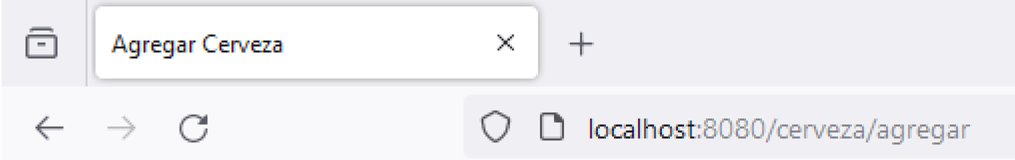
Tipo: Pilsener - Amber Lager

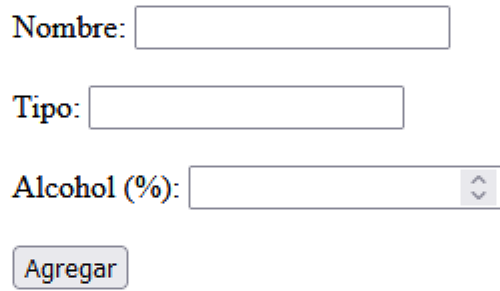
Alcohol: 6.4%

Editar Cerveza Eliminar Cerveza
Volver al listado

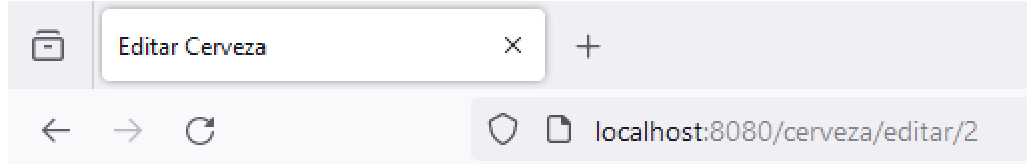
Operaciones CRUD sobre entidad

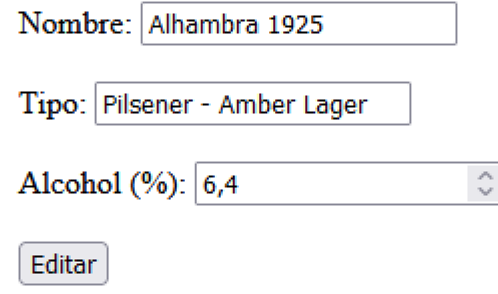
Agregar cerveza



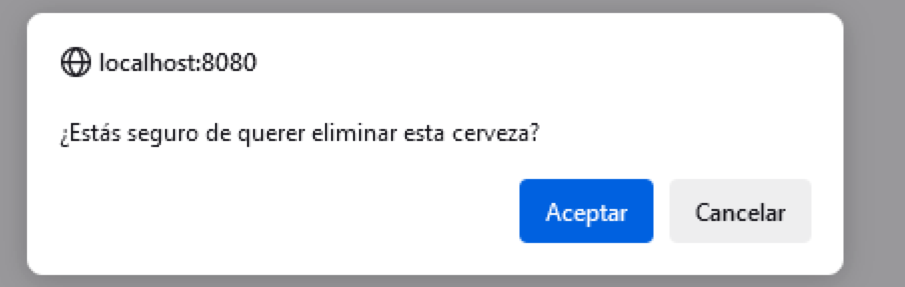


Editar cerveza





Aviso para eliminar cerveza




```
<a href="/cerveza/eliminar/{{cerveza.id}}" onclick="return confirm('¿Estás seguro de querer eliminar esta cerveza?');">Eliminar Cerveza</a>
```

Sistemas Distribuidos

Bloque II

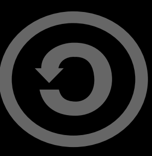
Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.2:

Spring Web y Mustache

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
“Atribución-CompartirIgual 4.0 Internacional” de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Sistemas Distribuidos

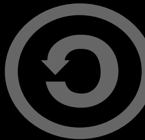
Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.3:

API REST

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **API REST**
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- Componentes

API (Interfaz de Programación de Aplicaciones)

- Una API es un conjunto de reglas y definiciones que permite que diferentes sistemas de software se comuniquen entre sí.
- Funciona como un intermediario que permite a dos aplicaciones interactuar o intercambiar datos.
- Las APIs definen la manera en que los desarrolladores deben realizar solicitudes y estructurar las respuestas para que las aplicaciones puedan comunicarse entre sí. Esto incluye especificaciones de rutas (endpoints), métodos de solicitud (como GET o POST), formatos de datos, etc.

REST (Transferencia de Estado Representacional)

- Estilo arquitectónico:
  - REST es un estilo arquitectónico para el diseño de servicios de red. Fue descrito por Roy Fielding en su tesis doctoral en el año 2000.
- Fundamentos de REST:
  - Se basa en un conjunto de principios y restricciones que, cuando se aplican correctamente, llevan a un sistema distribuido eficiente, escalable y flexible.
- Uso de recursos:
  - En REST, la interacción se realiza a través de recursos, que son cualquier tipo de objeto, dato o servicio que puede ser nombrado. Cada recurso es identificable por una URI (Identificador Uniforme de Recursos).
- Representación del estado:
  - Lo que se transfiere entre el cliente y el servidor no es el recurso en sí, sino una representación del estado del recurso, como HTML, XML, JSON, etc.

API REST

- Combinación de API y REST:
  - Una API REST es una API que sigue los principios del estilo arquitectónico REST. Ofrece una forma estandarizada de integrar sistemas distribuidos, aprovechando los métodos y protocolos de la web, principalmente HTTP.
- Operaciones a través de Métodos HTTP:
  - Utiliza métodos GET, POST, PUT, DELETE, etc., para realizar operaciones en los recursos.
- Independencia del formato de datos:
  - Aunque el formato más común es JSON, una API REST puede usar cualquier formato de representación de datos que sea entendible tanto por el cliente como por el servidor.
- Sin Estado y Sin sesión:
  - REST es sin estado, lo que significa que cada solicitud de un cliente debe contener toda la información necesaria para procesarla, sin depender de ninguna información almacenada en el servidor respecto a sesiones anteriores.

Índice

- API REST
- **Recursos**
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- Componentes

Recursos

- Un recurso en REST es cualquier elemento o entidad que puede ser nombrado, identificado, y sobre el cual se pueden realizar operaciones. Puede ser un objeto físico, un documento, una imagen, un servicio, etc.
- Características:
  - Representación: Un recurso puede tener varias representaciones, como JSON, XML, HTML, etc.
  - Estado: Los recursos tienen un estado que puede cambiar con el tiempo.
  - Identificación Única: Cada recurso es único y accesible a través de su URI.
- Ejemplos:
  - Un artículo en un blog, un usuario en un sistema, un producto en un catálogo.

Índice

- API REST
- Recursos
- **URIs**
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- Componentes

URIs

- Las URIs son la forma en que se identifican y localizan los recursos en una API REST. Son análogas a las direcciones web que utilizamos para acceder a páginas específicas en Internet.
- Características:
  - Uniformidad: Siguen un formato estándar que permite localizar recursos en la red.
  - Inmutabilidad: Idealmente, una URI no debe cambiar con el tiempo. Si un recurso cambia, su URI debe permanecer constante.
  - Legibilidad: Aunque no es un requisito, es una buena práctica diseñar URIs que sean fáciles de entender y predecir por humanos.

URIs

- Claridad y Simplicidad:
  - Las URIs deben ser intuitivas y fáciles de comprender. Por ejemplo, usar /usuarios para acceder a recursos de usuarios.
- Uso de nombres en plural:
  - Es común usar nombres en plural para colecciones de recursos (p. ej., /usuarios en lugar de /usuario).
- Estructura Jerárquica:
  - Las URIs pueden representar relaciones jerárquicas. Por ejemplo, /usuarios/123/posts podría representar todos los posts del usuario con ID 123.
- Evitar uso de verbos:
  - Las URIs deben centrarse en identificar recursos, no acciones. Las acciones se definen mediante los métodos HTTP (GET, POST, PUT, DELETE).

URIs

- Una URI típica para un recurso REST podría ser algo así como https://www.ejemplo.com/recursos/123:
  - https://www.ejemplo.com/ es el dominio base,
  - /recursos/ es el path que indica el tipo de recurso,
  - y 123 es un identificador único para un recurso específico.
- Los recursos representan entidades sobre las que se pueden realizar operaciones, y las URIs proporcionan una forma estandarizada y coherente de identificar y acceder a estos recursos.

Índice

- API REST
- Recursos
- URIs
- **Métodos HTTP**
- Códigos de estado
- JSON
- Postman
- Componentes

Métodos HTTP

1. GET

- Uso: Solicitar datos de un recurso específico o un conjunto de recursos.
- Características:
  - Es un método seguro, lo que significa que no modifica el estado del recurso.
  - Es idempotente, lo que implica que realizar la misma solicitud GET varias veces produce el mismo resultado.
- Ejemplo de Uso:
  - Obtener la información de un usuario (GET /usuarios/123) o listar todos los usuarios (GET /usuarios).

Métodos HTTP

2. POST

- Uso: Enviar datos para crear un nuevo recurso.
- Características:
  - No es ni seguro ni idempotente. Cada solicitud puede resultar en la creación de un nuevo recurso o en un cambio en el servidor.
- Ejemplo de Uso:
  - Crear un nuevo usuario con datos en el cuerpo de la solicitud (POST /usuarios).

Métodos HTTP

3. PUT

- Uso: Actualizar un recurso existente o crear un recurso en una URI específica si no existe.
- Características:
  - Es idempotente. Realizar la misma solicitud PUT varias veces tiene el mismo efecto que hacerlo una sola vez.
- Ejemplo de Uso:
  - Actualizar la información de un usuario (PUT /usuarios/123).

Métodos HTTP

4. DELETE

- Uso: Eliminar un recurso específico.
- Características:
  - Es idempotente. Una vez que el recurso se elimina, realizar la misma solicitud no tiene ningún efecto adicional.
- Ejemplo de Uso:
  - Eliminar un usuario (DELETE /usuarios/123).

Métodos HTTP

5. PATCH

- Uso: Aplicar actualizaciones parciales a un recurso.
- Características:
  - Puede ser idempotente o no, dependiendo de cómo se implemente.
- Ejemplo de Uso:
  - Actualizar ciertos campos de un usuario sin modificar todo el recurso (PATCH /usuarios/123).

Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- **Códigos de estado**
- JSON
- Postman
- Componentes

Códigos de estado

- En las APIs REST, las respuestas y los códigos de estado HTTP son esenciales para comunicar el resultado de las solicitudes de los clientes.
- Estos códigos informan al cliente sobre el éxito, el fallo o la necesidad de tomar una acción adicional.
- 1xx - Respuestas Informativas:
  - Indican que la solicitud fue recibida y está siendo procesada.
  - Ejemplo: 100 Continue

Códigos de estado

- 2xx - Éxito:
  - Significan que la solicitud fue recibida, entendida y aceptada correctamente.
  - 200 OK: Respuesta estándar para solicitudes exitosas.
  - 201 Created: Indica que un recurso fue creado exitosamente (generalmente en respuesta a solicitudes POST o PUT).
  - 204 No Content: La solicitud fue procesada con éxito, pero no hay contenido para devolver.
- 3xx - Redirecciones:
  - Indican que se deben tomar acciones adicionales para completar la solicitud.
  - Ejemplo: 301 Moved Permanently

Códigos de estado

- 4xx - Errores del cliente:
  - Se usan cuando la solicitud no pudo ser procesada debido a errores del lado del cliente.
  - 400 Bad Request: La solicitud no se pudo entender o procesar.
  - 401 Unauthorized: Autenticación requerida o fallida.
  - 403 Forbidden: El servidor entiende la solicitud, pero se niega a autorizarla.
  - 404 Not Found: El recurso solicitado no fue encontrado.
  - 405 Method Not Allowed: El método HTTP utilizado no está permitido para el recurso.
- 5xx - Errores del servidor:
  - Indican fallos del lado del servidor.
  - 500 Internal Server Error: Error genérico cuando el servidor falla al procesar la solicitud.
  - 503 Service Unavailable: El servidor no está disponible, generalmente por mantenimiento o sobrecarga.

Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- **JSON**
- Postman
- Componentes

JSON – Características principales

- JSON (JavaScript Object Notation) es un formato de representación de datos ampliamente utilizado en el desarrollo de APIs, especialmente en APIs REST.
- Su popularidad se debe a su simplicidad, eficiencia y facilidad de uso en aplicaciones web y móviles
- Basado en Texto:
  - JSON es un formato de datos basado en texto, lo que lo hace fácilmente legible por humanos y máquinas.

JSON – Características principales

- Estructura de Datos Ligera:
  - Ofrece una forma concisa y fácil de representar estructuras de datos como objetos y arrays.
  - Ejemplo de un objeto JSON: { "nombre": "Ana", "edad": 25 }
- Independiente del Lenguaje:
  - A pesar de estar basado en la sintaxis de JavaScript, es independiente de cualquier lenguaje de programación.
  - La mayoría de los lenguajes modernos soportan JSON a través de bibliotecas o funciones nativas.
- Fácil de Analizar:
  - La simplicidad de su estructura lo hace fácil de analizar y generar, lo que reduce la complejidad del código en el cliente y el servidor.

JSON – Uso

- Intercambio de Datos:
  - JSON es el formato más común para el envío y recepción de datos en APIs REST.
  - Ejemplo: Un cliente envía una solicitud POST con un cuerpo JSON para crear un nuevo recurso.
- Representación de Recursos:
  - Permite representar los recursos de una API de manera estructurada y clara.
  - Ejemplo: Un recurso de usuario se puede representar como un objeto JSON con propiedades como nombre, correo electrónico, etc.
- Negociación de Contenido:
  - Las APIs REST pueden usar JSON como uno de los formatos en la negociación de contenido, respondiendo con datos en formato JSON si el cliente lo solicita.

JSON – Estructura y sintaxis

- Objetos:
  - Representados por llaves { }, contienen un conjunto de pares de clave-valor.
  - Ejemplo: { "nombre": "Carlos", "edad": 40 }
- Arrays:
  - Representados por corchetes [ ], son colecciones ordenadas de valores.
  - Ejemplo: ["manzanas", "naranjas", "peras"]
- Valores:
  - Pueden ser cadenas de texto (strings), números, booleanos (true/false), arrays o incluso otros objetos.
  - Ejemplo: { "nombres": ["Ana", "Luis"], "activo": true }

JSON – Estructura y sintaxis

- Cadenas de Texto (Strings):
  - Deben estar entre comillas dobles.
  - Ejemplo: "nombre": "Juan"
- Números:
  - Pueden ser enteros o flotantes.
  - Ejemplo: "edad": 30
- Limitaciones:
  - No tiene soporte nativo para comentarios, lo que puede ser un inconveniente para la documentación en el código.
  - Limitado en términos de tipos de datos (por ejemplo, no hay distinción entre tipos de números).

JSON – Ejemplo

```
{
  "id": "12345",
  "nombre": "Nicolás Rodríguez",
  "edad": 33,
  "email": "nicolas.rodriguez@urjc.es",
  “enActivo": true,
  "roles": ["usuario", "profesor"],
  "direccion": {
    "calle": "Calle Falsa 123",
    "ciudad": "Alcorcón",
    "codigoPostal": "28922"
  }
}
```

- Strings: id, nombre, email
- Número: edad
- Boolean : enActivo
- Lista: roles
- Objeto: dirección

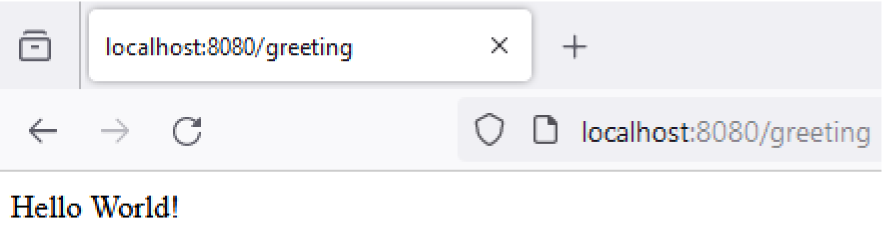
Ejemplo

- Aplicación en SpringBoot con la dependencia de SpringWeb para saludar desde una API REST

GreetingRESTController

```
@RestController
public class GreetingRESTController {
    @GetMapping("/greeting")
    public String greeting() {
        return "Hello World!";
    }
}
```

Salida:



Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- **Postman**
- Componentes

Postman

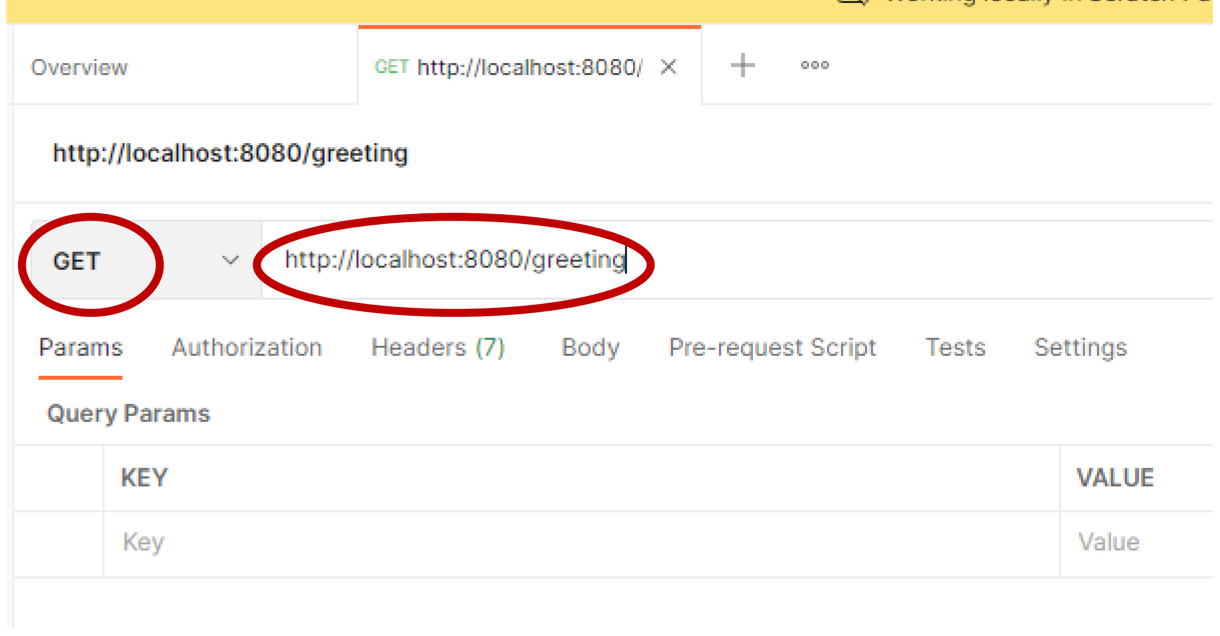
- Postman es una plataforma popular para el desarrollo y pruebas de APIs.
- Ofrece un conjunto de herramientas para construir, probar y documentar APIs, facilitando el trabajo de desarrolladores en el diseño de APIs, la verificación de respuestas y la colaboración en equipo.
- Construcción y Prueba de Solicitudes:
  - Permite a los usuarios crear solicitudes HTTP a APIs, incluyendo todos los métodos (GET, POST, PUT, DELETE, etc.).
  - Soporta la personalización de cabeceras, parámetros, cuerpos de solicitud y autenticación.

Postman

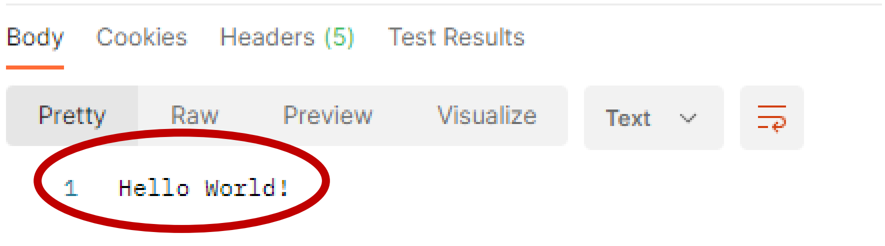
- Colecciones y Ambientes:
  - Las colecciones permiten organizar y guardar solicitudes, lo que es útil para agrupar y compartir sets de llamadas a APIs.
  - Los ambientes son conjuntos de variables que permiten cambiar fácilmente entre diferentes configuraciones, como desarrollo, pruebas y producción.
- Documentación:
  - Ofrece la capacidad de generar y publicar documentación de APIs a partir de las colecciones, facilitando la comunicación y colaboración entre equipos.
- https://www.postman.com/

Postman

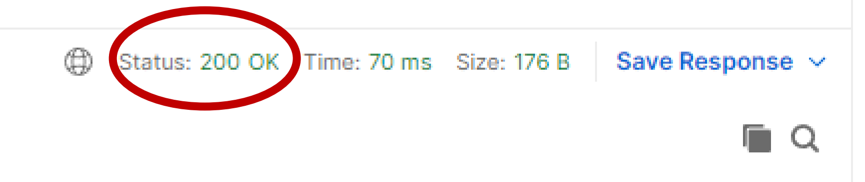
Petición vía Postman



Respuesta vía Postman



Respuesta HTTP Postman



Postman

Petición vía Postman

https://swapi.dev/api/people/1

- A la izquierda se puede observar la respuesta obtenida de realizar una petición GET a la URL arriba indicada.
- A la hora de obtener una respuesta, se puede elegir el formato de respuesta. En este caso, hemos elegido JSON.

Ejemplo

- Aplicación en SpringBoot con la dependencia de SpringWeb para añadir un Usuario a través de la API REST
- Se utiliza un mapa para almenar un usuario y su id.
- Este id es autoincremental.

```
public class Usuario {
    private String nombre;
    private int edad;

    // Constructor, getters y setters para 'nombre' y 'edad'
}
```

Ejemplo

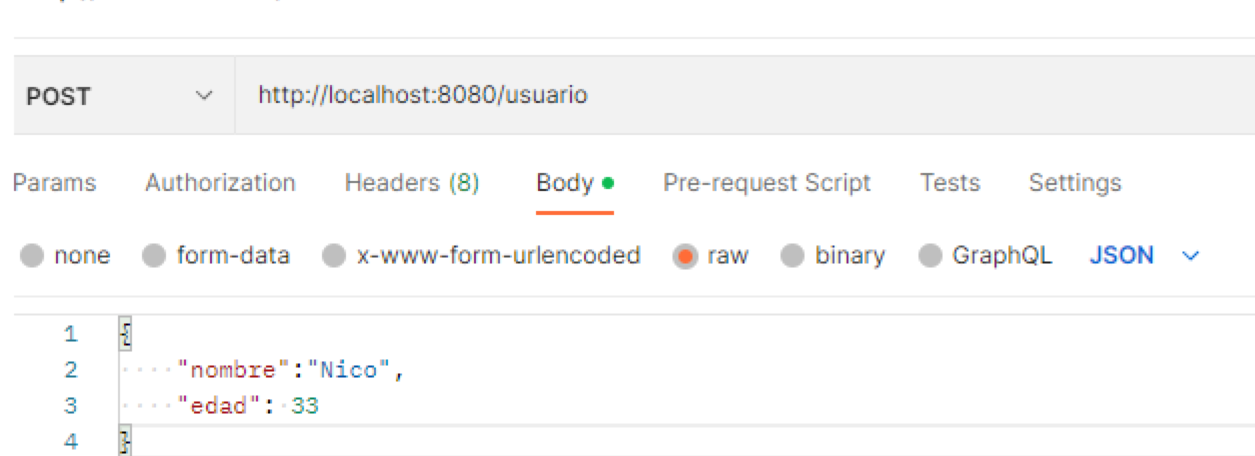
UsuarioRESTController.java

```
@PostMapping("/usuario")
public ResponseEntity<String> agregarUsuario(@RequestBody Usuario usuario) {
    int userId = idCounter.incrementAndGet();
    usuarios.put(userId, usuario);
    return ResponseEntity.status(201).body("Usuario agregado con ID: " + userId);
}
```

- Se utiliza el PostMapping para crear un Usuario. Por tanto, la petición es de tipo POST
- @RequestBody recibe una instancia de la clase Usuario, y la genera automáticamente.
- Se devuelve un ResponseEntity, que permite añadir código de respuesta y cuerpo de respuesta.

Ejemplo

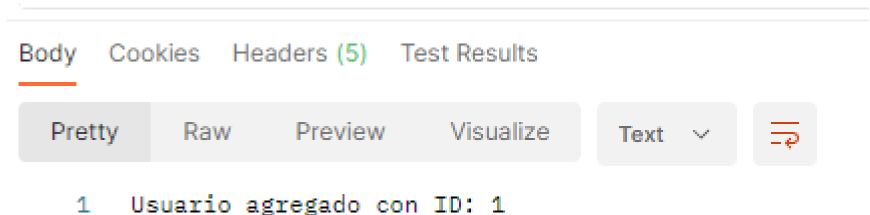
Petición vía Postman



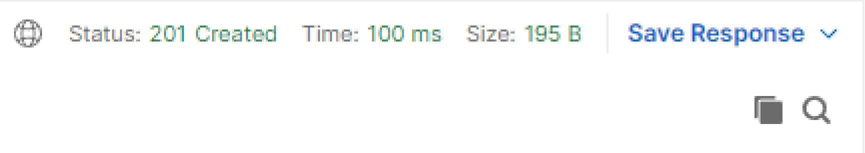
- La petición ahora es de tipo POST.
- Añadimos la URL del endpoint
- Tenemos que enviar la "instancia" de la clase Usuario
- Body/raw/JSON y escribimos la instancia

Ejemplo

Respuesta vía Postman



Respuesta vía Postman



Ejercicio

- Implementar todas las operaciones CRUD + PATCH sobre la entidad Cerveza con una API REST
- Generar una colección de Postman para todas los endpoint

Ejercicio

- Crear una nueva cerveza
  - Endpoint: /cervezas
  - Método HTTP: POST
  - Descripción: Este endpoint acepta un objeto Cerveza en el cuerpo de la solicitud y lo agrega al mapa de cervezas. Genera un ID único para cada nueva cerveza.
- Obtener una cerveza por ID
  - Endpoint: /cervezas/{id}
  - Método HTTP: GET
  - Descripción: Retorna los detalles de la cerveza con el ID especificado. Si no existe una cerveza con ese ID, devuelve un estado 404 (Not Found).

Ejercicio

- Actualizar una cerveza existente
  - Endpoint: /cervezas/{id}
  - Método HTTP: PUT
  - Descripción: Reemplaza completamente la cerveza con el ID especificado con la nueva información proporcionada en el cuerpo de la solicitud. Si la cerveza no existe, devuelve un estado 404 (Not Found).
- Eliminar una cerveza
  - Endpoint: /cervezas/{id}
  - Método HTTP: DELETE
  - Descripción: Elimina la cerveza con el ID especificado. Si no existe una cerveza con ese ID, devuelve un estado 404 (Not Found).

Ejercicio

- Actualizar parcialmente una cerveza
  - Endpoint: /cervezas/{id}
  - Método HTTP: PATCH
  - Descripción: Permite actualizar parcialmente los atributos de una cerveza existente (como el nombre, tipo o porcentaje de alcohol). Solo los campos proporcionados en el cuerpo de la solicitud serán actualizados.
- Obtener todas las cervezas:
  - URL: /cervezas
  - Método HTTP: GET
  - Descripción: Este endpoint maneja solicitudes GET para listar todas las cervezas almacenadas en la API. Devuelve una lista de objetos Cerveza con sus detalles.

Ejercicio

Elementos a insertar

```
[
    {
      "nombre": "Alhambra 1925",
      "tipo": "Lager",
      "alcohol": 6.4
    },
    {
      "nombre": "Mahou",
      "tipo": "Lager",
      "alcohol": 5.5
    },
    {
      "nombre": "Estrella Galicia",
      "tipo": "Lager",
      "alcohol": 5.5
    }
]
```

Listado de elementos

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1  [
2      {
3          "id": 1,
4          "nombre": "Alhambra 1925",
5          "tipo": "Lager",
6          "alcohol": 6.4
7      },
8      {
9          "id": 2,
10         "nombre": "Mahou",
11         "tipo": "Lager",
12         "alcohol": 5.5
13     },
14     {
15         "id": 3,
16         "nombre": "Estrella Galicia",
17         "tipo": "Lager",
18         "alcohol": 5.5
19     }
20 ]
```

Índice

- API REST
- Recursos
- URIs
- Métodos HTTP
- Códigos de estado
- JSON
- Postman
- **Componentes**

Componentes

- Las anotaciones @Service y @Autowired son componentes clave en Spring Framework, y se utilizan para implementar la inyección de dependencias y la separación de la lógica de negocio, lo que facilita la creación de aplicaciones escalables y mantenibles.
- @Service es ideal para clases que implementan la lógica de negocios.
- @Autowired es esencial para inyectar dependencias de manera limpia y desacoplada.

Componentes

- @Service es una anotación en Spring que se utiliza para marcar una clase en Java como un servicio. Es una especialización de la anotación @Component, que indica que una clase es un "bean" de Spring y que debe ser gestionada por el contenedor de Spring.
- Se coloca sobre las clases que implementan la lógica de negocios. En la arquitectura MVC (Modelo-Vista-Controlador), @Service se utiliza en la capa de servicio, que se sitúa entre los controladores y las capas de acceso a datos (repositorios).

Componentes

- @Autowired es una anotación que permite a Spring resolver e inyectar beans colaboradores en tu clase automáticamente. Es decir, Spring buscará y proporcionará las dependencias requeridas para un objeto sin necesidad de configuración manual.
- Se puede usar en constructores, métodos setter y campos para inyectar dependencias. Spring se encargará de buscar en su contenedor el bean adecuado que coincida con la propiedad o parámetro que necesita ser inyectado.

Ejercicio

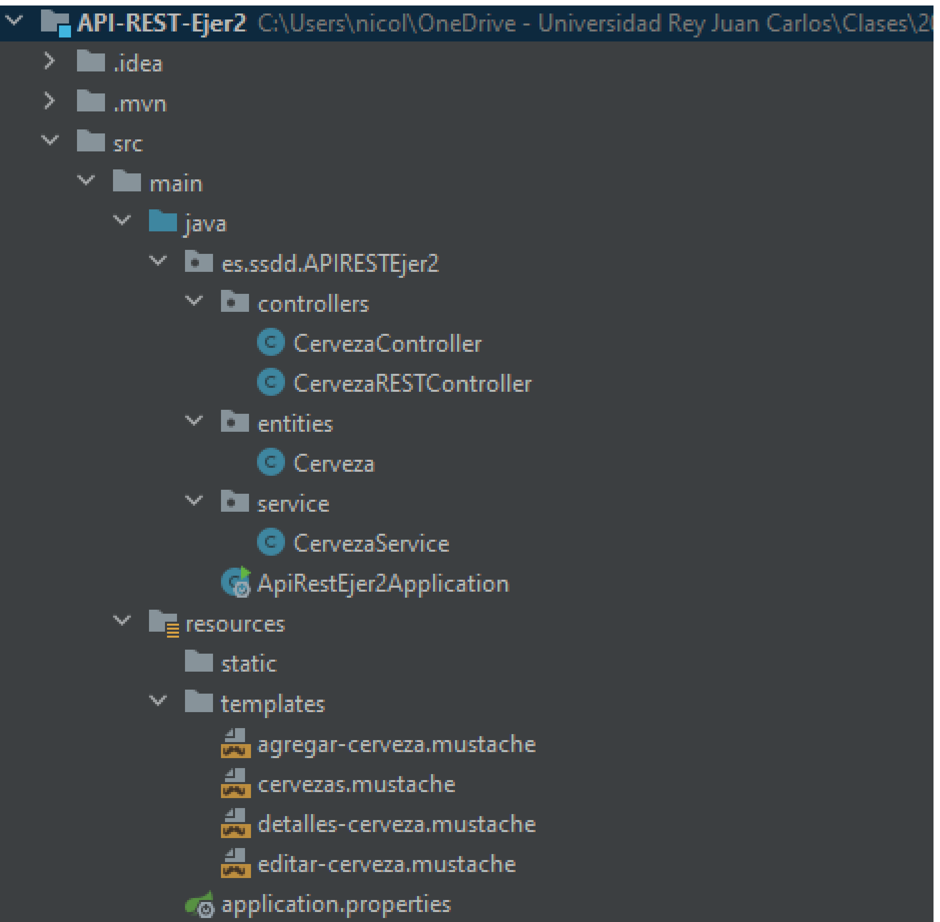
- Unir el controlador web y el controlador de la API REST de Cerveza, de manera en que si añado una cerveza desde la API REST sea capaz de verla en la web y viceversa.
- Para ello es necesario:
  - Implementar un @Servicio con el mapa.
  - Añadir el @Autowired a ambos controladores.
  - Modificar los métodos dentro de los controladores para que usen el @Autowired.
  - Evitar conflictos de endpoints.
  - Separar las clases en:
    - Entidades.
    - Controladores.
    - Servicios.

Ejercicio

ServiceCerveza.java

```
@Service
  public class CervezaService {
  private final Map<Long, Cerveza> cervezas = new HashMap<>();
  private final AtomicLong nextId = new AtomicLong();

  public Cerveza crearCerveza(Cerveza cerveza) {
    long id = nextId.incrementAndGet();
    cerveza.setId(id);
    cervezas.put(id, cerveza);
    return cerveza;
  }
```

Estructura



Ejercicio

CervezaRESTController.java

```
@RestController
@RequestMapping("/api/cervezas")
public class CervezaRESTController {

    @Autowired
    private CervezaService cervezaService;

    @PostMapping
    public ResponseEntity<Cerveza> crearCerveza(@RequestBody Cerveza cerveza) {
        return ResponseEntity.status(201).body(cervezaService.crearCerveza(cerveza));
    }
}
```

- @RequestMapping(“/api/cervezas”) factoriza las URL de la API REST. En este caso, se produce un cambio, y es que, a partir de ese punto, todas las URL de la API REST tienen ese inicio de URL.

Sistemas Distribuidos

Bloque II

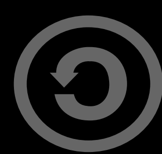
Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.3:

API REST

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
"Atribución-CompartirIgual 4.0 Internacional" de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Sistemas Distribuidos

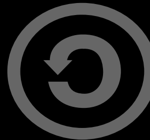
Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 2.4:

Spring Data

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **Introducción**
- JPA
- Repositorios
- H2
- Tipos de correspondencia

Introducción

- Dependencia diseñada para simplificar la implementación de capas de acceso a datos en aplicaciones Java.
- Reduce la cantidad de código boilerplate necesario para la implementación de capas de datos.
- Proporcionar una manera consistente y fácil de acceder a datos almacenados en diferentes tipos de bases de datos, tanto SQL como NoSQL.
- Fácil y rápida integración con Spring.

Introducción

- Esfuerzo reducido en el mantenimiento y actualización de las capas de acceso a datos.
- Facilidad de escalar y adaptarse a diferentes tecnologías de bases de datos.
- Permite a los desarrolladores enfocarse más en la lógica de negocio en lugar de las complejidades de la conectividad y operaciones de base de datos.

Introducción

- Spring Data JPA (Java Persistence API):
  - Proporciona integración con bases de datos relacionales mediante JPA.
  - Facilita la implementación de repositorios de datos para entidades JPA.
- Spring Data MongoDB:
  - Soporte específico para MongoDB, una base de datos NoSQL orientada a documentos.
  - Permite realizar operaciones de datos en documentos MongoDB con facilidad.
- Spring Data Redis:
  - Proporciona acceso y manejo de datos en Redis, una base de datos en memoria.
  - Ideal para casos de uso que requieren alta velocidad y escalabilidad.
- Spring Data Cassandra:
  - Soporte para Apache Cassandra, una base de datos NoSQL distribuida.
  - Permite manejar grandes cantidades de datos con alta disponibilidad.

Índice

- Introducción
- **JPA**
- Repositorios
- H2
- Tipos de correspondencia

JPA

- Spring Data JPA es un subproyecto de Spring Data que se centra en la integración con bases de datos relacionales usando la Java Persistence API (JPA).
- Facilita la implementación de capas de acceso a datos, proporcionando una manera más simple y concisa de realizar operaciones de base de datos.
- Proporciona interfaces de repositorio predefinidas, como CrudRepository y PagingAndSortingRepository.
- Simplifica las operaciones CRUD y de paginación/sorteo.

JPA

- Spring Data JPA es un subproyecto de Spring Data que se centra en la integración con bases de datos relacionales usando la Java Persistence API (JPA).
- Facilita la implementación de capas de acceso a datos, proporcionando una manera más simple y concisa de realizar operaciones de base de datos.
- Consta de varias características clave:

JPA

- Repositorio abstracción:
  - Proporciona interfaces de repositorio predefinidas, como CrudRepository y PagingAndSortingRepository.
  - Simplifica las operaciones CRUD y de paginación/sorteo.
- Mapeo Objeto-Relacional (ORM):
  - Integra JPA para mapear objetos en Java a tablas en bases de datos relacionales.
  - Automatiza la conversión entre registros de base de datos y objetos Java.
- Consultas derivadas:
  - Permite la creación de consultas a partir de los nombres de los métodos en las interfaces de repositorio.
  - Ejemplo: findByUsername(String username) automáticamente genera una consulta para buscar por el campo username.

JPA

- Annotaciones de consulta:
  - Utiliza la anotación @Query para definir consultas personalizadas utilizando JPQL (Java Persistence Query Language).
  - Permite escribir consultas complejas que no se pueden derivar directamente del nombre del método.
- Integración con Spring Framework:
  - Se integra sin problemas con otros componentes, como Spring Transaction Management.
  - Proporciona soporte para la gestión de transacciones y caché.
- Gestión de entidades:
  - Facilita la gestión del ciclo de vida de las entidades JPA, como persistencia, actualización y eliminación.
  - Soporta asociaciones entre entidades, como relaciones Uno-a-Uno, Uno-a-Muchos, y Muchos-a-Muchos.

Índice

- Introducción
- JPA
- **Repositorios**
- H2
- Tipos de correspondencia

Repositorios

- Un repositorio es un mediador entre el dominio de la aplicación y las operaciones de acceso a datos.
- Reducen la necesidad de código boilerplate y centralizan la lógica de acceso a datos.
- Tipos de Repositorios en Spring Data:
  - Diferentes tipos de interfaces de repositorio proporcionados por Spring Data, como CrudRepository, PagingAndSortingRepository, y JpaRepository.
- Operaciones CRUD:
  - Los repositorios simplifican las operaciones CRUD (Crear, Leer, Actualizar, Borrar).
  - Ejemplos de métodos comunes: save(), findAll(), findById(), y delete().

Repositorios

- CrudRepository es una de las interfaces de repositorio proporcionadas por Spring Data.
- Está diseñada para proporcionar operaciones CRUD (Crear, Leer, Actualizar, Borrar) para un tipo de entidad específico.
- Es una interfaz genérica que se parametriza con el tipo de la entidad y el tipo de su clave primaria.

UserRepository.java

```java
import org.springframework.data.repository.CrudRepository;
public interface UserRepository extends CrudRepository<User, Long> {
}
```

Repositorios

- Crear y Actualizar:
  - save(S entity): Guarda una entidad dada. Si la entidad ya existe, la actualiza; de lo contrario, la crea.
  - saveAll(Iterable<S> entities): Guarda todas las entidades dadas.
- Leer:
  - findById(ID id): Recupera una entidad por su clave primaria.
  - findAll(): Devuelve todas las entidades.
  - findAllById(Iterable<ID> ids): Recupera todas las entidades cuyas claves primarias están en el iterable proporcionado.
- Borrar:
  - deleteById(ID id): Elimina la entidad con la clave primaria dada.
  - delete(S entity): Elimina la entidad dada.
  - deleteAll(): Elimina todas las entidades.

Índice

- Introducción
- JPA
- Repositorios
- **H2**
- Tipos de correspondencia

H2

- La base de datos H2 es un sistema de gestión de bases de datos relacionales escrito en Java. Es conocida por su pequeño tamaño, alta velocidad y por ser de código abierto.
- Base de Datos en Memoria y Basada en Disco:
  - H2 puede ser configurada tanto para almacenar sus datos en memoria como en disco.
  - Se usa a menudo para pruebas unitarias y de integración.
  - El modo basado en disco permite la persistencia de datos entre reinicios de la aplicación.
- Soporte JDBC y Compatibilidad SQL:
  - H2 proporciona un driver JDBC para conectar con la base de datos, permitiendo su uso con una gran variedad de herramientas y frameworks de Java.
  - Es compatible con la mayoría de las sintaxis y características de SQL, facilitando la migración desde otros sistemas de gestión de bases de datos SQL.

H2

- Fácil de Usar y Configurar:
  - H2 es muy simple de configurar y no requiere de una instalación compleja.
  - Puede ejecutarse como un proceso independiente o integrarse dentro de aplicaciones Java.
- Consola Web para la Gestión de la Base de Datos:
  - Incluye una consola web para facilitar la administración de la base de datos, permitiendo ejecutar consultas SQL, administrar tablas, entre otras tareas.
- Alto Rendimiento:
  - H2 está optimizada para un alto rendimiento, con velocidades de lectura y escritura muy rápidas, especialmente en el modo en memoria.

H2

- Soporte para Funcionalidades Avanzadas:
  - A pesar de su tamaño reducido, H2 soporta características avanzadas como transacciones multiversión (MVCC), cifrado de base de datos, y compresión de datos.
- Uso en Desarrollo y Pruebas:
  - Es comúnmente usada en entornos de desarrollo y pruebas debido a su facilidad de configuración y rápido despliegue.
  - Ideal para casos en los que se requiere una base de datos ligera y no se necesitan las capacidades completas de un sistema de gestión de bases de datos más grande.
- Soporte para Clientes de Múltiples Lenguajes:
  - Aunque H2 está escrita en Java, existen formas de interactuar con ella desde otros lenguajes de programación.

Ejemplo

Cerveza.java

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Cerveza {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String nombre;
    private String tipo;
    private Double alcohol; // Porcentaje de alcohol
}
```

- @Entity permite convertir una clase en Java en una tabla en la BD.
- @Id marca un determinado campo como clave primaria en la tabla que se genera en la BD.
- @GeneratedValue(…) incrementa el Id de forma automática cada vez que se añade un registro.
- El resto de los campos se convierten a los tipos correspondientes.

Ejemplo

pom.xml

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

- Añadimos la dependencia de la BD h2.
- Creamos como Interfaz el CervezaRepository

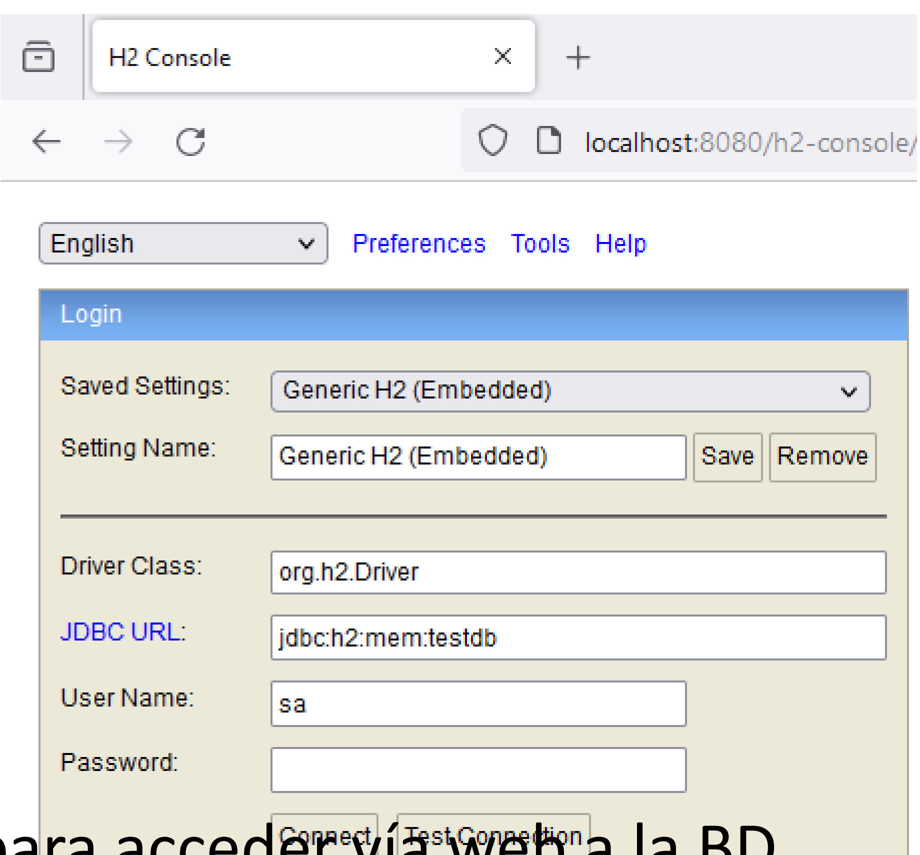
CervezaRepository.java

```
public interface CervezaRepository extends CrudRepository<Cerveza, Long> {
}
```

Ejemplo

application.properties

```
# Habilitar la consola H2
spring.h2.console.enabled=true

# (Opcional) Configurar el path de la consola H2
spring.h2.console.path=/h2-console

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```



- La primera línea nos permite habilitar la consola para acceder vía web a la BD.
- La segunda nos permite indicar la URL en la cual queremos que esté disponible la interfaz web.
- La tercera nos permite indicar donde está la BD.

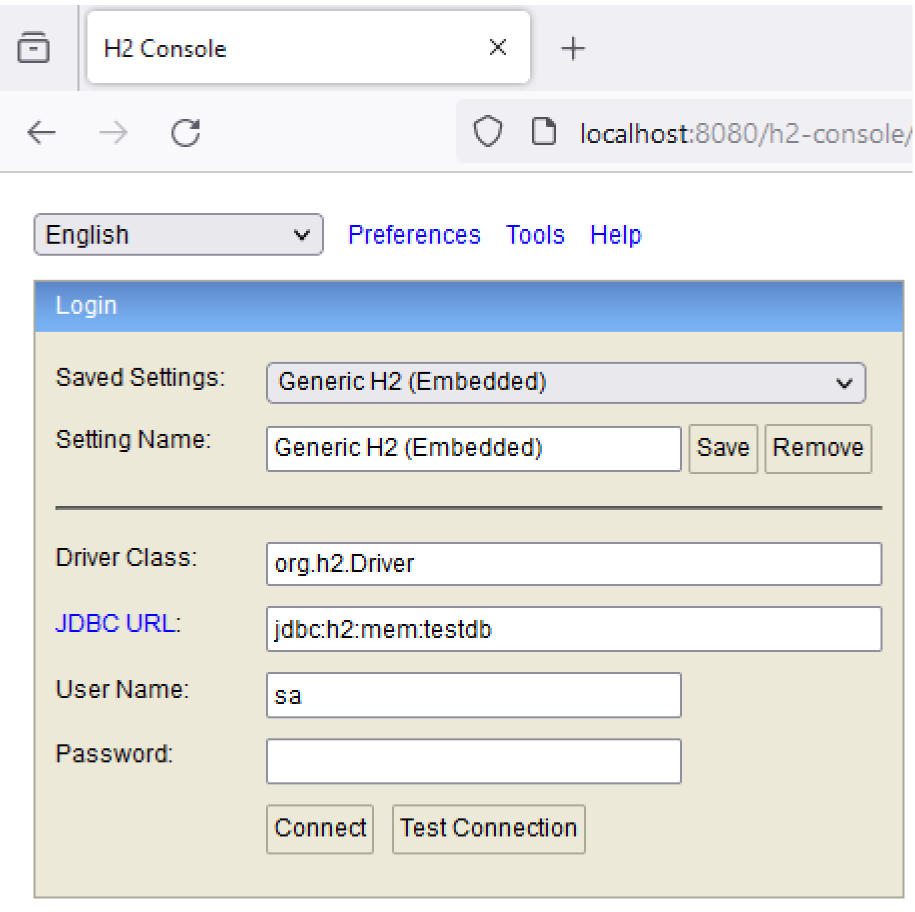
Ejemplo

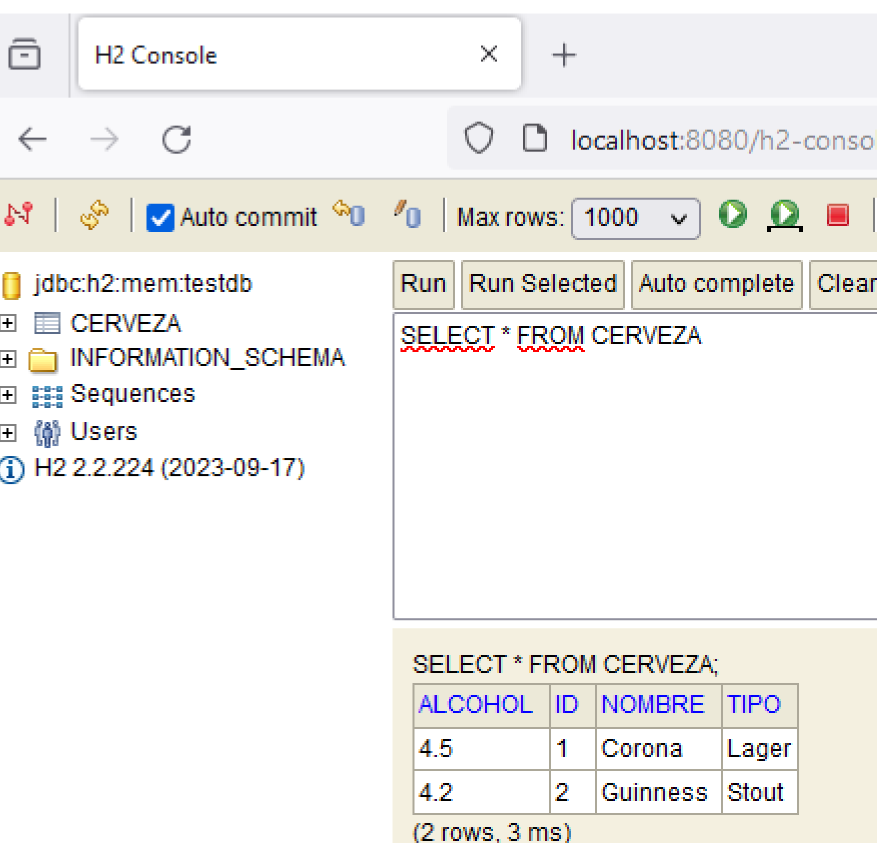
SpringDataEjemplo1Application.java

```
ConfigurableApplicationContext context =
SpringApplication.run(SpringDataEjemplo1Application.class, args);
CervezaRepository cervezaRepository = context.getBean(CervezaRepository.class);

// Añadir algunas cervezas
Cerveza cerveza1 = new Cerveza();
cerveza1.setNombre("Corona");
cerveza1.setTipo("Lager");
cerveza1.setAlcohol(4.5);
cervezaRepository.save(cerveza1);

Cerveza cerveza2 = new Cerveza();
cerveza2.setNombre("Guinness");
cerveza2.setTipo("Stout");
cerveza2.setAlcohol(4.2);
cervezaRepository.save(cerveza2);

// Leer y mostrar las cervezas
cervezaRepository.findAll().forEach(cerveza -> System.out.println(cerveza.getNombre() + " - " +
cerveza.getTipo() + " - " + cerveza.getAlcohol() + "%"));
```

Ejemplo

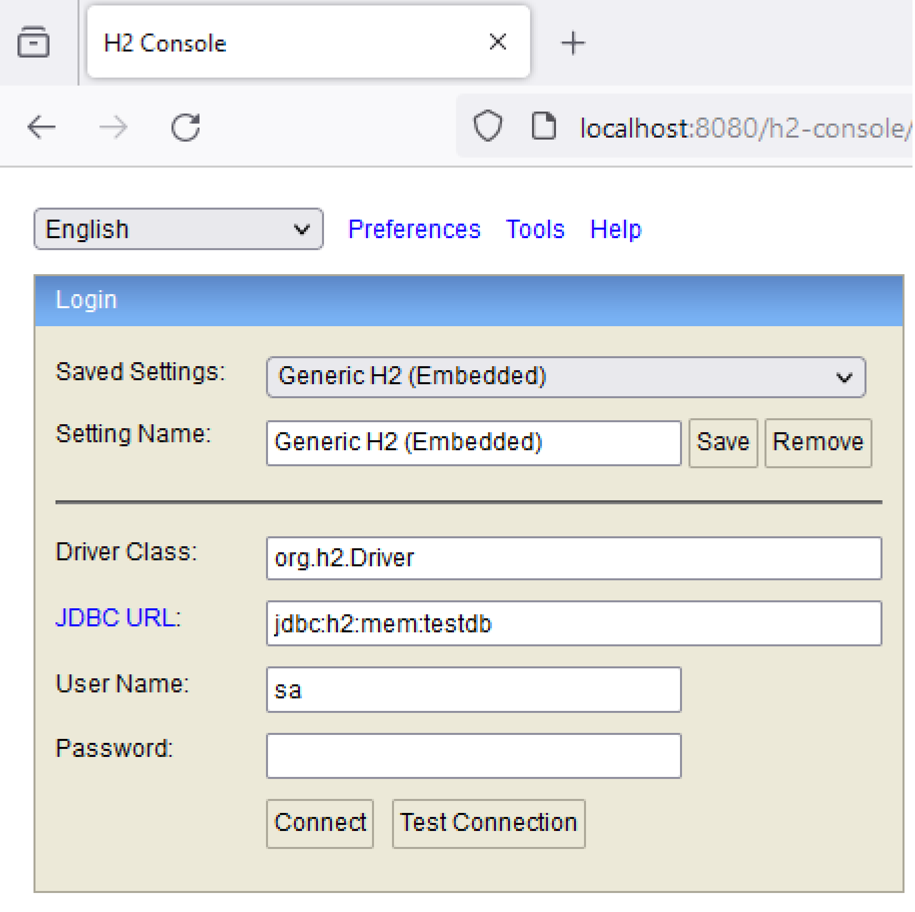
Resultado de la consulta

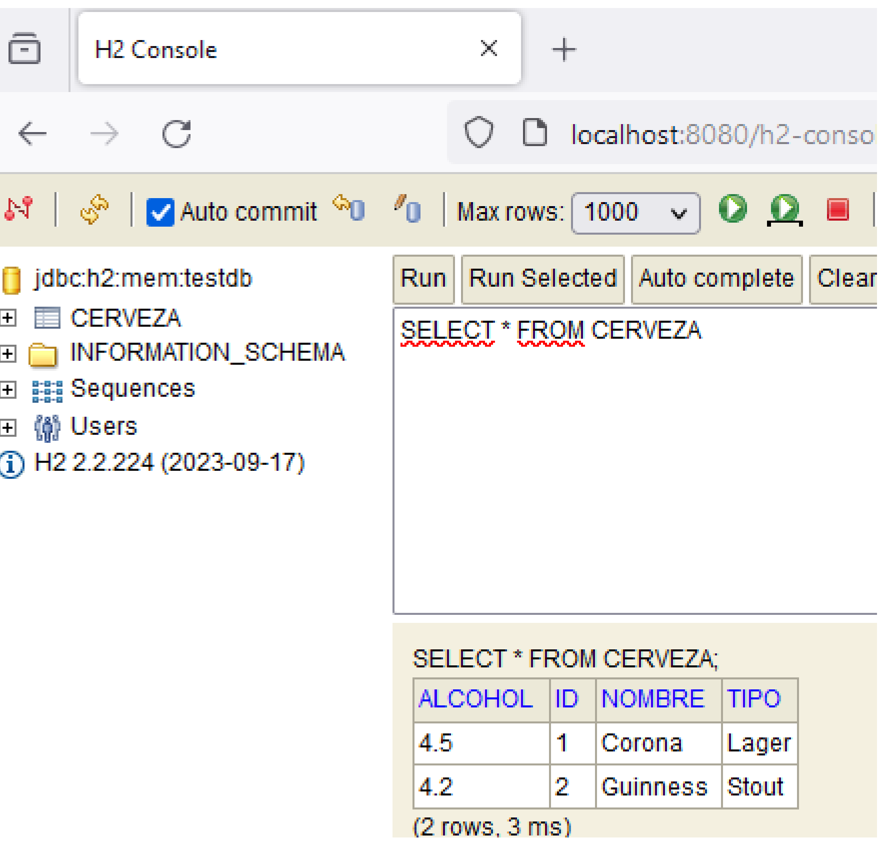




Ejemplo

Resultado de la consulta





Índice

- Introducción
- JPA
- Repositorios
- H2
- **Tipos de correspondencia**

Tipos de correspondencia

- La relación 1:N en JPA (Java Persistence API) es una de las relaciones más comunes.
- En este tipo de correspondencia una entidad (el "padre") puede tener múltiples instancias de otra entidad (los "hijos"), pero cada instancia hija está relacionada con una sola instancia padre.

Tipos de correspondencia

Autor.java

```
@Entity
public class Autor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private Set<Libro> libros = new HashSet<>();
    // Constructor, Getters y Setters
}
```

- Se añade @OneToMany para indicar en qué dirección se propaga la clave ajena.
- Se indica en autor el nombre del campo en la otra tabla.
- Se indica que el borrado será en cascada.

Tipos de correspondencia

Libro.java

```
@Entity
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String titulo;
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "autor_id")
    private Autor autor;
    // Constructor, Getters y Setters
}
```

- Se añade @ManyToOne para indicar en qué dirección viene la clave ajena.
- En términos generales, suele ser de ayuda para el programador.

Tipos de correspondencia

LibroRepository.java

```
public interface LibroRepository extends JpaRepository<Libro, Long> {
}
```

AutorRepository.java

```
public interface AutorRepository extends JpaRepository<Autor, Long> {
}
```

- Se añaden los repositorios de las entidades.
- Se puede usar el CrudRepository también.

Tipos de correspondencia

DataLoader.java

```
@Component
public class DataLoader implements CommandLineRunner
{
   @Autowired
   private AutorRepository autorRepository;

   @Autowired
   private LibroRepository libroRepository;

   @Override
   public void run(String... args) throws Exception {
      Autor autor = new Autor();
      autor.setNombre("Gabriel García Márquez");

      Libro libro = new Libro();
      libro.setTitulo("Cien años de soledad");
      libro.setAutor(autor);
      autor.getLibros().add(libro);

      autorRepository.save(autor);
   }
}
```

- DataLoader, que implementa CommandLineRunner, se usa para poblar la base de datos con un autor y un libro al iniciar la aplicación.
- Se añaden con el @Autowired ambos repositorios.
- El libroRepository es de utilidad para operaciones directas sobre libros.
- **Es necesario configurar H2 en el application.properties.**
- Una buena idea es revisar a través de la interfaz web el contenido de las tablas.

Tipos de correspondencia

- El tipo de correspondencia N:M en JPA (Java Persistence API) permite que múltiples instancias de una entidad estén asociadas con múltiples instancias de otra entidad.
- Este tipo de relación se gestiona mediante una tabla de unión que mantiene las referencias cruzadas entre las dos entidades.
- Consideremos un ejemplo clásico: la relación entre Estudiante y Curso, donde un estudiante puede inscribirse en varios cursos y cada curso puede tener varios estudiantes.

Tipos de correspondencia

Estudiante.java

```
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Estudiante {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nombre;

    @ManyToMany(mappedBy = "estudiantes")
    private Set<Curso> cursos = new HashSet<>();
}
```

pom.xml

```
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <version>1.18.20</version>
   <scope>provided</scope>
</dependency>
```

- Lombok es una dependencia que permite evitar el código repetido a través de anotaciones.

Tipos de correspondencia

Curso.java

```
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Curso {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String titulo;

    @ManyToMany
    @JoinTable(
            name = "curso_estudiante",
            joinColumns = @JoinColumn(name = "curso_id"),
            inverseJoinColumns = @JoinColumn(name = "estudiante_id")
    )
    private Set<Estudiante> estudiantes = new HashSet<>();
}
```

- @ManyToMany permite indicar el nombre la tabla que se genera a partir del tipo de correspondencia N:M.
- Además, es necesario indicar la clave primaria. Esta clave primaria está compuesta por dos claves ajenas. Una de estas claves sale de esta misma tabla, que se indica en el joinColumns. La otra parte de la clave viene de la otra tabla y se indica como inverseJoinColumns.
- No se recomienda el uso de @Data en este tipo de correspondencia.

Tipos de correspondencia

DataLoader.java

```
@Component
public class DataLoader implements CommandLineRunner {
    @Autowired
    private CursoRepository cursoRepository;
    @Autowired
    private EstudianteRepository estudianteRepository;
    @Override
    public void run(String... args) throws Exception {
        // Crear estudiantes
        Estudiante estudiante1 = new Estudiante();
        estudiante1.setNombre("Ana");
        Estudiante estudiante2 = new Estudiante();
        estudiante2.setNombre("Luis");
        // Crear cursos
        Curso curso1 = new Curso();
        curso1.setTitulo("Matemáticas");
        Curso curso2 = new Curso();
        curso2.setTitulo("Literatura");
        // Asignar estudiantes a cursos
        curso1.getEstudiantes().add(estudiante1);
        curso1.getEstudiantes().add(estudiante2);
        curso2.getEstudiantes().add(estudiante1);
        estudiante1.getCursos().add(curso1);
        estudiante1.getCursos().add(curso2);
        estudiante2.getCursos().add(curso1);
        // Guardar en la base de datos
        estudianteRepository.save(estudiante1);
        estudianteRepository.save(estudiante2);
        cursoRepository.save(curso1);
        cursoRepository.save(curso2);
    }
}
```

- Es necesario implementar los repositorios correspondientes a cada entidad.
- Es necesario modificar el application.properties con la configuración de H2.
- Utilizamos el DataLoader para cargar los datos del ejemplo.
- A continuación, se puede utilizar la interfaz web con H2 para comprobar el correcto funcionamiento.

Ejercicio

- Para generar un modelo relacional que incluya la entidad Cerveza y tenga relaciones de tipo (1:N) y (N:M), podemos introducir dos entidades adicionales: Cervecero (para representar a un productor de cerveza) e Ingrediente.
- El diagrama de este modelo mostraría tres entidades (Cervecero, Cerveza, Ingrediente) con las relaciones correspondientes. La relación 1:N entre Cervecero y Cerveza se maneja mediante una clave foránea en Cerveza. La relación N:M entre Cerveza e Ingrediente se gestiona a través de una tabla de unión que conecta las claves primarias de ambas entidades.

Ejercicio – Entidades y relaciones

- Entidad Cervecero:
  - Representa a los productores de cerveza.
  - Cada Cervecero puede producir varias Cervezas, pero cada Cerveza es producida por un único Cervecero. Esto establece una relación (1:N) entre Cervecero y Cerveza.
- Entidad Cerveza:
  - Ya definida, con atributos como id, nombre, tipo, y alcohol.
- Entidad Ingrediente:
  - Representa los ingredientes utilizados en la elaboración de cervezas.
  - Una Cerveza puede contener varios Ingredientes, y un Ingrediente puede ser utilizado en varias Cervezas. Esto establece una relación "Muchos a Muchos" (N:M) entre Cerveza y Ingrediente.

Ejercicio – Modelo relacional

- Cervecero
  - id: Clave primaria.
  - nombre: Nombre del cervecero o de la empresa cervecera.
- Cerveza
  - (Los campos de siempre)
  - cervecero_id: Clave foránea que referencia a Cervecero.
- Ingrediente
  - id: Clave primaria.
  - nombre: Nombre del ingrediente (por ejemplo, lúpulo, malta, levadura, etc.).
- Cerveza_Ingrediente (Tabla de Unión para la relación N:M)
  - cerveza_id: Clave foránea que referencia a Cerveza.
  - ingrediente_id: Clave foránea que referencia a Ingrediente.

Sistemas Distribuidos

Bloque II

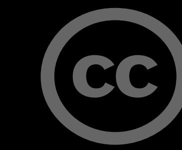
Desarrollo de sistemas distribuidos y aplicaciones web.

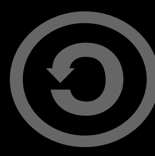
Tema 2.4:

Spring Data

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
“Atribución-CompartirIgual 4.0 Internacional” de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

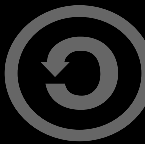
Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 3 .1

Fundamentos de las tecnologías de comunicación de aplicaciones

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **Falacias de la computación distribuida**
  - Tolerancia a fallos
  - Sincronización
  - El Problema de los Dos Generales
  - El Problema de los Generales Bizantinos
- Replicación y particionamiento
- Consistencia

Falacias de la computación distribuida

- Las "Falacias de la Computación Distribuida" son un conjunto de suposiciones erróneas hechas por los programadores cuando se desarrollan sistemas distribuidos.
- Estas falacias pueden llevar a errores de diseño y problemas operativos.

Falacias de la computación distribuida

- La red es confiable: Suponer que la red siempre estará disponible y que no se perderán mensajes puede llevar a fallos graves.
- La latencia es cero: Ignorar el tiempo de transmisión puede causar problemas en el rendimiento y la sincronización.
- El ancho de banda es infinito: Suponer un ancho de banda ilimitado puede llevar a diseños que no escalan bien.
- La red es segura: Ignorar las preocupaciones de seguridad puede exponer el sistema a ataques.

Falacias de la computación distribuida

- La topología no cambia: Las redes cambian; los sistemas deben ser diseñados para adaptarse a estos cambios.
- Hay un administrador: Suponer que habrá una intervención manual en la operación de la red es poco realista en grandes sistemas.
- El coste de transporte es cero: Ignorar el coste de los datos puede llevar a soluciones ineficientes.
- La red es homogénea: Suponer que todo el sistema usa las mismas normas y protocolos es una simplificación excesiva.

Tolerancia a fallos – Introducción

- La tolerancia a fallos es un concepto clave en la computación distribuida y en el diseño de sistemas en general.
- Se refiere a la capacidad de un sistema para seguir operando de manera satisfactoria en la presencia de fallos parciales.
- Este concepto es crucial para garantizar la fiabilidad y la disponibilidad continua de un sistema, especialmente en entornos donde los fallos son inevitables debido a la complejidad del hardware, del software o de la red.

Tolerancia a fallos – Conceptos clave

- Detección de fallos: Identificar rápidamente las partes del sistema que han fallado.
- Aislamiento de fallos: Limitar los efectos del fallo a la menor parte posible del sistema para evitar una falla total.
- Recuperación de fallos: Restaurar la operación normal del sistema después de un fallo, ya sea mediante la reanudación del componente fallido o su sustitución.

Tolerancia a fallos – Conceptos clave

- Redundancia: Tener componentes o sistemas duplicados para que, en caso de fallo, otro pueda tomar el relevo. La redundancia puede ser:
  - De hardware (servidores duplicados).
  - De software (múltiples instancias de un servicio) o
  - De datos (como replicación de bases de datos).
- Reintento: Capacidad para intentar nuevamente operaciones que han fallado y manejar errores de manera ágil.
- Persistencia del estado: Diseñar sistemas sin estado facilita la recuperación, mientras que, en sistemas con estado, es crucial asegurar la persistencia y coherencia de este.

Tolerancia a fallos – Importancia

- Disponibilidad: Un sistema tolerante a fallos puede ofrecer una alta disponibilidad, lo cual es esencial para servicios críticos y operaciones empresariales continuas.
- Fiabilidad: Aumenta la confianza en el sistema, ya que puede manejar fallos sin interrumpir el servicio.
- Seguridad: En algunos casos, la tolerancia a fallos también implica mantener la seguridad del sistema en presencia de componentes defectuosos.

Tolerancia a fallos – Desafíos

- Complejidad: Diseñar e implementar sistemas tolerantes a fallos puede ser complejo y costoso.
- Rendimiento: La redundancia y los mecanismos de recuperación pueden afectar el rendimiento del sistema.
- Pruebas: Probar completamente la tolerancia a fallos puede ser difícil, ya que implica simular una variedad de fallos.

Tolerancia a fallos – Ejemplos

- Sistemas de Bases de Datos: Usan replicación y transacciones para asegurar la integridad de los datos.
- Sistemas de Archivos Distribuidos: Como HDFS, que almacena múltiples copias de datos para proteger contra la pérdida de datos.
- Infraestructura de Red: Diseñada para re-rutear el tráfico en caso de fallo de un enlace.

Sincronización – Introducción

- La sincronización en sistemas distribuidos es un desafío clave que implica:
  - Coordinar acciones.
  - Mantener la consistencia de los datos a través de múltiples nodos.
  - Estos no están necesariamente operando al unísono.
- En sistemas distribuidos, los nodos pueden estar separados físicamente y comunicarse a través de una red, lo que introduce incertidumbre y retardo en la comunicación.

Sincronización – Conceptos clave

- Sincronización de relojes:
  - Mantener los relojes de los diferentes nodos sincronizados es crucial para operaciones que dependen del tiempo.
  - Algoritmos como NTP (Network Time Protocol) se utilizan para sincronizar relojes a una fuente de tiempo común.
- Sincronización de datos:
  - Asegurar que todos los nodos tengan una vista consistente de los datos.
  - Complejo debido a la latencia de la red y las actualizaciones concurrentes.
  - Mecanismos como bloqueo distribuido, transacciones distribuidas y protocolos de consenso (como Paxos o Raft) se utilizan para mantener la coherencia de los datos.
- Sincronización de procesos:
  - Coordinar la ejecución de procesos en diferentes nodos.
  - Esto incluye asegurarse de que las operaciones se realicen en un orden específico cuando sea necesario.

Sincronización – Desafíos

- Latencia de red: La comunicación a través de la red introduce retrasos, lo que puede llevar a inconsistencias.
- Fallos y particiones de red: Los fallos en los nodos o en la red pueden interrumpir los mecanismos de sincronización.
- Concurrencia: La gestión de acceso concurrente a recursos compartidos es compleja y requiere mecanismos de sincronización.

Sincronización – Estrategias y mecanismos

- Algoritmos de Exclusión Mutua Distribuida: Garantizan que solo un nodo a la vez pueda realizar una operación crítica.
- Relojes Lógicos y Físicos: Los relojes lógicos (como los relojes de Lamport o relojes vectoriales) se utilizan para mantener un orden causal de eventos, mientras que los relojes físicos se sincronizan para reflejar el tiempo real.
- TrueTime de Google: permite un nivel de consistencia y coordinación que es difícil de lograr en sistemas distribuidos tradicionales

Sincronización – Ejemplos

- Bases de Datos Distribuidas: Mantienen la consistencia de los datos a través de transacciones distribuidas y protocolos de replicación.
- Sistemas de Archivos Distribuidos: Garantizan la coherencia de los datos entre diferentes nodos.
- Computación en la Nube y Microservicios: Coordinan múltiples servicios y procesos que se ejecutan en paralelo y a menudo de forma independiente.

El Problema de los Dos Generales

- Escenario:
  - El problema describe dos generales que planean atacar una ciudad fortificada.
  - Están acampados en colinas separadas, con la ciudad en el valle entre ellos.
- Desafío de comunicación:
  - Para tener éxito, deben atacar simultáneamente.
  - Sin embargo, la única forma de comunicarse es a través de mensajeros que deben atravesar el valle, donde hay riesgo de ser capturados por el enemigo.
- Dilema central:
  - Cada general no puede estar seguro de que sus mensajes de confirmación para coordinar el ataque hayan sido recibidos.
  - Un general envía un mensajero para proponer un plan de ataque, pero no puede estar seguro de que el mensaje haya llegado.
  - Si recibe una confirmación, no puede estar seguro de que su confirmación de la confirmación haya llegado, y así sucesivamente.

El Problema de los Dos Generales

- Confianza en la comunicación:
  - El problema resalta la imposibilidad de garantizar la fiabilidad absoluta en la comunicación sobre canales no fiables.
  - Es especialmente relevante en SSDD donde la comunicación entre nodos puede ser incierta.
- Acuerdo y consenso: Muestra las dificultades para alcanzar un consenso o acuerdo en la presencia de incertidumbre comunicacional, lo que es un aspecto fundamental en la computación distribuida.

El Problema de los Generales Bizantinos

- Escenario:
  - Un grupo de generales bizantinos, cada uno al mando de una parte del ejército bizantino, que deben acordar un plan común de acción (como atacar o retirarse) frente a una ciudad enemiga.
- Comunicación:
  - Los generales están separados y deben comunicar sus planes a través de mensajeros.
- Desafío:
  - Al menos uno de los generales (o más) podría ser un traidor que intentará confundir a los demás para que el plan falle.
  - El desafío es llegar a un consenso en presencia de estos elementos no confiables.

El Problema de los Generales Bizantinos

- Confianza y traición: El problema resalta la dificultad de alcanzar un consenso en un sistema distribuido cuando hay componentes no confiables que pueden proporcionar información falsa o contradictoria.
- Tolerancia a Fallos Bizantinos: Esto llevó al concepto de Tolerancia a Fallos Bizantinos (BFT), que se refiere a la capacidad de un sistema para continuar operando correctamente incluso en presencia de nodos defectuosos o malintencionados.

El Problema de los Generales Bizantinos

- Algoritmos de Consenso Bizantino:
  - Se han desarrollado varios algoritmos para tratar de resolver este problema en la práctica, como el algoritmo PBFT (Practical Byzantine Fault Tolerance).
  - Estos algoritmos son fundamentales en áreas como las redes de blockchain y criptomonedas, donde la confiabilidad y la seguridad son cruciales.
- Complejidad y recursos: Los algoritmos que abordan la Tolerancia a Fallos Bizantinos suelen:
  - Ser complejos.
  - Requerir mayor comunicación y coordinación entre los nodos.
  - Como consecuencia, aumentar el uso de recursos y la latencia.

Índice

- Falacias de la computación distribuida
- **Replicación y particionado**
  - Replicación
  - Particionado
  - Transacciones
- Consistencia

Replicación y particionado – Introducción

- La replicación y el particionamiento son técnicas fundamentales en el diseño de bases de datos distribuidas y sistemas de almacenamiento.
- Estas estrategias se utilizan para mejorar el rendimiento, la disponibilidad y la escalabilidad de los sistemas, así como para asegurar la resistencia a fallos.
- La replicación implica mantener copias de los mismos datos en múltiples nodos o servidores. Esto mejora la disponibilidad y la resistencia a fallos, pero introduce desafíos en mantener la consistencia de los datos.
- El particionamiento divide los datos en diferentes nodos para mejorar la escalabilidad y el rendimiento.

Replicación

- Replicación basada en el líder
  - Un nodo (líder) maneja todas las escrituras y actualizaciones. Las replicas (seguidores) se sincronizan con el líder.
  - Común en sistemas que requieren fuerte consistencia y en situaciones donde es aceptable tener un punto único de escritura.
- Replicación multilíder
  - Varios nodos actúan como líderes, permitiendo escrituras simultáneas en diferentes nodos.
  - Útil para mejorar la escritura y la disponibilidad de datos en múltiples regiones geográficas, aunque puede complicar la resolución de conflictos.
- Replicación sin líderes (quorum)
  - Las escrituras y lecturas se hacen en un conjunto de nodos, y se requiere un quórum (mayoría) para considerar una operación exitosa.
  - Equilibra la carga y aumenta la disponibilidad, pero puede ser más complejo manejar la consistencia.

Particionado

- Por Clave-Valor
  - Los datos se particionan basándose en las claves de los registros. Cada partición contiene un rango específico de claves.
  - Simplifica el acceso y la distribución de los datos, pero puede llevar a particiones desequilibradas si las claves no están bien distribuidas.
- Índices secundarios
  - En sistemas con índices secundarios, el particionamiento se vuelve más complejo, ya que los índices pueden apuntar a registros en diferentes particiones.
  - Necesario en consultas complejas, pero puede aumentar la complejidad del sistema y afectar el rendimiento.
- Rebalanceado
  - Implica mover datos entre nodos para mantener un balance de carga. Es esencial en sistemas dinámicos donde la carga o el tamaño de los datos cambian con el tiempo.
  - Mantiene el sistema equilibrado y eficiente, pero requiere mecanismos sofisticados para manejar el movimiento de datos sin afectar la disponibilidad.

Transacciones

- Las transacciones en sistemas informáticos, especialmente en bases de datos y sistemas distribuidos, son fundamentales para garantizar la integridad y coherencia de los datos.
- Una transacción es una secuencia de operaciones que se tratan como una única unidad lógica de trabajo.
- En bases de datos, las transacciones aseguran que las operaciones sobre los datos se realicen de manera segura y confiable, incluso en presencia de fallos y errores.

Transacciones – Propiedades

- Atomicidad:
  - Garantiza que todas las operaciones en la transacción se ejecutan o ninguna se ejecuta.
  - Si una parte de la transacción falla, el sistema revierte todas las operaciones de la transacción.
- Consistencia:
  - Asegura que la transacción lleva la base de datos de un estado consistente a otro.
  - No se deben violar las restricciones de integridad de la base de datos.
- Aislamiento:
  - Determina cómo y cuándo los cambios realizados por una transacción son visibles para otras transacciones.
  - El aislamiento ayuda a prevenir conflictos de concurrencia.
- Durabilidad:
  - Una vez que una transacción se ha comprometido, sus cambios son permanentes, incluso en caso de fallo del sistema.

Índice

- Falacias de la computación distribuida
- Replicación y particionamiento
- **Consistencia**
  - Linealizabilidad
  - Consistencia causal
  - Teorema CAP
  - 2FC
  - Teorema FLP
  - Algoritmos de Consenso

Linealizabilidad – Introducción

- Es el tipo más fuerte de consistencia que un sistema distribuido puede ofrecer y juega un papel crucial en garantizar la fiabilidad y la previsibilidad de las operaciones en dichos sistemas.
- Es una propiedad de los SSDD en la que las operaciones realizadas parecen haber ocurrido en un orden secuencial único, incluso si se ejecutan en paralelo en diferentes nodos.
- Una serie de operaciones parece ejecutarse instantáneamente en un punto entre su inicio y finalización. Esto significa que el sistema debería comportarse como si cada operación se ejecutara una tras otra, sin solapamiento.

Linealizabilidad – Conceptos clave

- Orden Global: Existe un orden global de todas las operaciones que es consistente con el orden en que se ejecutaron en cada nodo individual.
- Predictibilidad: Los usuarios pueden predecir el resultado de sus operaciones basándose en el conocimiento de operaciones anteriores.
- Consistencia a través de nodos: Los cambios realizados en un nodo son visibles para todos los demás nodos en un orden que refleja la secuencia real de operaciones.

Consistencia causal – Introducción

- Es un modelo de consistencia en SSDD que relaja algunas de las restricciones de los modelos de consistencia más estrictos, como la linealizabilidad
- El objetivo es mejorar el rendimiento y la escalabilidad.
- A diferencia de la consistencia fuerte, que puede ser costosa en términos de rendimiento y no siempre es necesaria, la consistencia causal se centra en mantener un orden lógico de las operaciones que refleja las causas y los efectos dentro del sistema.

Consistencia causal – Conceptos clave

- Orden Causal:
  - Asegura que si una operación A "causa" otra operación B en cualquier nodo del sistema, entonces todos los nodos verán A antes que B. Esto significa que la secuencia de eventos respeta la causalidad lógica.
- Independencia de operaciones no relacionadas:
  - Operaciones que no están causalmente relacionadas pueden ser vistas en un orden diferente por diferentes nodos.
  - Esto permite más flexibilidad y un mejor rendimiento en comparación con la consistencia fuerte.
- Aplicabilidad:
  - En sistemas donde el mantenimiento de un orden global exacto es menos crítico, como en redes sociales, sistemas de documentos colaborativos, y sistemas de mensajería.
- Diseño del Sistema:
  - Determinar las dependencias causales entre operaciones puede ser complicado.
- Compromiso:
  - Necesidad de equilibrar un orden coherente con los beneficios del rendimiento y la escalabilidad.

Teorema CAP – Introducción

- El Teorema CAP, también conocido como el Principio de Brewer, es un concepto fundamental en la teoría de sistemas distribuidos.
- Formulado por Eric Brewer en 2000, el teorema establece que en cualquier sistema de computación distribuido, solo se pueden garantizar dos de las siguientes tres propiedades al mismo tiempo:
  - Consistencia (Consistency)
  - Disponibilidad (Availability)
  - Tolerancia a Particiones (Partition Tolerance)

Teorema CAP – Conceptos clave

- Consistencia (Consistency)
  - Cada lectura recibe la escritura más reciente o un error.
  - Significa que todos los nodos ven los mismos datos al mismo tiempo.
  - No hay discrepancias en los datos entre los distintos nodos del sistema, incluso después de una actualización de datos.
  - Por ejemplo, en una base de datos distribuida, después de que un dato es actualizado, todas las futuras lecturas reflejarán ese cambio (o devolverán un error si el cambio no se puede reflejar)
- Disponibilidad (Availability)
  - Cada solicitud recibe una respuesta, sin garantizar que contenga la última versión del dato.
  - Significa que el sistema siempre responde a las solicitudes de los clientes, aunque esos datos puedan no ser los más recientes.
  - Por ejemplo, un sistema de base de datos distribuida responde a todas las consultas de lectura/escritura, pero los datos que devuelve pueden no ser los más actualizados si parte del sistema está desconectado.

Teorema CAP – Conceptos clave

- Tolerancia a Particiones (Partition Tolerance)
  - El sistema sigue funcionando a pesar de la pérdida arbitraria de mensajes o fallos en parte del sistema.
  - Significa que el sistema puede seguir operando incluso si hay fallas de red que impiden que algunos nodos se comuniquen entre sí.
  - Por ejemplo, en una red distribuida, si un segmento se desconecta, los nodos restantes pueden seguir operando y procesando solicitudes.
- Implicaciones del Teorema CAP
  - Elección de diseño: No es posible diseñar un sistema distribuido que garantice simultáneamente todas estas tres propiedades al 100%.
  - Realidad de las particiones: Las particiones son inevitables en cualquier red distribuida, por lo que la tolerancia a particiones suele ser una necesidad.
  - Escenarios de aplicación: El teorema CAP es especialmente relevante en el diseño de bases de datos distribuidas y sistemas de almacenamiento de datos, donde la gestión de la consistencia y la disponibilidad son críticas.

Transacción en Dos Fases – Introducción

- La Transacción en Dos Fases (2PC, por sus siglas en inglés) es un protocolo de control de transacciones distribuidas utilizado en BBDD y SSDD para garantizar la atomicidad en transacciones que involucran múltiples nodos o recursos.
- El 2PC es fundamental para mantener la integridad de los datos en sistemas distribuidos donde las operaciones deben ser atómicas (es decir, todas las operaciones en la transacción deben completarse con éxito o, en caso de fallo, ninguna debe aplicarse).

Transacción en Dos Fases – Funcionamiento

- Fase de votación (Preparación)
  - Coordinador: Un nodo actúa como coordinador de la transacción y envía un mensaje de "preparación" a todos los participantes (nodos implicados en la transacción).
  - Participantes: Cada participante ejecuta la transacción hasta el punto de compromiso, pero no realiza cambios permanentes. Luego, vota "sí" (preparado) si puede comprometer la transacción o "no" si no puede.
  - Respuesta: Los participantes envían su voto al coordinador.
- Fase de compromiso
  - Decisión basada en Votos: Si todos los participantes votan "sí", el coordinador envía un mensaje de "compromiso" a todos los participantes. Si alguno vota "no", el coordinador envía un mensaje de "abortar".
  - Acción de los participantes: Después de recibir el mensaje de compromiso, los participantes completan la operación y liberan los recursos y bloqueos que mantenían. Si reciben un mensaje de abortar, deshacen las operaciones realizadas en la fase de preparación.
  - Confirmación de Compromiso/Aborto: Los participantes informan al coordinador una vez que han completado el compromiso o el aborto.

Transacción en Dos Fases – Conceptos clave

- Atomicidad: Asegura que todas las partes de la transacción se comprometen o ninguna se compromete, manteniendo la integridad de los datos.
- Bloqueo de recursos: Durante la fase de preparación, los recursos que están siendo utilizados por la transacción se bloquean, lo que puede llevar a bloqueos y problemas de rendimiento en sistemas de alta concurrencia.
- Resistencia a fallos: Aunque el 2PC puede manejar fallos, si el coordinador falla después de enviar mensajes de compromiso, pero antes de recibir confirmaciones, puede resultar en incertidumbre sobre el estado de la transacción.

Teorema FLP – Introducción

- El Teorema FLP, nombrado por sus autores Michael Fischer, Nancy Lynch y Michael Paterson, es fundamental en la teoría de SSDD.
- Presentado en 1985, el teorema aborda la imposibilidad de alcanzar el consenso en sistemas distribuidos asincrónicos si se permite incluso un único fallo.
- Se centra en sistemas distribuidos asincrónicos, donde no hay suposiciones sobre la velocidad de procesamiento o los tiempos de transmisión de mensajes.
- El teorema demuestra que, en tales sistemas, no es posible garantizar que todos los nodos alcancen el mismo consenso (decisión) en presencia de al menos un fallo de tipo "crash" (un nodo deja de funcionar).

Teorema FLP – Implicaciones

- Consenso determinista: El resultado principal es que no existe un algoritmo determinista que pueda garantizar el consenso en un sistema distribuido asincrónico si se permite la posibilidad de un fallo.
- Seguridad vs. Vivacidad: El teorema muestra que no se pueden garantizar simultáneamente la seguridad (todos los nodos no defectuosos llegan al mismo acuerdo) y la vivacidad (eventualmente se toma una decisión) en tales sistemas.

Teorema FLP – Importancia

- Desafío de diseño: El teorema FLP presenta un gran desafío en el diseño de algoritmos de consenso para sistemas distribuidos, especialmente aquellos que necesitan operar de manera confiable en presencia de fallos.
- Compromisos prácticos: En la práctica, los sistemas deben hacer compromisos, a menudo favoreciendo la vivacidad sobre la seguridad total o viceversa, dependiendo de los requisitos del sistema.

Algoritmos de consenso: Paxos – Introducción

- Paxos es un protocolo de consenso desarrollado por Leslie Lamport en los años 90, ampliamente reconocido y utilizado en SSDD.
- Su objetivo principal es alcanzar un acuerdo entre los nodos de un sistema distribuido, especialmente en situaciones donde los nodos pueden experimentar fallos.
- Paxos se divide en varias fases y roles.
- Roles en Paxos:
    - Proposers: Proponen valores a ser acordados por el sistema.
    - Acceptors: Votan para aceptar o rechazar propuestas.
    - Learners: Aprenden el valor acordado por los acceptors.

Algoritmos de consenso: Paxos – Fases

- Fase 1 - Preparación:
  - Un proposer genera una propuesta con un número de secuencia único.
  - Envía una solicitud a los acceptors para que acepten la propuesta.
  - Los acceptors responden, indicando si aceptarán la propuesta basándose en el número de secuencia.
- Fase 2 - Aceptación:
  - Si la mayoría de los acceptors aprueban la propuesta, el proposer envía un valor a ser aceptado junto con el número de propuesta.
  - Los acceptors votan sobre la propuesta final.
  - Si la mayoría acepta, el valor se considera elegido.
- Elección del Valor:
  - El valor elegido se comunica a los learners.
  - El sistema asegura que, a pesar de los fallos, un único valor consensuado se elija.

Algoritmos de consenso: Paxos – Conceptos clave

- Tolerancia a fallos: Paxos puede tolerar fallos de nodos hasta cierto punto sin perder la capacidad de alcanzar un consenso.
- Consenso distribuido: Permite a un conjunto de nodos, que pueden no estar siempre disponibles o ser confiables, llegar a un acuerdo sobre un valor específico.
- Robustez: Es resistente a problemas de red y fallos de nodos individuales.

Algoritmos de consenso: Paxos – Limitaciones

- Complejidad: Paxos es conocido por ser difícil de entender y de implementar correctamente.
- Rendimiento: Puede ser lento debido a la cantidad de rondas de comunicación necesarias, especialmente en redes con alta latencia o en sistemas con alta tasa de conflictos de propuestas.
- Adaptabilidad: Ajustar Paxos a casos de uso específicos o integrarlo en sistemas existentes puede ser complejo.

Algoritmos de consenso: Paxos – Aplicaciones

- Sistemas de almacenamiento distribuido: Como en sistemas de bases de datos distribuidas y sistemas de archivos distribuidos.
- Coordinación de servicios: Utilizado en sistemas de orquestación y coordinación como Apache ZooKeeper.
- Infraestructura de cloud computing: Para manejar la consistencia de datos y el estado en entornos de computación en la nube.

Algoritmos de consenso: Zab – Introducción

- Zab, que significa "Zookeeper Atomic Broadcast", es un protocolo de consenso para el sistema de coordinación de Apache ZooKeeper.
- ZooKeeper es un servicio centralizado para mantener la configuración de información, nombrar, proporcionar sincronización distribuida, y proporcionar servicios de agrupación en SSDD.
- Zab es fundamental para garantizar que ZooKeeper pueda manejar roles de manera confiable, especialmente en entornos donde se pueden experimentar fallos de nodos.
- Roles en Zab:
  - Líder: Maneja todas las solicitudes de escritura y coordina la actualización de los seguidores.
  - Seguidores: Mantienen una copia del estado y responden a las solicitudes de lectura.

Algoritmos de consenso: Zab – Fases

- Fase de descubrimiento: Cuando se elige un nuevo líder, establece el estado más actualizado del sistema.
- Fase de sincronización: El líder sincroniza su estado con los seguidores para asegurarse de que todos tengan los mismos datos.
- Fase de difusión: Una vez sincronizados, el líder puede difundir nuevas transacciones a los seguidores.
- Transacciones:
  - Cada cambio en el estado se maneja como una transacción.
  - Las transacciones se difunden de manera atómica y en orden.

Algoritmos de consenso: Zab – Conceptos clave

- Consistencia: Asegura que todos los nodos (seguidores) mantengan una copia consistente del estado.
- Recuperación de fallos: En caso de falla del líder, el protocolo puede elegir y cambiar a un nuevo líder, manteniendo la continuidad del servicio.
- Orden de entrega garantizado: Las transacciones se entregan en el mismo orden en todos los nodos.

Algoritmos de consenso: Zab – Limitaciones

- Dependencia del líder: El rendimiento y la disponibilidad del sistema dependen en gran medida del líder, lo que puede ser un punto único de fallo.
- Escalabilidad: Aunque Zab maneja bien los entornos distribuidos, la escalabilidad puede verse afectada en sistemas con un gran número de nodos debido a la sobrecarga de coordinación.

Algoritmos de consenso: Raft – Introducción

- Raft es un algoritmo de consenso para sistemas de computación distribuida que se centra en la comprensibilidad y la facilidad de implementación.
- Diseñado por Diego Ongaro y John Ousterhout en 2014, Raft ofrece una alternativa más accesible al complejo algoritmo Paxos, manteniendo un enfoque en la seguridad y eficiencia.
- https://thesecretlivesofdata.com/raft/

Algoritmos de consenso: Raft – Fases

Elección del Líder:

- Inicio de la Elección:
  - Si un nodo no recibe comunicación del líder en un tiempo determinado (denominado "tiempo de espera"), se convierte en candidato y comienza una elección.
- Votación:
  - El candidato solicita votos de otros nodos. Si recibe votos de la mayoría de los nodos, se convierte en el líder.
- Heartbeat:
  - Una vez elegido, el líder envía mensajes periódicos de "heartbeat" a todos los nodos para mantener su autoridad y prevenir nuevas elecciones.

Algoritmos de consenso: Raft – Fases

Replicación de log:

- Propuestas de Clientes:
  - El líder recibe propuestas de cambio de estado (como comandos de clientes) y las agrega a su log.
- Replicación:
  - El líder replica estas entradas de log a los demás nodos (seguidores).
- Compromiso:
  - Una vez que una entrada de log ha sido replicada en la mayoría de los nodos, se considera "comprometida" y se aplica al estado del sistema.

Algoritmos de consenso: Raft – Fases

Manejo de fallos y cambios de Líder:

- Fallos de Líder:
  - Si el líder falla, los nodos esperarán un tiempo de espera sin recibir el "heartbeat" y luego comenzarán una nueva elección.
- Consistencia de log:
  - Si los logs entre el líder y los seguidores difieren, el líder sobrescribe las entradas inconsistentes en los seguidores con las suyas para garantizar la consistencia.

Algoritmos de consenso: Raft – Conceptos clave

- División del problema: Raft separa claramente el problema del consenso en dos partes: la elección del líder y la replicación de log.
- Seguridad y robustez: Raft asegura que el log replicado sea consistente y que, una vez comprometida una entrada de log, permanezca como parte permanente del log.
- Comprensibilidad: Una de las metas principales de Raft es ser más fácil de entender y razonar que otros algoritmos de consenso.

Algoritmos de consenso: PoW – Introducción

- El "Proof of Work" (PoW), es un mecanismo fundamental utilizado en varios sistemas criptográficos, siendo más notablemente asociado con Bitcoin y otras criptomonedas.
- Su propósito principal es prevenir el spam y los ataques de denegación de servicio.

Algoritmos de consenso: PoW – Fases

- Desafío Criptográfico:
  - En PoW, se plantea un desafío criptográfico que requiere un esfuerzo significativo para resolverlo.
  - Este desafío generalmente involucra encontrar un valor (nonce) que, cuando se combina con los datos de la transacción y se pasa a través de una función hash criptográfica, produce un hash que cumple con ciertos criterios (por ejemplo, un número específico de ceros al principio).
- Minería:
  - Este proceso de resolver el desafío criptográfico se conoce como "minería".
  - Los mineros utilizan computadoras de alto rendimiento para realizar cálculos hash de manera repetitiva hasta que uno encuentra la solución correcta.
- Recompensa:
  - Una vez que un minero resuelve el desafío, presenta la solución a la red para verificación.
  - Si la solución es correcta y cumple con las reglas de la red, el minero es recompensado, por ejemplo, con bitcoins en el caso de la red Bitcoin.
- Creación de un Nuevo Bloque:
  - La solución se utiliza para validar un bloque de transacciones, que luego se agrega a la blockchain.

Algoritmos de consenso: PoW – Conceptos clave

- Seguridad: Dificulta la manipulación de la blockchain, ya que requiere una cantidad significativa de esfuerzo computacional para resolver los desafíos y validar transacciones.
- Descentralización: Permite a cualquier persona con el hardware necesario participar en el proceso de minería, manteniendo la red descentralizada.
- Prevención de doble gasto: Asegura que las mismas monedas digitales no se gasten más de una vez.

Algoritmos de consenso: PoW – Limitaciones

- Consumo de energía: PoW es intensivo en energía, lo que ha generado preocupaciones sobre su impacto ambiental, especialmente en redes como Bitcoin que requieren una gran cantidad de esfuerzo computacional.
- Centralización de la minería: La minería se ha vuelto cada vez más centralizada debido a la necesidad de hardware especializado y acceso a electricidad barata.
- Escalabilidad: La velocidad a la que se pueden agregar nuevos bloques y procesar transacciones es limitada, lo que plantea desafíos de escalabilidad.

Algoritmos de consenso: PoW – Introducción

- Proof of Stake (PoS), es un mecanismo alternativo al "Proof of Work" (PoW) para alcanzar el consenso en las redes de blockchain.
- Fue introducido como una solución a algunos de los problemas asociados con el PoW, particularmente el alto consumo de energía.
- En PoS, la capacidad de un nodo para validar bloques de transacciones se basa en la cantidad de moneda (o 'stake') que tiene en la red, en lugar de su capacidad para resolver desafíos criptográficos como en PoW.

Algoritmos de consenso: PoW – Fases

- Selección de validadores:
  - Los validadores son seleccionados para crear un nuevo bloque, basándose en su participación en la red. Esta participación se mide generalmente por la cantidad de moneda que poseen y, en algunos casos, por el tiempo durante el cual la han mantenido.
- Validación de bloques:
  - Los validadores son responsables de verificar las transacciones y añadir nuevas a la blockchain.
  - A diferencia del PoW, no hay un proceso de minería intensivo en recursos.
- Recompensas:
  - Los validadores son recompensados con tarifas de transacción o, en algunos casos, con nuevas monedas.
  - No se requiere un hardware especializado, lo que reduce significativamente el consumo de energía en comparación con PoW.

Algoritmos de consenso: PoW – Ventajas

- Eficiencia energética: Elimina la necesidad de poder computacional intensivo y, por lo tanto, consume mucha menos energía que PoW.
- Menor riesgo de centralización: Dado que no se requiere equipo especializado, existe un menor riesgo de que la minería se centralice en aquellos con recursos para comprar hardware especializado.
- Seguridad: Algunas variantes de PoS incorporan medidas adicionales para garantizar la seguridad y reducir la posibilidad de ataques maliciosos, como el "Nothing at Stake".

Algoritmos de consenso: PoW – Desventajas

- Riqueza = Poder: Una crítica es que PoS favorece a aquellos que ya tienen grandes cantidades de moneda, potencialmente llevando a una centralización del poder financiero.
- Nada en Juego: Existe el desafío del problema de "Nothing at Stake", donde los validadores pueden querer validar en múltiples cadenas de bloques para maximizar sus recompensas, aunque esto se aborda en diseños más recientes de PoS.
- Ataques del 51%: Mientras que en PoW un ataque del 51% requiere controlar el 51% del esfuerzo de cómputo, en PoS, esto implicaría poseer el 51% de la moneda, lo cual es teóricamente más difícil de lograr.

Sistemas Distribuidos

Bloque II

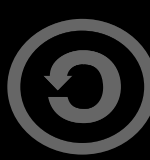
Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 3 .1

Fundamentos de las tecnologías de comunicación de aplicaciones

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia
"Atribución-CompartirIgual 4.0 Internacional" de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Sistemas Distribuidos

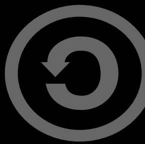
Bloque II

Desarrollo de sistemas distribuidos y aplicaciones web.

Tema 3 .2

Tecnologías de comunicación de aplicaciones

Nicolás Rodríguez
nicolas.rodriguez@urjc.es



Índice

- **Formatos de datos (JSON, XML)**
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- Servicios Web (SOAP y REST)
- Protocolos de mensajería (MQTT, AMQP)
- Middleware y colas de mensajes
- Apache Kafka
- RabbitMQ
- AWS SQS

Formatos de datos: JSON y XML

- Los formatos de datos son esenciales para el intercambio de información entre diferentes sistemas y aplicaciones.
- JSON (JavaScript Object Notation):
  - Formato ligero de intercambio de datos.
  - Fácil de leer y escribir para humanos, y fácil de analizar y generar para máquinas.
  - Basado en el subconjunto de JavaScript, pero independiente del lenguaje: se usa en muchos lenguajes de programación.
- XML (eXtensible Markup Language):
  - Formato que define un conjunto de reglas para codificar documentos de forma legible tanto para máquinas como para humanos.
  - Altamente personalizable y extensible.
  - A menudo utilizado en aplicaciones que requieren un alto grado de complejidad y estructura de datos, como en la industria de software empresarial.

Formatos de datos: JSON y XML

- Comparación y uso:
  - Verbosidad: XML es más detallado y extenso, mientras que JSON es más compacto y eficiente en términos de tamaño.
  - Facilidad de Uso: JSON es generalmente más fácil de usar y rápido en el procesamiento, ideal para aplicaciones web y móviles.
  - Estructura de Datos: XML es más adecuado para documentos con una estructura compleja y datos jerárquicos.
- Ejemplos de aplicaciones:
  - JSON en APIs web, configuraciones de aplicaciones y almacenamiento de datos simples.
  - XML en servicios web, documentos de Office y configuraciones de software más complejas.

Índice

- Formatos de datos (JSON, XML)
- **Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)**
- Servicios Web (SOAP y REST)
- Protocolos de mensajería (MQTT, AMQP)
- Middleware y colas de mensajes
- Apache Kafka
- RabbitMQ
- AWS SQS

Protocolos de red: TCP/IP, HTTP/HTTPS/SMTP

- Los protocolos de red son un conjunto de reglas que permiten la comunicación entre dispositivos en una red.
- Son fundamentales para el intercambio de datos en Internet y en redes locales.
- TCP/IP (Protocolo de Control de Transmisión/Protocolo de Internet):
  - Es un conjunto de protocolos de comunicación que interconectan dispositivos en Internet.
  - TCP se encarga de la entrega confiable de datos, mientras que IP se ocupa de la dirección y enrutamiento de paquetes.
  - Importancia: Esencial para la mayoría de las formas de comunicación en Internet, como la navegación web y el envío de correos electrónicos.

Protocolos de red: TCP/IP, HTTP/HTTPS/SMTP

- HTTP/HTTPS (Protocolo de Transferencia de Hipertexto/Protocolo Seguro):
  - HTTP es el protocolo utilizado para transferir datos en la World Wide Web.
  - HTTPS es la versión segura de HTTP, encriptando la comunicación para proteger los datos intercambiados.
  - Importancia: HTTP es fundamental para la navegación web, y HTTPS ha aumentado en importancia debido a la necesidad de seguridad y privacidad en línea.
- Funcionamiento y aplicaciones:
  - TCP/IP funciona dividiendo los mensajes en paquetes y ensamblándolos en el destino.
  - HTTP/HTTPS se utiliza en cada solicitud y respuesta en la web, desde cargar páginas hasta realizar transacciones en línea.

Protocolos de red: TCP/IP, HTTP/HTTPS/SMTP

- Simple Mail Transfer Protocol es un protocolo estándar para el envío de correos electrónicos en Internet:
  - Envío de correo electrónico: SMTP se utiliza principalmente para enviar mensajes de correo electrónico desde clientes de correo electrónico a servidores y entre servidores de correo.
  - Protocolo basado en texto: Utiliza comandos de texto simples para la comunicación, lo que facilita su implementación y depuración.
  - Proceso de entrega: En una transacción SMTP típica, un cliente se conecta al servidor SMTP del remitente, el servidor autentica al remitente (si es necesario), y luego el servidor transfiere el mensaje al servidor SMTP del destinatario.
  - Relay y routing: SMTP se encarga de dirigir y retransmitir los mensajes a través de una serie de servidores de correo hasta que el mensaje llega a su destino.

Índice

- Formatos de datos (JSON, XML)
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- **Servicios Web (SOAP y REST)**
- Protocolos de mensajería (MQTT, AMQP)
- Middleware y colas de mensajes
- Apache Kafka
- RabbitMQ
- AWS SQS

Servicios Web: SOAP y REST

- Los servicios web son interfaces que permiten la interacción entre aplicaciones a través de la red. Utilizan un conjunto de protocolos y estándares para asegurar que distintas máquinas puedan comunicarse entre sí.
- SOAP (Protocolo Simple de Acceso a Objetos):
  - Basado en XML para el intercambio de información.
  - Utiliza principalmente HTTP y SMTP para la comunicación.
  - Orientado a acciones con un enfoque en la funcionalidad y métodos.
- REST (Transferencia de Estado Representacional):
  - No depende de un protocolo específico, pero comúnmente usa HTTP.
  - Basado en la arquitectura de recursos y servicios web.
  - Utiliza métodos HTTP estándar como GET, POST, PUT y DELETE.

Servicios Web: SOAP y REST

- Comparación clave:
  - Flexibilidad: REST es generalmente más flexible y fácil de usar que SOAP.
  - Seguridad: SOAP tiene un estándar de seguridad más robusto (WS-Security).
  - Peso de los Datos: Las solicitudes REST son más ligeras en comparación con SOAP, mejorando la velocidad y eficiencia.
- Ejemplos de uso:
  - SOAP es comúnmente usado en sistemas bancarios y financieros por su seguridad y formalidad en las transacciones.
  - REST se utiliza ampliamente en aplicaciones web y móviles debido a su simplicidad y eficiencia.

Índice

- Formatos de datos (JSON, XML)
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- Servicios Web (SOAP y REST)
- **Protocolos de mensajería (MQTT, AMQP)**
- Middleware y colas de mensajes
- Apache Kafka
- RabbitMQ
- AWS SQS

Protocolos de Mensajería: MQTT y AMQP

- Son conjuntos de reglas que permiten el intercambio eficiente de mensajes entre sistemas y aplicaciones.
- Juegan un papel crucial en la Internet de las Cosas (IoT) y en la comunicación entre sistemas distribuidos.

Protocolos de Mensajería: MQTT y AMQP

- MQTT (Message Queuing Telemetry Transport):
  - Diseñado para conexiones con ancho de banda limitado y alta latencia.
  - Utiliza el modelo de publicación/suscripción, siendo ideal para dispositivos IoT.
  - Ofrece tres niveles de calidad de servicio (QoS), garantizando así la entrega de mensajes bajo diferentes condiciones de red.
- AMQP (Advanced Message Queuing Protocol):
  - Protocolo de mensajería orientado a mensajes, robusto y seguro.
  - Permite una amplia variedad de interacciones de mensajería, incluyendo enrutamiento de mensajes, transacciones y colas.
  - Adaptable para sistemas empresariales complejos y para garantizar la fiabilidad y seguridad en la entrega de mensajes.

Protocolos de Mensajería: MQTT y AMQP

- Diferencias clave:
  - Escalabilidad: MQTT es más adecuado para dispositivos y redes con recursos limitados, mientras que AMQP está diseñado para sistemas con necesidades más complejas.
  - Modelo de Mensajería: MQTT se centra en el modelo de publicación/suscripción, mientras que AMQP ofrece una gama más amplia de patrones de mensajería.
- Ejemplos de uso:
  - MQTT en dispositivos domésticos inteligentes, vehículos autónomos y apps de salud.
  - AMQP en sistemas financieros, logística y comunicaciones internas de empresas.

Índice

- Formatos de datos (JSON, XML)
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- Servicios Web (SOAP y REST)
- Protocolos de mensajería (MQTT, AMQP)
- **Middleware y colas de mensajes**
- Apache Kafka
- RabbitMQ
- AWS SQS

Middleware y colas de mensajes

- El middleware es un software que proporciona servicios comunes y capacidades de comunicación entre aplicaciones y componentes de software.
- Actúa como un puente entre diferentes aplicaciones y bases de datos.
- Funciones clave del Middleware:
  - Facilita la comunicación y el manejo de datos entre aplicaciones distribuidas.
  - Proporciona servicios como autenticación, autorización, y gestión de sesiones.
  - Ofrece funcionalidades como balanceo de carga y manejo de transacciones.

Middleware y colas de mensajes

- Colas de mensajes:
  - Son componentes de middleware que ayudan a gestionar y distribuir mensajes entre diferentes aplicaciones.
  - Permiten la comunicación asíncrona, donde el emisor y el receptor no necesitan estar en línea al mismo tiempo.
  - Ayudan a desacoplar procesos en SSDD, aumentando la escalabilidad y la resiliencia.
- Ejemplos de Middleware y colas de mensajes:
  - Middleware: Oracle Middleware, IBM WebSphere.
  - Colas de Mensajes: Apache Kafka, RabbitMQ, AWS SQS.
- Ventajas del uso de Middleware y colas de mensajes:
  - Mejora la interoperabilidad entre diferentes sistemas y tecnologías.
  - Facilita la escalabilidad y la gestión de grandes volúmenes de datos y transacciones.
  - Aumenta la fiabilidad y la disponibilidad de los sistemas distribuidos.

Colas de mensajes

- Las colas de mensajes son componentes fundamentales en la arquitectura de sistemas distribuidos.
- Permiten la comunicación asíncrona entre diferentes partes de un sistema, donde los productores envían mensajes y los consumidores los reciben y procesan.
- Cómo funcionan las colas de mensajes:
  - Los mensajes se almacenan en una cola hasta que pueden ser procesados.
  - Aseguran que los mensajes se entreguen y procesen en el orden en que se recibieron.
  - Proporcionan un mecanismo para manejar picos de carga, distribuyendo los mensajes a medida que los sistemas están disponibles para procesarlos.

Colas de mensajes

- Características importantes:
  - Fiabilidad: Garantizan que los mensajes no se pierdan, incluso en caso de fallos en el sistema.
  - Escalabilidad: Facilitan el manejo de grandes volúmenes de mensajes y la expansión de sistemas.
  - Desacoplamiento: Los productores y consumidores operan independientemente, mejorando la modularidad y mantenibilidad.
- Ejemplos y herramientas Populares:
  - Apache Kafka: Orientado a alto rendimiento y distribución.
  - RabbitMQ: Ampliamente usado, enfocado en la simplicidad y facilidad de uso.
  - AWS SQS (Simple Queue Service): Solución en la nube, integrable con otros servicios de AWS.

Índice

- Formatos de datos (JSON, XML)
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- Servicios Web (SOAP y REST)
- Protocolos de mensajería (MQTT, AMQP)
- Middleware y colas de mensajes
- **Apache Kafka**
- RabbitMQ
- AWS SQS

Apache Kafka

- Kafka es una plataforma de código abierto diseñada para procesar y manejar streams de datos en tiempo real.
- Fue desarrollado inicialmente por LinkedIn y luego donado a la Apache Software Foundation.
- Características principales:
  - Alto Rendimiento y Escalabilidad: Puede manejar miles de mensajes por segundo, escalando horizontalmente para soportar grandes volúmenes de datos.
  - Modelo de Publicación/Suscripción: Los productores envían mensajes a topics (temas), y los consumidores los leen, facilitando una comunicación efectiva y eficiente.
  - Durabilidad y Fiabilidad: Almacena los mensajes en discos, lo que garantiza que no se pierdan incluso en caso de fallos del sistema.

kafka

Apache Kafka

- Arquitectura de Kafka:
  - Brokers: Servidores que almacenan los datos y sirven a los consumidores.
  - ZooKeeper: Utilizado para la coordinación y gestión de los brokers de Kafka.
  - Producers y Consumers: Aplicaciones que publican y leen mensajes desde los topics.
- Casos de uso comunes:
  - Procesamiento de streams en tiempo real: Análisis de datos en vivo, como monitoreo de transacciones financieras.
  - Sistemas de recomendación: Procesamiento de actividades de usuarios para generar recomendaciones personalizadas.
  - Integración de datos y microservicios: Como una capa de comunicación entre diferentes microservicios en una arquitectura empresarial.

Índice

- Formatos de datos (JSON, XML)
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- Servicios Web (SOAP y REST)
- Protocolos de mensajería (MQTT, AMQP)
- Middleware y colas de mensajes
- Apache Kafka
- **RabbitMQ**
- AWS SQS

RabbitMQ

- RabbitMQ es un sistema de mensajería de código abierto que actúa como un intermediario de mensajes (message broker) para aplicaciones.
- Es ampliamente utilizado para la comunicación entre diferentes componentes de un sistema, permitiendo la transmisión de mensajes de manera confiable y eficiente.
- RabbitMQ implementa AMQPaunque también soporta otros protocolos como MQTT, STOMP y HTTP.

RabbitMQ – Conceptos clave

- Colas de mensajes: RabbitMQ permite a las aplicaciones enviar y recibir mensajes a través de colas, que son almacenamientos de mensajes hasta que son procesados.
- Publicación y suscripción: Los productores publican mensajes en colas y los consumidores los suscriben y procesan.
- Modelo de intercambio (Exchange): Controla cómo se enrutan los mensajes entre las colas. Existen varios tipos de intercambio, como directo, tópico, encabezados y fanout.

RabbitMQ – Conceptos clave

- Confiabilidad: Garantiza la entrega de mensajes a través de funcionalidades como la confirmación de mensajes y la persistencia.
- Escalabilidad y alto rendimiento: Puede manejar un gran volumen de mensajes y es altamente escalable.
- Clustering y tolerancia a fallos: Soporta clustering para mayor disponibilidad y resistencia a fallos.
- Soporte para múltiples lenguajes y plataformas: Puede ser utilizado con una variedad de lenguajes de programación a través de bibliotecas cliente.

RabbitMQ – Aplicaciones

- Desacoplamiento de aplicaciones: Permite que diferentes componentes de una aplicación se comuniquen de manera asincrónica, aumentando la modularidad.
- SSDD: Facilita la comunicación en sistemas distribuidos, mejorando la escalabilidad y la eficiencia.
- Manejo de cargas de trabajo: Utilizado para distribuir tareas entre varios trabajadores (workers) y balancear la carga.
- Integraciones de sistemas: Facilita la integración de sistemas y aplicaciones diferentes.

Índice

- Formatos de datos (JSON, XML)
- Protocolos de red (TCP/IP, HTTP/HTTPS, SMTP)
- Servicios Web (SOAP y REST)
- Protocolos de mensajería (MQTT, AMQP)
- Middleware y colas de mensajes
- Apache Kafka
- RabbitMQ
- **AWS SQS**

AWS SQS

- AWS SQS (Amazon Web Services Simple Queue Service) es un servicio de colas de mensajes completamente gestionado que permite la desvinculación y el escalado de microservicios, sistemas distribuidos y aplicaciones sin servidor.
- Ofrece una solución robusta y escalable para el manejo de mensajes entre componentes de software en la nube de AWS.

amazon SQS

AWS SQS – Conceptos clave

- SQS almacena mensajes en múltiples servidores para asegurar su persistencia, ofreciendo durabilidad a largo plazo (hasta 14 días).
- Con SQS, los usuarios pueden enviar, almacenar y recibir un número ilimitado de mensajes entre componentes de software en cualquier momento, sin pérdida de mensajes.
- Dos tipos de colas:
  - Colas Estándar: Ofrecen throughput máximo, entrega al menos una vez y un orden de mensajes que puede no ser exacto.
  - Colas FIFO (First-In-First-Out): Aseguran que los mensajes se entreguen y procesen exactamente una vez y en el orden exacto en que se envían.

AWS SQS – Conceptos clave

- Escalabilidad: SQS escala automáticamente con la aplicación, por lo que no es necesario administrar la infraestructura de mensajería.
- Seguridad: Los usuarios pueden utilizar IAM (Identity and Access Management) para controlar el acceso a las colas SQS y crifrar mensajes para asegurar datos sensibles.
- Integración: SQS se integra con otros servicios de AWS, así como con sistemas de notificaciones y BBDD, lo que facilita la orquestación de flujos de trabajo y el procesamiento de datos.

AWS SQS – Aplicaciones

- Desacoplamiento de aplicaciones: Permite que las diferentes partes de una aplicación se comuniquen sin estar conectadas directamente, mejorando la fiabilidad y la flexibilidad.
- Manejo de cargas de trabajo: SQS puede ser utilizado para suavizar picos de tráfico almacenando mensajes hasta que los sistemas estén listos para procesarlos.
- Orquestación de flujos de trabajo: Al integrarse con otros servicios de AWS, SQS se puede utilizar para orquestar flujos de trabajo complejos y procesamiento de transacciones.

Sistemas Distribuidos

Bloque II

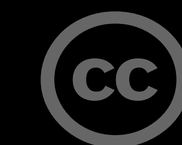
Desarrollo de sistemas distribuidos y aplicaciones web.

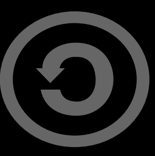
Tema 3 .2

Tecnologías de comunicación de aplicaciones

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

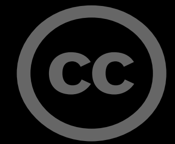
Bloque III

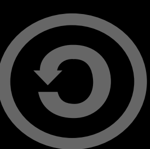
Orientación a servicios y soluciones de SSDD orientadas a servicios

Tema 4

Virtualización y computación en la nube

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Índice

- **Introducción a la virtualización**
- Beneficios de la virtualización en SSDD
- Tipos de virtualización aplicados a SSDD
- Herramientas y tecnologías relevantes
- Contenedores (Docker, Kubernetes)
- Diferencias en la virtualización
- Ejemplo

Introducción a la virtualización

- La virtualización es el proceso de crear una representación virtual o simulada de algo, como un sistema operativo, un servidor, un dispositivo de almacenamiento o recursos de red.
- En el contexto de la tecnología de la información, se refiere a la creación de una versión virtual de un recurso informático, como un hardware físico.

Introducción a la virtualización

- La virtualización desacopla los recursos de hardware de las funciones de software, permitiendo que varios sistemas operativos y aplicaciones se ejecuten simultáneamente en un solo sistema físico.
- Utiliza un software, conocido como hypervisor, para emular el hardware y crear entornos virtuales (máquinas virtuales) que pueden gestionar y ejecutar procesos independientemente.

Introducción a la virtualización

- Virtualización de servidores: Permite múltiples sistemas operativos en un solo servidor físico.
- Virtualización de redes: Simula recursos de red para proporcionar conectividad virtual.
- Virtualización de almacenamiento: Agrupa el almacenamiento físico de múltiples dispositivos de red.

Introducción a la virtualización

- Eficiencia mejorada: Optimización del uso de recursos físicos, lo que reduce costes y mejora la eficiencia energética.
- Flexibilidad y escalabilidad: Facilita la gestión y el despliegue de recursos, permitiendo un escalado rápido y flexible.
- Aislamiento y seguridad: Cada entorno virtual es aislado de los demás, mejorando la seguridad y la gestión de fallos.

Introducción a la virtualización

- Computación en la nube: Base para ofrecer servicios en la nube como IaaS, PaaS y SaaS.
- Centros de datos: Permite una gestión más eficiente y una reducción de los costos operativos en centros de datos.
- Desarrollo y pruebas: Proporciona entornos de prueba y desarrollo flexibles y aislados.

Introducción a la virtualización

- Orígenes en la Década de 1960:
  - La virtualización comenzó con el desarrollo de la tecnología de tiempo compartido en mainframes.
  - Ejemplos iniciales: IBM y su sistema CP-40 y el proyecto CP-67 que llevaron al desarrollo de la serie IBM System/370.
- Avances en los años 70 y 80:
  - Expansión de la virtualización en mainframes.
  - Desarrollo de tecnologías de VM más sofisticadas.
- El renacimiento en los Años 90:
  - Aparición de plataformas x86, marcando el comienzo de la virtualización para sistemas más pequeños y menos costosos.
  - VMware lanza las primeras soluciones comerciales para virtualización de estaciones de trabajo y servidores.

Introducción a la virtualización

- Auge en los Años 2000:
  - Virtualización se convierte en una tecnología clave para centros de datos y servicios en la nube.
  - Innovaciones en la virtualización de servidores, redes y almacenamiento.
- Era actual y la nube:
  - Virtualización como pilar fundamental de la infraestructura de nube.
  - Desarrollo de tecnologías de contenedores (como Docker) y orquestación (como Kubernetes).

Introducción a la virtualización

- En la infraestructura de TI:
  - Piedra angular en la modernización de la infraestructura de TI.
  - Importancia en la consolidación de servidores, optimización de recursos y reducción de costos operativos.
- Computación en la Nube:
  - Virtualización como base para los servicios en la nube (IaaS, PaaS, SaaS).
  - Ejemplos de proveedores de nube que dependen de la virtualización, como AWS, Azure, y Google Cloud.
- Impacto en la flexibilidad y agilidad empresarial:
  - La virtualización permite a las empresas desplegar y gestionar aplicaciones y servicios rápidamente.
  - Facilita la escalabilidad y adaptabilidad ante cambios en la demanda y necesidades del negocio.

Introducción a la virtualización

- Continuidad del negocio y recuperación ante desastres:
  - Simplificación de la copia de seguridad y recuperación de datos.
  - Mejora de la resistencia empresarial a través de la replicación virtual y las estrategias de recuperación ante desastres.
- Avance en la innovación y el desarrollo de software:
  - Importancia en el desarrollo, prueba y despliegue de software (contenedores, entornos virtuales).
  - Facilita la implementación de DevOps y metodologías ágiles.

Índice

- Introducción a la virtualización
- **Beneficios de la virtualización en SSDD**
- Tipos de virtualización aplicados a SSDD
- Herramientas y tecnologías relevantes
- Contenedores (Docker, Kubernetes)
- Diferencias en la virtualización
- Ejemplo

Beneficios de la virtualización en SSDD

- Mejora de la eficiencia y utilización de recursos:
  - La virtualización permite una mejor utilización del hw, reduciendo la cantidad de recursos físicos necesarios.
  - Aumenta la eficiencia energética al consolidar servidores y reducir el consumo de energía en centros de datos.
- Flexibilidad y escalabilidad:
  - Facilita la escalabilidad rápida de recursos y servicios sin la necesidad de hw adicional.
  - Permite una rápida reconfiguración de recursos para adaptarse a las cambiantes demandas del negocio.
- Aislamiento y seguridad:
  - Cada máquina virtual opera de manera aislada, lo que reduce el riesgo de fallas y mejora la seguridad.
  - Facilita la implementación de políticas de seguridad y recuperación ante desastres.

Beneficios de la virtualización en SSDD

- Reducción de costes:
  - Menor necesidad de inversión en hardware físico y mantenimiento asociado.
  - Reduce los costes operativos relacionados con la energía y la gestión de centros de datos.
- Agilidad en el desarrollo y pruebas:
  - Permite a los desarrolladores y probadores crear y desplegar rápidamente entornos replicables y consistentes.
  - Facilita el desarrollo y las pruebas de aplicaciones en múltiples entornos sin requerir múltiples configuraciones de hardware.
- Continuidad del negocio y recuperación ante desastres:
  - La virtualización permite estrategias más efectivas y eficientes para la continuidad del negocio y la recuperación ante desastres.
  - Facilita la replicación rápida de entornos virtuales y la recuperación de datos.

Beneficios de la virtualización en SSDD

- IaaS proporciona recursos de computación virtualizados a través de Internet.
- Se trata de una infraestructura de TI completa, incluyendo servidores virtuales, almacenamiento, redes y sistemas operativos.
- Características clave:
  - Flexibilidad y control sobre la infraestructura de TI.
  - Escalabilidad para ajustar recursos según la demanda.
  - Modelo de pago por uso: los usuarios pagan solo por los recursos que utilizan.
- Ejemplos: MS Azure, AWS, Google Cloud Plataform

Beneficios de la virtualización en SSDD

- PaaS proporciona un entorno de desarrollo y despliegue en la nube, incluyendo herramientas para el desarrollo de aplicaciones, bases de datos, sistemas de gestión de contenido y más.
- Características clave:
  - Entorno integrado de desarrollo, prueba, despliegue y gestión.
  - Facilita el desarrollo rápido de aplicaciones.
  - Gestión simplificada de la infraestructura subyacente.
- Ejemplos: Heroku, Microsoft Azure App Services, Google App Engine

Beneficios de la virtualización en SSDD

- SaaS ofrece software y aplicaciones a través de Internet, accesibles desde navegadores web o aplicaciones ligeras.
- Características clave:
  - Accesibilidad universal a través de Internet.
  - No requiere instalación, mantenimiento ni actualización de software por parte del usuario.
  - Modelo de suscripción generalmente basado en usuarios o uso.
- Ejemplos: Gmail, Microsoft Office 365 y Slack.
- Ventajas:
  - Facilidad de uso y acceso desde cualquier lugar.
  - Reducción de costes de TI y complejidad operativa para los usuarios.

Índice

- Introducción a la virtualización
- Beneficios de la virtualización en SSDD
- **Tipos de virtualización aplicados a SSDD**
- Herramientas y tecnologías relevantes
- Contenedores (Docker, Kubernetes)
- Diferencias en la virtualización
- Ejemplo

Tipos de virtualización

- Virtualización de hw:
  - Crea una representación virtual de los recursos de hardware, como CPU, RAM y almacenamiento.
  - Permite que múltiples sistemas operativos y aplicaciones se ejecuten en una sola pieza de hardware físico, mejorando la utilización y la eficiencia.
  - Ejemplos: Hypervisores como VMware ESXi, Microsoft Hyper-V, y Citrix XenServer.
- Virtualización de sw (Sistemas Operativos):
  - Permite la ejecución de múltiples instancias de SSOO en un solo sistema físico.
  - Facilita el aislamiento y la independencia del sistema operativo, permitiendo diferentes entornos operativos en un mismo hardware.
  - Ejemplos: Parallels Desktop, Oracle VirtualBox.

Tipos de virtualización

- Virtualización a nivel de red:
  - Simula hardware de red, como switches, routers y firewalls.
  - Ofrece flexibilidad para configurar redes virtuales, mejorando la seguridad y la eficiencia de la gestión de tráfico de red.
  - Ejemplos: VMware NSX, Cisco Nexus 1000V.
- Virtualización de almacenamiento:
  - Agrupa el almacenamiento físico de múltiples dispositivos de red en un único dispositivo de almacenamiento virtual.
  - Mejora la eficiencia y accesibilidad del almacenamiento de datos, facilitando la gestión y escalabilidad.
  - Ejemplos: SAN (Red de Área de Almacenamiento) y sistemas NAS (Network-Attached Storage) virtualizados.

Tipos de virtualización

- Virtualización de aplicaciones:
  - Descripción: Desacopla las aplicaciones de los sistemas operativos subyacentes, permitiendo su ejecución en entornos virtualizados.
  - Aplicación en SSDD: Permite una entrega y gestión más flexibles de las aplicaciones, facilitando la movilidad y compatibilidad.
  - Ejemplos: Citrix XenApp, Microsoft App-V.

Índice

- Introducción a la virtualización
- Beneficios de la virtualización en SSDD
- Tipos de virtualización aplicados a SSDD
- **Herramientas y tecnologías relevantes**
- Contenedores (Docker, Kubernetes)
- Diferencias en la virtualización
- Ejemplo

Herramientas y tecnologías relevantes

- VMware:
  - Líder en soluciones de virtualización y infraestructura de nube.
  - VMware vSphere para virtualización de servidores, VMware NSX para virtualización de redes, VMware vSAN para almacenamiento definido por software.
- Microsoft Hyper-V:
  - La solución de virtualización integrada en Windows Server.
  - Ofrece gestión de máquinas virtuales, redes virtuales y almacenamiento virtual.
- Citrix XenServer:
  - Una plataforma de virtualización de código abierto centrada en la eficiencia y la escalabilidad.
  - Ampliamente utilizado para virtualización de servidores y escritorios.

Herramientas y tecnologías relevantes

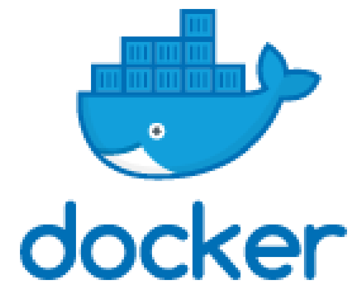
- Oracle VirtualBox:
  - Software de virtualización de código abierto para ejecutar múltiples SSOO.
  - Ideal para pruebas y entornos de desarrollo.
- Contenedores (Docker, Kubernetes):
  - Docker: Permite empaquetar aplicaciones en contenedores, facilitando la portabilidad y la consistencia.
  - Kubernetes: Sistema de orquestación para la gestión automatizada, escalado y despliegue de contenedores.
- Herramientas de automatización y gestión:
  - Ansible, Puppet, Chef: Utilizadas para automatizar la configuración y el despliegue de infraestructuras virtualizadas.
  - Estas herramientas facilitan la gestión eficiente y reducen el margen de error humano.

Índice

- Introducción a la virtualización
- Beneficios de la virtualización en SSDD
- Tipos de virtualización aplicados a SSDD
- Herramientas y tecnologías relevantes
- **Contenedores (Docker, Kubernetes)**
- Diferencias en la virtualización
- Ejemplo

Contenedores – Docker

- Docker es una plataforma de contenedores que permite a los desarrolladores empaquetar aplicaciones y sus dependencias en un contenedor virtual.
- Este contenedor puede ejecutarse en cualquier entorno que tenga Docker instalado, asegurando la consistencia en diferentes entornos de desarrollo, pruebas y producción.



Contenedores – Docker

- Imágenes de Docker:
  - Plantillas de solo lectura utilizadas para crear contenedores.
  - Contienen la aplicación y todos sus entornos y dependencias.
- Contenedores Docker:
  - Instancias ejecutables de imágenes de Docker.
  - Proporcionan un entorno aislado donde se ejecuta la aplicación.
- Docker Hub y registros:
  - Repositorios para compartir y administrar imágenes de Docker.
  - Docker Hub es el registro público de Docker, donde los usuarios pueden subir y descargar imágenes.

Contenedores – Docker

- Virtualización a nivel de sistema operativo: A diferencia de las máquinas virtuales tradicionales, los contenedores Docker comparten el kernel del sistema operativo anfitrión, pero pueden tener sus propios espacios de usuario. **Esto los hace más ligeros y rápidos.**
- Despliegue y escalabilidad: Docker facilita el despliegue rápido y la escalabilidad de aplicaciones al permitir que los contenedores se inicien y detengan rápidamente.

Contenedores – Docker

- Consistencia en entornos diversos: Asegura que las aplicaciones funcionen de manera uniforme en diferentes entornos.
- Eficiencia de recursos: Menor sobrecarga en comparación con las máquinas virtuales tradicionales.
- Desarrollo y despliegue rápidos: Facilita la integración y entrega continuas (CI/CD) de aplicaciones.
- Aislamiento y seguridad: Cada contenedor opera de forma aislada, lo que mejora la seguridad.

Contenedores – Kubernetes

- Kubernetes, también conocido como K8s, es un sistema de orquestación de contenedores de código abierto.
- Fue originalmente desarrollado por Google y ahora es mantenido por la Cloud Native Computing Foundation.
- Diseñado para automatizar la implementación, el escalado y la gestión de aplicaciones en contenedores.

kubernetes

Contenedores – Kubernetes

- Orquestación de contenedores: Gestiona la vida útil de los contenedores en un entorno de sistema distribuido.
- Autoscaling: Ajusta automáticamente la cantidad de contenedores basándose en el uso de los recursos.
- Balanceo de carga y descubrimiento de servicios: Distribuye el tráfico de red entre los contenedores y los hace descubribles dentro del clúster.

Contenedores – Kubernetes

- Gestión automatizada: Facilita la gestión de aplicaciones a gran escala y reduce la necesidad de intervención manual.
- Escalabilidad: Permite que las aplicaciones se escalen hacia arriba o hacia abajo según la demanda.
- Resiliencia: Mejora la disponibilidad de las aplicaciones al gestionar el despliegue y la recuperación de fallos de los contenedores.

Contenedores – Kubernetes

- Gestión automatizada: Facilita la gestión de aplicaciones a gran escala y reduce la necesidad de intervención manual.
- Escalabilidad: Permite que las aplicaciones se escalen hacia arriba o hacia abajo según la demanda.
- Resiliencia: Mejora la disponibilidad de las aplicaciones al gestionar el despliegue y la recuperación de fallos de los contenedores.

Índice

- Introducción a la virtualización
- Beneficios de la virtualización en SSDD
- Tipos de virtualización aplicados a SSDD
- Herramientas y tecnologías relevantes
- Contenedores (Docker, Kubernetes)
- **Diferencias en la virtualización**
- Ejemplo

Diferencias en la virtualización – Arquitectura

- Máquinas virtuales:
  - Virtualizan el hardware físico para ejecutar un sistema operativo completo con su propio kernel.
  - Cada VM incluye una copia del sistema operativo, las aplicaciones, los binarios necesarios y las bibliotecas, todo corriendo sobre un hipervisor que gestiona la VM y el hardware subyacente.
- Contenedores:
  - Los contenedores virtualizan el sistema operativo en lugar del hardware.
  - Corren en un solo kernel del sistema operativo (el del host) y comparten este kernel con otros contenedores.
  - Son esencialmente aislados entre sí, pero utilizan las mismas binarias y bibliotecas del sistema operativo host cuando es posible, lo que les permite ser más ligeros en términos de recursos.

Diferencias en la virtualización – Uso de recursos

- Máquinas virtuales:
  - Debido a que cada VM tiene su propio sistema operativo completo, tienden a utilizar más recursos del sistema, como CPU y memoria.
  - El arranque de una VM también puede ser más lento, ya que implica iniciar un sistema operativo completo.
- Contenedores:
  - Los contenedores son más eficientes en cuanto a recursos.
  - Dado que comparten el kernel del sistema operativo con el host y otros contenedores, y solo necesitan cargar las aplicaciones y sus dependencias inmediatas, utilizan menos memoria y espacio en disco.
  - Además, se inician casi instantáneamente.

Diferencias en la virtualización – Portabilidad

- Máquinas virtuales:
  - La portabilidad de una VM puede ser algo limitada debido a su tamaño y a que están ligadas a la configuración específica del hardware virtualizado por el hipervisor.
- Contenedores:
  - Los contenedores son altamente portátiles.
  - Pueden ser fácilmente trasladados y ejecutados en cualquier sistema que tenga el mismo sistema operativo y soporte para la tecnología de contenedores (como Docker), independientemente de las configuraciones de hardware subyacentes.

Diferencias en la virtualización – Aislamiento

- Máquinas virtuales:
  - Ofrecen un alto nivel de aislamiento y seguridad, ya que el hipervisor ofrece una barrera robusta que separa cada VM del hardware subyacente y de otras VMs.
  - Esto hace que las VMs sean ideales para ejecutar aplicaciones que requieren un alto nivel de seguridad o aislamiento.
- Contenedores:
  - Mientras que los contenedores son aislados, comparten el mismo kernel del sistema operativo, lo que puede representar un riesgo mayor si un contenedor se ve comprometido.
  - Sin embargo, tecnologías modernas como Docker han continuado mejorando en seguridad y técnicas de aislamiento.

Diferencias en la virtualización – Escalabilidad

- Máquinas virtuales:
  - Pueden ser más laboriosas para mantener y escalar debido a la necesidad de administrar cada sistema operativo por separado.
- Contenedores:
  - Son más fáciles de escalar y mantener
  - Se pueden orquestar múltiples contenedores a través de plataformas como Kubernetes, lo cual es ideal para aplicaciones microservicios y entornos de alta disponibilidad.

Índice

- Introducción a la virtualización
- Beneficios de la virtualización en SSDD
- Tipos de virtualización aplicados a SSDD
- Herramientas y tecnologías relevantes
- Contenedores (Docker, Kubernetes)
- Diferencias en la virtualización
- **Ejemplo**

Ejemplo

- Generamos un proyecto con Spring Initializr con Spring Web, Spring Data, MySQL Drive y Lombok.
- Añadimos las clases Cerveza, CervezaRESTController, CervezaService y CervezaRepository.
- Añadimos las siguientes líneas al application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/testDB
spring.datasource.username=root
spring.datasource.password=pass
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

Ejemplo

- Para instalar Docker en Windows:
- https://docs.docker.com/desktop/install/windows-install/
- Reiniciar:

Complete the installation of Docker Desktop.

- (●) Use recommended settings (requires administrator password)

  Docker Desktop automatically sets the necessary configurations that work for most developers.

- ( ) Use advanced settings

  You manually set your preferred configurations.

- En CMD o PowerShell: docker --version

```
PS C:\Users\nicor> docker --version
Docker version 25.0.3, build 4debf41
PS C:\Users\nicor>
```

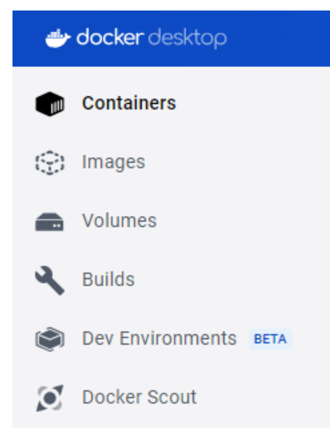
Ejemplo

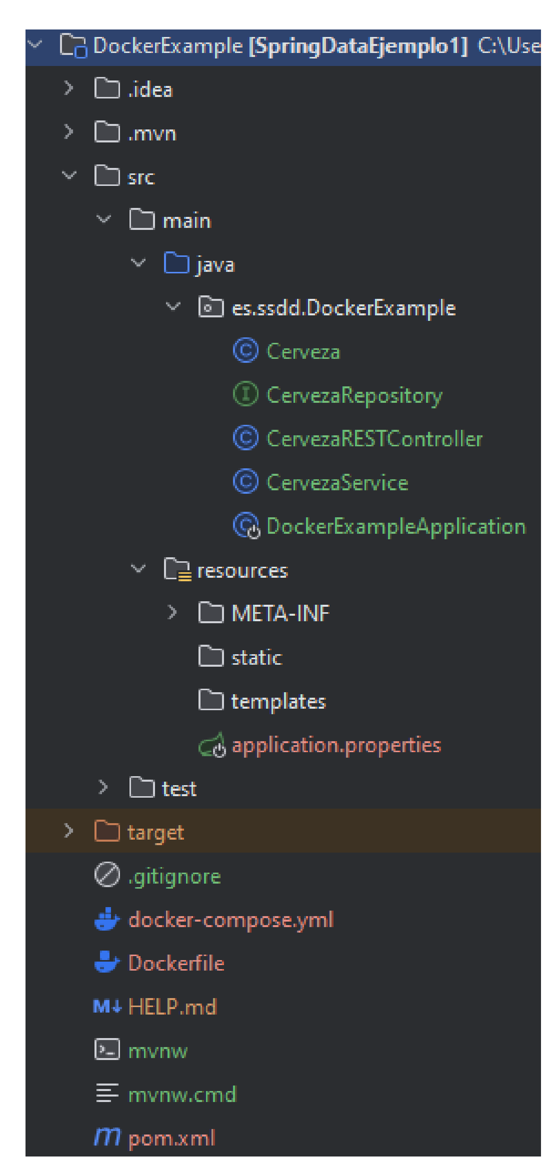
- Estructura del proyecto:

(Tanto Dockerfile como docker-compose están al mismo nivel)

Revisar: https://hub.docker.com/

Es necesario iniciar Docker:





Ejemplo

- Dockerfile
  - Sin extensión.
  - (1/2)
  - La primera línea permite indicar cómo compilar nuestro proyecto.
  - El resto de las líneas hacen referencias al contenedor.
  - En la segunda se establece el directorio de trabajo.
  - En la tercera se copia de las dependencias del proyecto.
  - En la cuarta línea se descargan las dependencias.
  - En la quinta línea se “mueve” el código.
  - En la sexta, se compila nuestro proyecto a través de Maven.

```
#################################################
# Imagen base para el contenedor de compilación
#################################################
FROM maven:3.8.5-openjdk-17 as builder

# Define el directorio de trabajo donde ejecutar comandos
WORKDIR /project

# Copia las dependencias del proyecto
COPY pom.xml /project/

# Descarga las dependencias del proyecto
RUN mvn clean verify

# Copia el código del proyecto
COPY /src /project/src

# Compila proyecto
RUN mvn package -o -DskipTests=true
```

Ejemplo

- Dockerfile
  - Continuación (2/2)
  - La primera línea índica el jdk que se usará.
  - En la segunda se establece el directorio de trabajo.
  - En la tercera se copia el .jar a un directorio.
  - En la cuarta línea se expone el contenedor en el 8080.
  - En la quinta línea se ejecuta el .jar resultante de la compilación.
    Nota. El nombre del .jar tiene que coincidir con el del pom.xml

```
####################################################
# Imagen base para el contenedor de la aplicación
####################################################
FROM openjdk:17

# Define el directorio de trabajo donde se encuentra el JAR
WORKDIR /usr/app/

# Copia el JAR del contenedor de compilación
COPY --from=builder /project/target/*.jar /usr/app/

# Indica el puerto que expone el contenedor
EXPOSE 8080

# Comando que se ejecuta al hacer docker run
CMD [ "java", "-jar", "app.jar" ]
```

Ejemplo

- docker-compose.yml
  - Define dos contenedores, ‘app’ y ‘db’.
  - “build: .” especifica que la imagen del Dockerfile se construye en el directorio actual.
  - “ports: -8080:8080” enlazamiento de puertos
  - “environment […]” establece variables de entorno en el contenedor. Mismas que en el application.properties.
  - “depends_on: - db” indica que el servicio “db” se ha de iniciar antes que este. NO NECESARIAMENTE TIENE PORQUE ESTAR ARRANCADO ANTES”
  - “restart: on-failure” indica a Docker reiniciar el servicio si falla.

```yaml
services:
  app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://db/testDB
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=password
    depends_on:
      - db
    restart: on-failure

  db:
    image: mysql:8.0
    environment:
      MYSQL_DATABASE: testDB
      MYSQL_ROOT_PASSWORD: password
    volumes:
      - db-data:/var/lib/mysql
    restart: always

volumes:
  db-data:
```

Ejemplo

- docker-compose.yml
  - “image: mysql:8.0” usa la imagen oficial de MySQL que se encuentra el Docker Hub.
    “environment” variables de entorno para BBDD.
  - “volumes” sección de base de datos persistente correspondiente a la base de datos. Se mantiene independiente del contenedor en sí.
  - “restart: always” se reinicia el contenedor mientras que no esté funcionando.
  - volumes. Define el nombre del volumen de la BBDD.

```yaml
services:
  app:
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://db/testDB
      - SPRING_DATASOURCE_USERNAME=root
      - SPRING_DATASOURCE_PASSWORD=password
    depends_on:
      - db
    restart: on-failure

  db:
    image: mysql:8.0
    environment:
      MYSQL_DATABASE: testDB
      MYSQL_ROOT_PASSWORD: password
    volumes:
      - db-data:/var/lib/mysql
    restart: always

volumes:
  db-data:
```

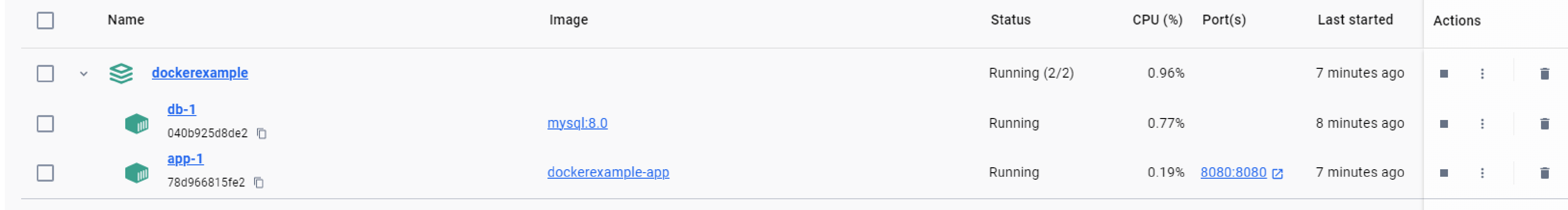
Ejemplo

- pom.xml
  - Es necesario revisar los conectores.
  - Hay que quitar los referentes a H2 y añadir el de mysql. Se recomienda este de mysql.
  - Es necesario definir el nombre final del .jar generado. Para que se corresponda con el Dockerfile se le llama “app”.
  - Es necesario marcar la clase principal a ejecutar, para ello se se utiliza la etiqueta <mainClass>
  - Si se usa lombok, se excluye.

```xml
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.19</version>
  </dependency>
</dependencies>
<build>
  <finalName>app</finalName> <!-- Update this line -->
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <mainClass>es.ssdd.DockerExample.DockerExampleApplication</mainClass>  <!-- Update this line -->
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Ejemplo

- Lanzamos en PowerShell: docker-compose up –build
- Si revisamos “Containers” en Docker-desktop:



Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
dockerexample		Running (2/2)	0.96%		7 minutes ago	
db-1 040b925d8de2	mysql:8.0	Running	0.77%		8 minutes ago	
app-1 78d966815fe2	dockerexample-app	Running	0.19%	8080:8080	7 minutes ago	

- Si revisamos Images en Docker-desktop:

Name	Tag	Status	Created	Size	Actions
dockerexample-app 44772b7f5a01	latest	In use	7 minutes ago	519.58 MB	
mysql f5f171121fa3	8.0	In use	1 month ago	602.94 MB	

Sistemas Distribuidos

Bloque III

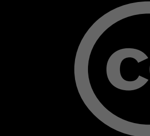
Orientación a servicios y soluciones de SSDD orientadas a servicios

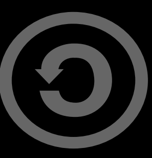
Tema 4

Virtualización y computación en la nube

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Sistemas Distribuidos

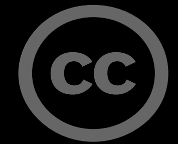
Bloque IV

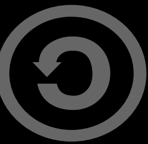
Sistemas de persistencia distribuidos

Tema 5

Bases de Datos Distribuidas

Nicolás Rodríguez
nicolas.rodriguez@urjc.es





Índice

- **Sistemas de persistencia distribuidos**
- Arquitecturas
- Centralización vs. Distribución
- Tecnologías y herramientas

Sistemas de persistencia distribuidos

- Es un tipo de sistema de almacenamiento de datos que distribuye y gestiona la información a través de múltiples ubicaciones físicas o lógicas.
- Los datos no se almacenan en una única ubicación central, sino repartidos en varios nodos.
- Principios fundamentales:
  - Distribución de datos: Los datos se almacenan en múltiples nodos, lo que aumenta la redundancia y la resistencia a fallos.
  - Independencia de ubicación: Los datos pueden ser accesibles desde cualquier nodo, mejorando el acceso y la disponibilidad.
  - Escalabilidad: La capacidad de agregar más nodos para incrementar el almacenamiento y el rendimiento sin interrumpir el servicio existente.

Sistemas de persistencia distribuidos

- Nodos de almacenamiento:
  - Cada nodo puede ser un servidor, un PC o un dispositivo de almacenamiento dedicado.
  - Cada nodo puede operar de manera independiente, pero también se coordina con otros nodos.
- Red de comunicaciones:
  - Red que conecta los nodos, permitiendo el intercambio de datos y mensajes de control.
  - La eficiencia y la seguridad de la red impactan en el rendimiento del sistema.
- Software de gestión:
  - Software que administra la distribución de datos, la consistencia, y la recuperación en caso de fallos.
  - Ejemplos de operaciones gestionadas: replicación de datos, balanceo de carga, y manejo de errores.

Sistemas de persistencia distribuidos

- Datos estructurados:
  - Bases de datos relacionales, donde los datos se organizan en tablas y columnas.
- Datos no estructurados:
  - Archivos multimedia, documentos, y registros de logs que no siguen un modelo estructurado.
- Datos semiestructurados:
  - Datos que no están organizados en un formato rígido como el relacional, pero que contienen etiquetas o marcas que permiten identificar elementos de los datos, como XML y JSON.

Sistemas de persistencia distribuidos

- Estos sistemas subyacen en servicios comunes como:
  - Almacenamiento en la nube.
  - Plataformas de streaming.
  - Redes sociales.
- Permiten gestionar y acceder a enormes cantidades de datos de manera eficiente.
- Importante en áreas como:
  - Salud (registros médicos electrónicos).
  - Finanzas (transacciones y análisis de mercado en tiempo real).
  - Comercio electrónico (gestión de inventarios y procesamiento de pedidos).

Sistemas de persistencia distribuidos

- Escalabilidad y flexibilidad:
  - Los SSDD pueden manejar incrementos en la demanda de datos y tráfico de usuarios más eficientemente que los sistemas centralizados, adaptándose a las necesidades cambiantes sin interrupciones significativas.
- Resiliencia y disponibilidad:
  - Estos sistemas mantienen la operatividad incluso ante fallos de hardware o software en ciertos nodos, asegurando así una mayor continuidad del servicio.
- Eficiencia en el procesamiento y acceso a datos:
  - La distribución de datos y carga de trabajo facilita un acceso más rápido y eficiente a los datos, crucial para aplicaciones que requieren tiempos de respuesta bajos.

Índice

- Sistemas de persistencia distribuidos
- **Arquitecturas:**
  - P2P
  - Cliente-Servidor
  - Fragmentación
  - Federada
- Centralización vs. Distribución
- Tecnologías y herramientas

Arquitecturas: P2P

- Descentralización:
  - En los sistemas P2P, no existe un servidor centralizado o nodo administrador.
  - Cada nodo (o "par") en la red tiene capacidades similares y puede actuar tanto como cliente como servidor.
  - Esta descentralización conlleva una distribución equitativa de las responsabilidades de almacenamiento, procesamiento y transmisión de datos entre todos los nodos.
- Auto-organización:
  - Los nodos en una red P2P se organizan y coordinan automáticamente para compartir recursos y datos.
  - Esta capacidad de auto-organización hace que los sistemas P2P sean robustos y flexibles frente a cambios dinámicos en la red, como la adición o eliminación de nodos.

Arquitecturas: P2P

- Descubrimiento y conexión de Pares:
  - Los nodos utilizan mecanismos de descubrimiento para encontrar otros nodos y establecer conexiones. Esto puede incluir directorios centralizados en redes P2P híbridas o algoritmos descentralizados en redes puramente P2P.
- Intercambio y replicación de datos:
  - Los datos se comparten directamente entre nodos sin pasar por un servidor central (tanto la transmisión de archivos como la distribución de fragmentos de datos).
  - La replicación de datos en varios nodos mejora la disponibilidad y la resistencia a fallos.
- Balance de carga y redundancia:
  - En una red P2P, la carga se distribuye de manera más uniforme entre los nodos, evitando cuellos de botella.
  - La redundancia inherente en estos sistemas asegura la disponibilidad continua de los datos, incluso si algunos nodos fallan.

Arquitecturas: P2P

- Redes de intercambio de archivos:
  - Las redes P2P son conocidas por su uso en el intercambio de archivos, donde los usuarios comparten archivos de manera directa sin la necesidad de un servidor central.

- Sistemas de comunicación:
  - Aplicaciones de mensajería y llamadas que utilizan una infraestructura P2P para una comunicación directa y eficiente entre usuarios.
- Blockchains y criptomonedas:
  - Las tecnologías blockchain, como Bitcoin, utilizan redes P2P para mantener un registro distribuido y seguro de transacciones.

Arquitecturas: P2P

- Ventajas:
  - Resistencia a fallos y censura debido a la falta de un punto central de control o fallo.
  - Escalabilidad, ya que añadir más nodos a menudo aumenta la capacidad y el rendimiento de la red.
- Desafíos:
  - Dificultades en garantizar la seguridad y la privacidad, dada la naturaleza abierta y descentralizada de la red.
  - Problemas de consistencia de datos, especialmente en redes grandes y altamente dinámicas.

Arquitecturas: Cliente – Servidor

- Estructura de rol definido:
  - En esta arquitectura, los roles están claramente definidos: los servidores proporcionan recursos o servicios, mientras que los clientes solicitan y utilizan estos servicios.
  - A diferencia de los sistemas P2P, hay una distinción clara entre los nodos que ofrecen servicios (servidores) y los que los consumen (clientes).
- Distribución de servidores:
  - Los servidores en este modelo están distribuidos geográficamente o en diferentes sistemas para mejorar la escalabilidad y la disponibilidad.
  - Esta distribución puede ser transparente para los clientes, quienes interactúan con el sistema como si fuera un único servidor.

Arquitecturas: Cliente – Servidor

- Balanceo de carga:
  - Los SSDD cliente-servidor a menudo implementan mecanismos de balanceo de carga para distribuir las solicitudes de los clientes de manera uniforme entre múltiples servidores, evitando así la sobrecarga de cualquier servidor individual.
- Replicación y redundancia de datos:
  - Los datos importantes pueden ser replicados en múltiples servidores, lo que garantiza que la información esté disponible incluso si uno de los servidores falla.
  - La replicación puede ser manejada a nivel de aplicación o mediante infraestructura de almacenamiento distribuido.
- Gestión de sesiones:
  - En entornos donde los clientes interactúan con el sistema en sesiones (como en aplicaciones web), la arquitectura debe manejar la persistencia y consistencia de la sesión a través de múltiples servidores.

Arquitecturas: Cliente – Servidor

- Aplicaciones web y móviles:
  - Las aplicaciones web y móviles a menudo utilizan una arquitectura cliente-servidor distribuida para servir contenido y servicios a una gran base de usuarios, distribuyendo la carga entre múltiples servidores de back-end.
- Juegos en línea y servicios de streaming:
  - Los juegos en línea y las plataformas de streaming utilizan servidores distribuidos para proporcionar una experiencia de usuario fluida y de baja latencia, manejar grandes volúmenes de tráfico y datos en tiempo real.

Arquitecturas: Cliente – Servidor

- Ventajas:
  - Mejor escalabilidad y capacidad de manejar grandes volúmenes de solicitudes y datos.
  - Mayor disponibilidad y redundancia, lo que reduce el riesgo de tiempo de inactividad.
- Desafíos:
  - Mayor complejidad en la gestión y mantenimiento de la infraestructura distribuida.
  - Necesidad de mecanismos efectivos para la sincronización y coherencia de datos entre servidores.

Arquitecturas: Fragmentación

- En un modelo basado en shards, la base de datos se divide en piezas más pequeñas y manejables, conocidas como "shards" o fragmentos.
- Cada shard contiene una porción del total de datos.
- Esta división se realiza para mejorar el rendimiento, la escalabilidad y la gestión de los datos.
- Distribución de shards:
  - Los shards se distribuyen a través de múltiples nodos o servidores, lo que permite que las operaciones de datos se realicen paralelamente en diferentes shards.
  - La distribución puede basarse en diversos criterios, como rango de claves, hash de claves o incluso de manera manual.

Arquitecturas: Fragmentación

- Clave de shard y enrutamiento:
  - Cada shard está asociado con una clave o un rango de claves. Cuando se realiza una solicitud de datos, un mecanismo de enrutamiento determina a cuál shard dirigir la consulta basándose en esta clave.
- Equilibrio y replicación de shards:
  - Para evitar desequilibrios en la carga de trabajo entre los shards, se implementan estrategias de equilibrio que redistribuyen los datos cuando es necesario.
  - Los shards suelen replicarse en múltiples nodos para garantizar la disponibilidad y la resistencia a fallos.

Arquitecturas: Fragmentación

- BBDD de gran volumen:
  - Los modelos basados en shard son especialmente útiles en bases de datos que manejan grandes volúmenes de datos y requieren operaciones de lectura y escritura de alta velocidad (MongoDB).
  - Ejemplos: BBDD y RRSS

- Sistemas de análisis de datos en tiempo real:
  - Aplicaciones que requieren análisis y procesamiento rápido de datos, como sistemas de monitoreo y análisis en tiempo real, se benefician del paralelismo que ofrecen los shards.

Arquitecturas: Fragmentación

- Ventajas:
  - Mayor escalabilidad, ya que agregar más nodos permite distribuir los shards de manera más amplia y manejar más datos.
  - Mejora en el rendimiento, debido al procesamiento paralelo y la reducción de cuellos de botella.
- Desafíos:
  - Complejidad en el diseño y la gestión, especialmente en cuanto a la distribución equitativa de los datos y la gestión de shards.
  - Dificultades en mantener la consistencia y realizar transacciones complejas a través de múltiples shards.

Arquitecturas: Federadas

- Una arquitectura federada conecta diferentes sistemas de BBDD o sistemas de almacenamiento, cada uno con su propia gestión y almacenamiento de datos, en un único sistema federado.
- Permite a los usuarios acceder y manipular datos en múltiples bases de datos como si fueran una sola entidad.
- Autonomía y heterogeneidad:
  - Cada sistema en una federación mantiene cierto grado de autonomía, operando bajo su propia administración y con sus propios esquemas de datos.
  - La arquitectura es capaz de manejar la heterogeneidad, integrando sistemas que pueden diferir en términos de modelos de datos, plataformas y tecnologías.

Arquitecturas: Federadas

- Interfaz de acceso unificado:
  - La arquitectura federada proporciona una interfaz unificada para acceder a los datos, lo que simplifica las consultas y transacciones que abarcan múltiples bases de datos.
- Traducción y mapeo de esquemas:
  - Se utilizan mecanismos para traducir y mapear diferentes esquemas de datos, permitiendo que las consultas se realicen de manera coherente a través de sistemas heterogéneos.
- Gestión de consultas y transacciones:
  - La arquitectura debe manejar eficientemente las consultas y transacciones distribuidas, asegurando que los resultados sean consistentes y fiables a pesar de la distribución y autonomía de los sistemas individuales.

Arquitecturas: Federadas

- Entornos empresariales multidepartamentales:
  - Las organizaciones con diferentes departamentos que operan sistemas de datos separados pueden utilizar arquitecturas federadas para un acceso y análisis de datos integrados sin comprometer la autonomía departamental.
- Colaboración entre organizaciones:
  - Permite a distintas organizaciones colaborar y compartir datos sin necesidad de fusionar sus sistemas de almacenamiento de datos existentes.

Arquitecturas: Federadas

- Ventajas:
  - Flexibilidad y capacidad para integrar sistemas de datos diversos manteniendo la independencia de cada sistema.
  - Facilita el acceso y análisis de datos a gran escala, potenciando la toma de decisiones basada en información más completa y variada.
- Desafíos:
  - Complejidad en la coordinación y gestión de datos entre sistemas autónomos.
  - Problemas potenciales en el rendimiento debido a la sobrecarga generada por la integración de múltiples sistemas.

Índice

- Sistemas de persistencia distribuidos
- Arquitecturas
- **Centralización vs. Distribución**
- Tecnologías y herramientas

Centralización vs. Distribución

- Ubicación de los datos:
  - En sistemas centralizados, todos los datos se almacenan en un único servidor o ubicación central. Esto puede simplificar la gestión y el control, pero también crea un punto único de fallo y puede limitar la escalabilidad.
  - Los sistemas distribuidos almacenan datos en múltiples nodos, lo que reduce el riesgo de un punto único de fallo y permite una mayor escalabilidad y resistencia a fallos.
- Acceso a los datos:
  - Los sistemas centralizados pueden sufrir cuellos de botella en el rendimiento cuando hay múltiples solicitudes simultáneas, debido a la limitada capacidad de procesamiento del servidor central.
  - En los SSDD, las solicitudes se pueden manejar en paralelo a través de varios nodos, mejorando el rendimiento y la eficiencia.

Centralización vs. Distribución

- Complejidad administrativa:
  - Los sistemas centralizados suelen ser más fáciles de administrar debido a su estructura unificada.
  - La actualización, el mantenimiento y la seguridad pueden gestionarse de forma centralizada.
  - Los SSDD requieren una gestión más compleja debido a su naturaleza dispersa.
  - La coordinación entre nodos, la sincronización de datos y la seguridad distribuida presentan desafíos adicionales.
- Resiliencia y recuperación de desastres:
  - En sistemas centralizados, la recuperación de desastres puede ser más complicada, ya que un fallo en el servidor central puede impactar todo el sistema.
  - Los SSDD, con su redundancia de datos incorporada y la distribución geográfica, suelen ser más resistentes a fallos y desastres, facilitando la recuperación.

Centralización vs. Distribución

- Escalabilidad:
  - Los sistemas centralizados pueden tener dificultades para escalar, ya que esto a menudo requiere actualizaciones de hardware caras y complejas.
  - Los SSDD permiten la escalabilidad horizontal, agregando más nodos para aumentar la capacidad, lo cual es generalmente más económico y flexible.
- Rendimiento bajo alta demanda:
  - Los sistemas centralizados pueden sufrir degradaciones en el rendimiento bajo cargas de trabajo pesadas, mientras que los SSDD pueden manejar mejor estas demandas al distribuir la carga entre varios nodos.

Índice

- Sistemas de persistencia distribuidos
- Arquitecturas
- Centralización vs. Distribución
- **Tecnologías y herramientas**
  - BBDD relacionales distribuidas
  - BBDD NoSQL
  - Sistemas de archivos distribuidos

BBDD relacionales distribuidas

- Las BBDD distribuidas almacenan datos en varios nodos, que pueden estar ubicados en diferentes servidores o incluso en diferentes centros de datos.
- A pesar de la distribución física, estos sistemas presentan una vista unificada de los datos al usuario o a la aplicación.
- La replicación involucra mantener copias de datos en múltiples nodos para garantizar la disponibilidad y resistencia a fallos.
- El particionamiento distribuye los datos en diferentes nodos para mejorar el rendimiento y la escalabilidad.

BBDD relacionales distribuidas

- Las bases de datos relacionales distribuidas representan una extensión del modelo relacional tradicional para operar en un entorno distribuido.

- Estructura relacional en un entorno distribuido:
  - Mantienen la estructura relacional de tablas, filas y columnas, pero los datos se almacenan y procesan en múltiples nodos de una red.
  - A pesar de la distribución física, presentan una vista unificada de la base de datos al usuario.

BBDD relacionales distribuidas

- Coherencia de datos:
  - Al igual que las bases de datos relacionales tradicionales, enfatizan la integridad y la coherencia de los datos, manteniendo las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) a través de transacciones distribuidas.
- Replicación y particionamiento de datos:
  - Implementan la replicación de datos para garantizar la disponibilidad y la tolerancia a fallos.
  - El particionamiento de datos permite distribuir las cargas de trabajo de manera más eficiente a través de los nodos.

BBDD relacionales distribuidas

- Transacciones distribuidas:
  - Gestionan transacciones que involucran datos en múltiples ubicaciones, asegurando la coherencia y atomicidad a lo largo de toda la operación.
  - Utilizan protocolos como el compromiso de dos fases (Two-Phase Commit) para garantizar que todas las partes de una transacción se completen con éxito o se deshagan por completo.

- Consultas y acceso a datos:
  - Las consultas se ejecutan de manera que pueden acceder y combinar datos de diferentes nodos, proporcionando resultados como si los datos provinieran de una sola fuente.

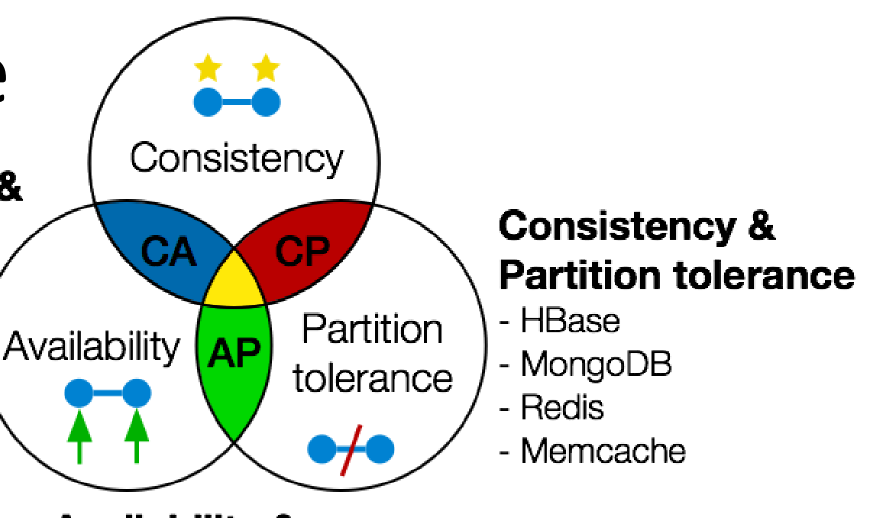
BBDD relacionales distribuidas

- Oracle Real Application Clusters (RAC):
  - Permite que múltiples instancias de la base de datos Oracle accedan y procesen un conjunto compartido de datos.
  - Es ampliamente utilizado en entornos empresariales para aplicaciones críticas que requieren alta disponibilidad y escalabilidad.

- Microsoft SQL Server Always On:
  - Una solución de alta disponibilidad y recuperación ante desastres que permite a los usuarios configurar múltiples réplicas de bases de datos SQL Server en diferentes nodos.
  - Proporciona un entorno operativo continuo y protección contra fallos.

Teorema CAP

- El teorema CAP, también conocido como Principio de Brewer, es un concepto fundamental en el diseño de sistemas de bases de datos distribuidas.



Fuente: https://debeando.com/teorema-cap.html

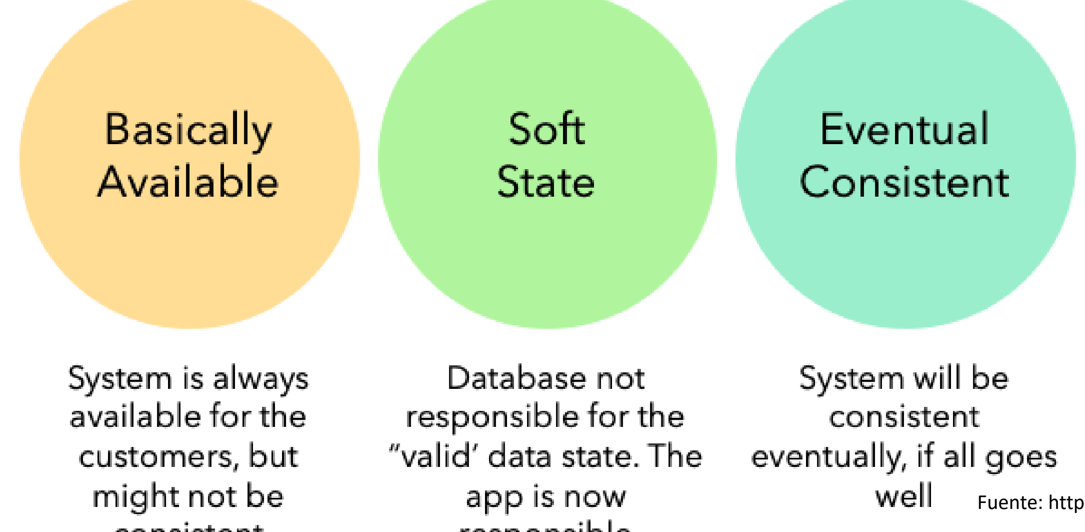
- Este teorema establece que un sistema de base de datos distribuida solo puede proporcionar dos de las siguientes tres garantías simultáneamente:

Teorema CAP

- Consistencia (C):
  - Todos los nodos ven los mismos datos al mismo tiempo. Cualquier lectura recibirá la versión más reciente de un dato escrito.
- Disponibilidad (A):
  - Cada solicitud recibe una respuesta sobre si fue exitosa o fallida, pero sin garantizar que todos los nodos contengan la misma versión más reciente del dato.
- Tolerancia a Particiones de Red (P):
  - El sistema continúa funcionando a pesar de cualquier número de fallos de comunicación entre los nodos.
- En la práctica, se tiene que elegir un par.

BASE

- BASE es un acrónimo que describe las propiedades de los sistemas que no siguen estrictamente las propiedades ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) típicas de las bases de datos relacionales tradicionales.
- En cambio, BASE se enfoca en la escalabilidad y rendimiento en sistemas de bases de datos distribuidas.



Fuente: https://f5sal.medium.com/database-selection-design-part-iv-f4577ba049c5

BASE

- Basicamente Disponible (Basically Available):
  - Indica que el sistema garantiza la disponibilidad en términos de capacidad para realizar operaciones, aunque no garantiza una respuesta inmediata o coherente.
- Soft-state:
  - El estado del sistema puede cambiar con el tiempo, incluso sin entrada. Esto se debe a que el sistema puede eventualmente propagar actualizaciones a todos los nodos, pero no garantiza que ocurra inmediatamente.
- Consistencia Eventual (Eventual Consistency):
  - Aunque el sistema no garantiza la consistencia inmediata, asegura que, eventualmente, todos los nodos contendrán los mismos datos.

BBDD NoSQL

- BBDD Not only SQL (BBDD NoSQL).
- Son un conjunto diverso de tecnologías de bases de datos que se diseñaron para abordar varias limitaciones de las bases de datos relacionales tradicionales.
- Especialmente útiles en contextos de grandes volúmenes de datos, alta escalabilidad y operaciones distribuidas.
- A diferencia de las bases de datos relacionales, las BBDD NoSQL no se basan estrictamente en tablas y a menudo no usan SQL como su principal lenguaje de consulta.

BBDD NoSQL

- Esquemas flexibles:
  - No requieren un esquema fijo o definido previamente.
  - Esto permite almacenar datos que pueden tener estructuras diversas sin necesidad de modificar el diseño de la base de datos.
- Escalabilidad horizontal:
  - Están diseñadas para escalar fácilmente añadiendo más máquinas o nodos al sistema, lo que es ideal para aplicaciones que necesitan manejar un gran volumen de tráfico o datos.
- Modelos de datos diversos:
  - Soportan una variedad de modelos de datos, incluyendo clave-valor, documentos, columnas anchas y grafos, cada uno adecuado para diferentes tipos de aplicaciones y requisitos de uso.

BBDD NoSQL

- Coherencia eventual:
  - Muchas bases de datos NoSQL ofrecen una consistencia eventual, lo que significa que todas las copias de los datos eventualmente se sincronizarán, pero no necesariamente inmediatamente después de una operación de escritura.
- Optimizadas para rendimiento y velocidad:
  - En general, están optimizadas para operaciones rápidas de lectura y escritura y pueden ser más eficientes que las bases de datos relacionales para ciertos tipos de consultas y operaciones.

BBDD NoSQL

- Clave-Valor:

  - Almacenan los datos en pares de clave-valor.
  - Son simples y eficientes para operaciones de lectura y escritura.
  - Ejemplo: Redis.

- Documentos:
  - Almacenan datos en documentos (generalmente en formatos como JSON o BSON), lo que los hace ideales para almacenar datos complejos y jerárquicos.
  - Ejemplo: MongoDB.

BBDD NoSQL

- Columnas anchas:
  - Optimizadas para consultas en grandes conjuntos de datos, almacenan datos en filas y columnas, pero de manera más flexible y distribuida que las BBDD relacionales. Ejemplo: Apache Cassandra.

cassandra

- Grafos:
  - Diseñadas para almacenar y manejar datos relacionales complejos, son ideales para representar redes y relaciones.
  - Ejemplo: Neo4j.

neo4j

Sistemas de archivos distribuidos

- Los sistemas de archivos distribuidos son tecnologías clave en los sistemas de persistencia distribuidos.

- Están diseñados para almacenar y acceder a archivos y datos a través de una red de nodos, facilitando el acceso y la manipulación de datos distribuidos como si estuvieran almacenados en una ubicación local.

Sistemas de archivos distribuidos

- Almacenamiento de datos en múltiples nodos:
  - En un sistema de archivos distribuido, los archivos y datos se almacenan en varios servidores o nodos en una red.
  - Esto permite que los datos se almacenen de manera eficiente y estén accesibles desde cualquier nodo del sistema.
- Transparencia de ubicación:
  - Para los usuarios y aplicaciones, el sistema parece ser un único sistema de archivos coherente, independientemente de dónde se almacenen físicamente los datos.
  - Esto es conocido como transparencia de ubicación.
- Escalabilidad y fiabilidad:
  - Estos sistemas están diseñados para ser altamente escalables, permitiendo añadir más nodos de almacenamiento para aumentar la capacidad.
  - También ofrecen redundancia y tolerancia a fallos, replicando datos en nodos.

Sistemas de archivos distribuidos

- Replicación y distribución de datos:
  - Los datos se replican en múltiples nodos para garantizar la disponibilidad y la resistencia a fallos.
  - La distribución de datos puede ser basada en varios factores, como la carga, la proximidad geográfica o la frecuencia de acceso.
- Consistencia y sincronización:
  - Los sistemas de archivos distribuidos utilizan diversos algoritmos para mantener la consistencia de los datos entre los nodos.
  - Estos pueden incluir modelos de coherencia eventual o sistemas más estrictos de coherencia.

Sistemas de archivos distribuidos

- Hadoop Distributed File System (HDFS):
  - Un componente clave del ecosistema Hadoop, HDFS es ideal para aplicaciones con grandes conjuntos de datos.
  - Distribuye los datos en bloques a través de múltiples nodos, proporcionando alta tolerancia a fallos y facilidad de escalabilidad.
- Google File System (GFS):
  - Diseñado para aplicaciones que requieren un procesamiento de datos extenso, GFS maneja el almacenamiento de datos en una gran cantidad de nodos, optimizando la eficiencia y la confiabilidad.
- Network File System (NFS):
  - Uno de los sistemas de archivos distribuidos más antiguos y comunes, NFS permite a los usuarios acceder a archivos en una red de manera similar a cómo accederían a los almacenados en su propio sistema.

Sistemas de archivos distribuidos

- Ventajas:
  - Escalabilidad para manejar grandes volúmenes de datos.
  - Alta disponibilidad de datos y resistencia a fallos a través de la replicación.
  - Acceso y gestión de archivos eficientes a través de redes.

- Desafíos:
  - Mantener la consistencia de los datos puede ser complejo, especialmente en sistemas con alta latencia de red o cuando se requiere coherencia en tiempo real.
  - La gestión y el mantenimiento de un sistema de archivos distribuido a gran escala pueden ser técnicamente desafiantes.

Sistemas Distribuidos

Bloque IV

Sistemas de persistencia distribuidos

Tema 5

Bases de Datos Distribuidas

©2023 Autor Nicolás H. Rodríguez Uribe
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Nicolás Rodríguez
nicolas.rodriguez@urjc.es