# SERVLET

Hasta ahora hemos pasado por una serie de capítulos que describen las funcionalidades de back-end del servidor Java EE 7. Hemos hablado de la API de persistencia de Java. Hablamos de los beans Java empresariales. Hablamos de los servicios web. Servicios de mensajería. Ahora vamos a hablar sobre la parte frontal, el contenido del contenedor web y las tecnologías del contenedor web, y habrá una serie de capítulos sobre eso, y comenzaremos con los servlets: crear aplicaciones web Java con el uso de servlets.

Bueno, mientras pasamos del back-end al front-end, nos gustaría comenzar aquí con los conceptos básicos de un protocolo HTTP porque vamos a usar HTTP no solo para servlets sino también para otras tecnologías de contenedores web. Por lo tanto, debemos sentirnos cómodos con la forma en que funciona el protocolo HTTP y luego comprender cómo podemos manejar las llamadas HTTP con servlets y, más adelante, con GSP, GSF, servicios REST, etcétera.

Así que crear servlets, por supuesto, es otra cosa que cubriremos en este capítulo. Servicio de mapeo a URLs. Manejo de llamadas HTTP. Encabezados, parámetros, cookies, todos los aspectos de las conversaciones HTTP. Y usaremos servlets para generar contenido dinámico y entregarlo a nuestros clientes. Necesitamos entender cómo gestionamos el ciclo de vida del servlet con devoluciones de llamadas de contenedores y varios oyentes.

Por supuesto, usaremos beans CDI de nuestros servlets. También hablaremos sobre el uso de filtros, la tecnología WebFilter, los servlets asíncronos, el uso de la API NIO con servlets y el manejo de errores. Comencemos con lo básico.

Echemos un vistazo a los conceptos básicos del protocolo HTTP. Cuando realiza una solicitud HTTP, utiliza uno de estos métodos HTTP diferentes: GET, POST, PUT, DELETE, OPTIONS, HEADERS, para generar la solicitud y enviarla al servidor. Ahora, de forma predeterminada, si solo escribe la URL en un navegador y la envía, lo que sucederá es que usará el método GET y todos los parámetros, si tiene alguno, se concatenarán en una sola URL. . Todo se pasará como una sola URL. Todos los pares de parámetros de valor de nombre.

Eso podría no ser muy conveniente si tiene muchos parámetros o tamaños de parámetros grandes. Ya sabes, todos están presentes en el mismo lugar. Pero supongo que está bien si llamas a una página inicial a través del método GET. Es posible que no haya ningún parámetro allí, así que está bien. Y una vez que necesite comenzar a enviar cualquier valor, puede controlar la forma y el método con el que se envían los valores mediante el uso de formularios HTML.

Entonces puedes decir método POST. Y en este caso, observe que los parámetros han pasado como pares de valor de nombre separados en el cuerpo de un paquete HTTP. No están concatenados en una URL en la parte superior. Se transmiten como un cuerpo de paquete. Entonces, supongo que es más conveniente usar estos otros métodos HTTP si desea enviar algunos datos del cliente.

Ahora, en respuesta a la solicitud HTTP que ha enviado, un servidor le devolverá algún tipo de respuesta HTTP, que contendrá un código de respuesta. Puede haber diferentes códigos de respuesta y, según el grupo de códigos de respuesta al que pertenezcan, puede determinar su tipo de naturaleza. Ya sabes, ¿qué representa ese código de respuesta?

Por ejemplo, cualquier código de recurso que comience con 100 significa que es algún tipo de código informativo. Como, por ejemplo, cambiar al protocolo. No sé, tal vez quieras cambiar al protocolo Web Socket o algo así. O si tiene una respuesta exitosa (obtuvo códigos de respuesta de una serie de 200 códigos, como 200OK), eso significa que se envió la solicitud y una respuesta solo confirma que está bien.

Los 300 códigos indican redirecciones. Si necesita ir a otra URL, por ejemplo. Así que moviendo las páginas. Páginas en movimiento temporal o permanente.

Los códigos 400 indican todo tipo de errores, como el 404 es un clásico, ¿no? Página no encontrada. Y los códigos 500 indican errores graves del servidor, como el código 500 que... no sé, servicio no disponible, 503, que dice que algo salió mal en un nivel de servidor.

Entonces el paquete contendrá el código de respuesta. Puede contener otros encabezados que describan la duración del contenido, el tiempo del contenido. Entonces, qué tipo de contenido tiene el paquete. Ya sabes, marcas de tiempo, encabezados de fecha y el cuerpo del paquete. Cualquier contenido que el servidor devolvió al cliente.

El cuerpo no está necesariamente siempre presente. Eso depende. Por ejemplo, imagine que está configurando la solicitud GET en el servidor y espera que se devuelva el cuerpo de una página. Está bien. Es un código 200 que puede obtener del servidor. Implica, con el método GET, recuperar algún documento.

Digamos que llama a la misma página pero con el método HEAD en lugar del método GET. HEAD dice explícitamente que no habrá ningún cuerpo en esa respuesta. El código de respuesta puede ser 200, está bien, pero solo habrá encabezados. Esa es la naturaleza del método HEAD. Funciona igual que el método GET pero solo devuelve encabezados sin el cuerpo.

Por lo general, en realidad se usa para saber cuándo el contenido aún es bueno para usar si está almacenado en caché. Por lo tanto, puede verificar, por ejemplo, los encabezados de caché, las fechas de vencimiento y cosas por el estilo, y decidir cuándo la copia en caché del documento aún está bien o si desea actualizarla. Ese es un caso de uso del método HEAD.

POST, el cuerpo de la respuesta debe contener el resultado de la acción solicitada. Así que tal vez has publicado algo en un servidor. TRACE, haciendo eco de la solicitud. Así que es principalmente por razones de depuración. Simplemente devolverá lo que haya enviado al servidor. Entonces, lo que suceda con la respuesta depende de la naturaleza de la solicitud y qué es lo que genera ese componente. Qué tipo de contenido dinámico debe producir el componente del sitio del servidor. Pero siempre tiene algún tipo de código de respuesta para indicar cuál es exactamente la naturaleza de la respuesta.

Los componentes del sitio del servidor que manejan estas comunicaciones HTTP son servlets. Un servlet se puede describir con una simple anotación de servlet web, donde puede asignarlo a la URL. Ahí tienes Y, sin duda, también podría describirse mediante un archivo web XML. Nuevamente, servlet mapeado a un patrón de URL. Los patrones de URL pueden contener comodines. Puede asignar los servlets no solo a una URL específica sino, no sé, para interceptar una cantidad de llamadas a diferentes URL.

Y luego, dentro de un servlet, escribirá un código que reaccionará a estos métodos HTTP: GET, POST. Entonces manejaremos estas llamadas dentro de un servlet. Y eso está en la página siguiente, así que echemos un vistazo.

Ahora tenemos varias opciones aquí. En primer lugar, puede anular el método llamado SERVICIO. Bien, ahí está. Un servicio de método acepta el parámetro de solicitud de servlet HTTP que encapsula todo lo que el cliente le envía. Parámetro de respuesta de servlet HTTP que encapsula todo lo que desea devolver al cliente y puede generar excepciones de servlet y E/S. Si anula el método SERVICIO, entonces está creando un código que reacciona a cualquier llamada del cliente, independientemente de cuándo sea GET, POST, PUT, DELETE, lo que sea. Entonces estás diciendo que quiero escribir un algoritmo que reaccione a cualquiera de las llamadas al método HTTP.

Sin embargo, una clase de servlet proporciona variantes del método SERVICIO, en realidad, llamado doGet, doPost, doPut, doDelete. Sus parámetros son exactamente los mismos que los parámetros del método SERVICIO, por lo que la solicitud del servlet HTTP, la respuesta del servlet HTTP siguen exactamente las mismas excepciones. Las firmas de método de estos métodos son idénticas.

Si no anula el método SERVICIO, si decide no hacerlo, y en su lugar está anulando los métodos GET, POST, PUT, DELETE, etcétera, el método predeterminado SERVICIO que ya existe en una clase de servlet. - porque fíjate, estás extendiendo aquí una clase llamada servlet HTTP, ¿verdad? Entonces, el método predeterminado SERVICIO que ya está allí en una clase de servlet HTTP tiene un poco de código que detecta qué método HTTP estaba usando el cliente y llama al método doGet, doPost, doPut, doDelete por usted. Entonces, si desea escribir el código que hace algo cuando el cliente invoca el servlet mediante el método GET y algo más cuando el cliente llama a un servlet con el método POST y algo más cuando algún otro método, en lugar de anular el método SERVICIO, puede simplemente anular métodos GET, POST, PUT para tener sus reacciones únicas a estos diferentes tipos de llamadas.

Finalmente, puede crear otro método, que técnicamente hablando no anula realmente un método de clase principal. Es solo un método arbitrario, de verdad. Se llama solicitud de proceso. Entonces, técnicamente hablando, este no es un nombre fijo porque no está anulando aquí un método de clase principal.

La idea de processRequest es esta. ¿Qué sucede si desea llamar a este método desde el método SERVICIO incondicionalmente, y luego, dentro de ese método, tiene una lógica que será común para todas las operaciones GET, POST, PUT y DELETE, y luego, desde este método, llama a doGet, doPost, doPut, doDelete, si así lo desea, para tener una lógica exclusiva de los métodos GET, POST, PUT o DELETE. Entonces, con processRequest, simplemente está creando un método personalizado en el que intenta aprovechar los beneficios de ambos mundos, tiene un poco de procesamiento que es común para todos los métodos HTTP, y luego también tiene estos otros métodos HTTP que procesan cosas específicas para GET, POST, PUT o DELETE, cualquier otra operación de método HTTP. Todos los métodos aceptan los mismos parámetros y todos lanzan las mismas excepciones. Es idéntico.

Aquí hay un ejemplo. Así que aquí hay un método, processRequest, ¿sí? Aceptar una solicitud HTTP y objetos de respuesta HTTP. Agregue estas excepciones de servlet y E/S. Y eso es lo primero que podría hacer: averiguar de qué método HTTP proviene la llamada. Entonces, ¿es el método GET? ¿Es el método PUT? Sea lo que sea, ¿sí? Y luego es solo una cadena, por lo que puede escribir en algún tipo de caso de cambio y decir, si es GET, haga esto. Si es POST, hazlo. O, si es lo mismo para usted, solo tenga una forma de manejar esa lógica.

Supongo que si desea generar algo para su cliente, debe indicar qué tipo de contenido va a producir. Dígale al cliente que será un contenido HTML. Va a ser, no sé, codificado en UTF o algo así. Obtenga el escritor de impresión e imprima lo que necesite. Al final, no olvides cerrarlo. Esa es la producción de la respuesta al cliente.

Cuando maneja la entrada del cliente en un servlet, es posible que desee obtener encabezados de paquetes HTTP: request.getHeaderNames. Obtenga su enumeración de encabezados, repita eso. Si bien tiene más elementos, consígueme el siguiente. Consígueme el valor del encabezado. Haz lo que quieras con estos encabezados. O parámetros. Consígame los nombres de los parámetros de la solicitud. Enumeración de nombres de parámetros, iterar a través de parámetros, obtener cada nombre de parámetro, obtener cada valor, hacer lo que quiera con eso.

O galletas. Ligeramente diferente. La solicitud, getCookies, le devuelve la matriz de cookies, que en realidad podría ser nula si no se enviaron cookies al cliente antes de ese punto. Si no es nulo, puede iterar a través de esta matriz de cookies. Obtenga cada cookie, nombre y valor. Haz lo que quieras con él.

Si desea crear una nueva cookie, ahí lo tiene. Nombra el par de valores, constrúyelo, establece su tiempo de caducidad, edad máxima y agrega la cookie a la respuesta. Recuerde, las cookies son valores que desea enviar al cliente para que el cliente las memorice en su nombre, por lo que la próxima vez que el cliente acceda al servidor, alguna otra página, tal vez, algún otro servlet que pueda obtener estas cookies como valores que usted he enviado a ese cliente antes. Entonces, es una forma de implementar algún tipo de almacenamiento dentro de un nivel de cliente de sus valores.

Por supuesto, cuando su servlet genera el contenido para el cliente, podría generar absolutamente cualquier contenido que desee. No tiene que ser texto o HTML. De hecho, en este caso particular, generé una imagen JPEG; bueno, en realidad no estoy generando una imagen JPEG, solo estoy leyendo el archivo y luego transmitiéndolo al flujo de salida del servlet, que es un poco un poco de trampa realmente, porque si solo quería obtener el archivo estático, bueno, ¿por qué necesita un servlet? Simplemente iría a la URL, señalaría algún archivo de imagen y terminaría. Pero imaginemos que no solo estoy leyendo el archivo. Tal vez realmente estoy generando dinámicamente esa imagen.

¿Entonces por qué no? Un servlet ciertamente podría producir otros tipos de contenido. Siempre y cuando le diga al cliente qué tipo de contenido esperar, responda ese tipo de contenido, está perfectamente bien generando lo que quiera. El tipo de contenido es realmente imprescindible, porque de lo contrario, el cliente no sabría realmente qué hacer con ese flujo de datos que le envías. La longitud del contenido es opcional, pero es una buena idea. Si el cliente no sabe cuál es la longitud del contenido, ¿cómo haría el cliente cosas como las barras de progreso? Ya sabes, muestra cuánto tarda en cargar la página. El navegador no podrá predecir eso porque no sabe qué tan grande es el archivo que estás enviando. Por lo tanto, probablemente sea algo elegante decirle al cliente qué tan grande es el contenido que estás generando.

Métodos de ciclo de vida. Un servlet no solo puede reaccionar a las solicitudes que un cliente envía a ese servlet, que es, recuerde, los métodos SERVICE, doGet, doPut, doPost, etcétera, sino que también puede reaccionar a los métodos de ciclo de vida que inicializan el servlet cuando es cargado en la memoria. Anulando el método llamado INIT. Y cuando un servlet está sobrecargado de memoria, anulando un método llamado DESTRUIR. Si tiene algún tipo de inicialización común que le gustaría hacer para ese servlet, opcionalmente, el método INIT puede aceptar el conflicto del servidor como argumento, lo que le permite obtener la configuración del archivo XML web, esencialmente. Entonces, su proyección de las propiedades web XML en el código del servlet.

De forma predeterminada, el servlet es realmente un singleton, por lo que se cargará en la memoria solo una vez. Y luego se usará la misma instancia de un servlet para atender todas sus otras solicitudes simultáneas. Entonces, muchos subprocesos diferentes se conectan simultáneamente al método del servlet. Pero si desea una inicialización o limpieza común, puede anular los métodos INIT y DESTROY.

Si desea inyectar CDI Beans en sus servlets, simplemente describa el CDI Bean, como este bean de ámbito de sesión, por ejemplo, e inyéctelos. Ahí tienes Inyectar ese bean y simplemente comenzar a usar los métodos de bean.

El archivo XML del bean, que se ve en la parte superior de esta página, es opcional. Puede usarlo para controlar el procesamiento de CDI Bean, como, por ejemplo, descubrir todos los beans anotados en su aplicación, que de todos modos es predeterminado. Más sobre eso en el apéndice de CDI Bean. Desde la perspectiva de cómo los servlets administran la inyección del CDI Bean, esta página le muestra todo lo que necesita saber. Simplemente inyecte y use el código y, de todos modos, estará en el ámbito correcto.

Si está en su servlet, use un código de seguimiento de sesión. Para lograr el seguimiento de la sesión, puede usar una cookie o su parámetro de URL llamado jsessionid. Desafortunadamente, la cookie podría no ser particularmente confiable. Verá, el seguimiento de la sesión es esta idea de asociar el cliente con una sesión HTTP particular en un servidor.

El servidor puede enviar una cookie al cliente y decir, bueno, esta es la identificación de la sesión, ¿verdad? Entonces, la próxima vez que el cliente llame a un servidor, pasará la cookie. Pero, ¿qué sucede si el cliente desactivó el soporte de cookies? ¿Qué pasa si el servlet no puede, por alguna razón, enviar la cookie al cliente?

A continuación, puede utilizar este método interesante. Se llama encodeURL. Agrega a la URL un parámetro jsessionid. Esto es lo que hace. Agrega este ID de sesión en particular. ID de sesión actual. [INAUDIBLE] selecciona automáticamente cualquiera que sea la sesión actual, coloca la ID en la URL, de modo que cuando construye un contenido dinámico, algún tipo de página, lo envía de vuelta al cliente y crea un enlace en el que se puede hacer clic allí. en la página, ese enlace contendrá este parámetro jsessionid, con la ayuda del cual cuando el cliente haga clic en ese enlace y avance a la página siguiente, la página siguiente sabrá qué sesión se supone que debe usar. Asociará la llamada con la misma sesión, aunque sea una llamada nueva de ese cliente.

Otros eventos del ciclo de vida del contenedor web se pueden manejar con varios detectores de eventos. Ahora bien, esto es un poco complicado porque aquí hay dos tipos de oyentes. Y hay diferentes ejemplos, así que déjame hablarte de ellos. En primer lugar, los oyentes pueden estar asociados con diferentes tipos de entorno de memoria. [INAUDIBLE] los oyentes rastrean y solicitan eventos o eventos de sesión o eventos de aplicación. Así que, hasta ahora, todo bien. Estas son estas categorías.

Pero dentro de cada categoría, también podría estar rastreando diferentes tipos de cosas. Por ejemplo, se puede utilizar un detector de solicitudes de servlet para rastrear el punto en el que se crea la solicitud y el punto en el que se destruye la solicitud. El detector de atributos de solicitud de servlet podría usarse para rastrear un punto en el que se insertó cualquier tipo de objeto como atributo de la solicitud, como CDI Bean asociado con la solicitud o ese bean se actualizó o el objeto se desasoció. Entonces, cada vez que solicita un atributo, lo elimina o lo actualiza, reacciona a eso.

Oyentes de sesión: oyentes que reaccionarán a la creación o destrucción de la sesión o a la colocación o eliminación de un atributo en una sesión. Oh oyentes de contexto de servlet. La misma idea. Cuando inicializa la aplicación, cuando destruye la aplicación, ese es el oyente de contexto de servlet. Y cuando coloca algunos atributos en la aplicación, contexto, como los atributos del ámbito de la aplicación.

Ahora he resaltado cuatro oyentes aquí como ejemplos. Eso es el oyente de sesión HTTP y el oyente de contexto de servlet. Se implementan como oyentes web en el primer ejemplo de esta página. Puede implementar en la misma clase una cantidad de oyentes, si lo desea, por lo que, en este caso particular, el código reaccionará a la creación de la sesión, la destrucción de la sesión, la inicialización del contexto o la destrucción del contexto. Estos son los eventos que ocurren con las sesiones y la aplicación.

Pero el segundo oyente es bastante interesante porque no es un oyente web. ¿Notaste que la anotación aquí es una anotación de CDI Bean? Aparentemente, CDI Bean puede contener un código que reacciona cuando ese bean se inyecta o elimina de varios contextos. En este caso, tenemos dos oyentes aquí: un oyente de activación y un oyente vinculante. El detector de activación de dos métodos: los dos primeros, session didActivate y session willPassEvade.

Por ejemplo, imagine que tiene este bean de ámbito de sesión. Agregar ámbito de sesión. Solo eso, ¿verdad? Luego lo inyecta en una variedad de servlets y páginas y el paradero en el que desea usarlo. Una sesión a la que se inyectó se serializó para pasar de un nodo de clúster a otro, replicando la sesión en todos los nodos de clúster. Entonces se desactivó en una máquina y se reactivó en otra.

¿Qué pasa si ese frijol de alcance de sesión quiere saber cuándo sucede eso? Como si necesitara reiniciar algo, algunos recursos que son específicos de la máquina. Si solo se migró y se restableció en una computadora diferente, tal vez le gustaría hacer algo al respecto. Entonces, por lo tanto, tiene estos métodos willPassEvade y didActivate.

Otro ejemplo interesante con un oyente vinculante es este: valor vinculado, valor no vinculado. Cuando el bean se inyecta por primera vez en algún componente de su aplicación, cuando se inicializa, entra en el ámbito de la sesión. ¿Sabe qué servlet, página o componente de su aplicación logró ser invocado primero para realizar dicha inyección? Ni idea. no se cual Sé que se inyectó en alguna parte, pero ¿cuál de ellos lo hizo primero?

Así que, independientemente de quién lo haya hecho primero, puedo escribir el código en el propio bean (valor limitado) en ese método que sé que se invocará justo en el momento en que este bean entra en el ámbito de la sesión. Cuando se instancia. Cuando entra en el ámbito de la sesión. Y cuando se quita del alcance, nuevamente, no tengo idea de cuándo sucederá. Podría ser en cualquier momento cuando expire la sesión. Pero es difícil de predecir porque podría haber sido inyectado en todo tipo de componentes. Pero puedo hacer que ese frijol reaccione al hecho de que ya no está en la sesión. Y ese es el valor sin vincular cuando lo elimina del contexto de la sesión.

Esos son solo algunos ejemplos de oyentes, pero lo más importante de esta página es que te brinda esta clasificación de tipos de eventos. El primer ejemplo mostraba eventos que podía escuchar para rastrear cosas alrededor de su ciclo de vida de una sesión o contexto de servlet o lo que sea. Un segundo ejemplo es un tipo de eventos que puede estar escuchando desde el CDI Bean, siendo el propio bean notificado por un contenedor sobre lo que le está sucediendo. Por lo tanto, podría escribir algún código tal vez para reiniciar cosas o limpiar o reaccionar a estos eventos que ocurren alrededor del bean dentro del contenedor.

REQUEST DISTPACHER

Ahora hablemos de los despachadores de solicitudes. Un despachador de solicitudes nos permite obtener acceso a otro servlet, a otra URL, y ayudar a un servlet en particular a realizar sus tareas. Manejo tan cooperativo de la solicitud. Entonces, en este ejemplo particular, imaginemos un par de servlets. Servlet A, servlet B.

Un usuario llama, el usuario invoca, el servlet A. El usuario no sabe nada del servlet B. Pero el servlet A decide internamente solicitar la ayuda del servlet B para completar sus tareas. Hay dos cosas que podrían hacer.

El servlet A puede reenviar una solicitud al servlet B, lo que significa que el servlet A dice que no produciría ningún resultado. Deje que el servlet B haga eso. O el servlet A puede comenzar a producir algún resultado y luego decir, bueno, hay una parte de la página, hay una parte en una página, que me gustaría que produzca el servlet B. Entonces, en este caso, el servlet A incluye cualquier cosa que haga el servlet B en su propia salida.

Por supuesto, en un segundo escenario en el que está utilizando la funcionalidad dispatcher.include, debe asegurarse de que está manejando correctamente esa inclusión. Porque si el servlet A ha abierto el flujo de salida y el servlet B obviamente podría producir contenido para ese flujo, el servlet B no debería cerrarlo porque la idea es que el servlet A pueda, luego de esa inclusión, continuar produciendo algún resultado. Supongo que quien abre el flujo, bueno, en este caso, el servlet A, es responsable de cerrarlo.

En el otro escenario, en un escenario directo, la idea era que el servlet A probablemente no esté produciendo ningún resultado. Entonces, en este caso, el servlet B está abriendo un flujo y el servlet B está cerrando un flujo. ¿Por qué se necesitaría el servlet A en ese escenario?

Bueno, verá, puede llevar a cabo tareas que no están relacionadas con la producción de la salida visual real del contenido para el cliente. Tal vez el servlet B esté manejando algunos parámetros, o tal vez esté resolviendo algunos problemas de seguridad. Y luego el servlet B está produciendo el contenido real. Y ese es el caso directo, supongo.

Y, ¿sabe? Probablemente nos dé un ángulo interesante sobre las formas de abordar el controlador de vista del modelo. Distribución de funcionalidades alrededor de la aplicación. Porque entonces puede tener algunos servlets enfocados, digamos, en manejar la lógica de negocios, y otros enfocados en producir la salida. Y pueden hacer estas cosas de manera cooperativa para manejar las solicitudes. Pero distribuyes las funcionalidades alrededor de los servlets.

Otra técnica de codificación interesante que puede emplear en el contenedor web es el filtro web. El filtro web se puede aplicar a servlets, páginas, bueno, a cualquier URL. De verdad, esa es la idea. El filtro web se asigna a algún patrón de URL. Podría ser una página específica, podría ser un comodín. Y obviamente puedes mapear múltiples filtros, tantos como quieras.

El filtro contiene un método doFilter. El método doFilter está secuestrando la llamada. Es muy similar en términos de la idea a los interceptores que ya hemos visto para los beans Java empresariales que interceptaban llamadas hacia los beans. Entonces, los filtros hacen algo similar, pero, por supuesto, para las URL en un contenedor web. Y un método doFilter básicamente está secuestrando esa llamada que es una llamada entrante del cliente.

El cliente no está llamando al filtro, por supuesto. El cliente está llamando a esa página, ese servlet, esa URL, y un filtro detrás de escena salta sobre esa solicitud y comienza a manejarla. Tenga en cuenta que los parámetros y el método doFilter no son una solicitud de servlet HTTP y una respuesta de servlet HTTP, sino solo una solicitud de servlet y una respuesta de servlet. Estas son clases principales de solicitud y respuesta de servlet HTTP. Y si desea obtener acceso a funciones específicas de HTTP como encabezados, método HTTP, etc., tendrá que reducir el tipo, reducir el tipo a solicitud de servlet HTTP y respuesta de servlet HTTP dentro del cuerpo del método de filtro.

Para pasar de un filtro a otro, invoque el método chain.doFilter. Chain es ese tercer argumento del doFilter. Y básicamente significa que llama a otro filtro que está asignado a la misma URL en una cadena y, finalmente, la llamada avanza a la página de destino o al servlet que está tratando de invocar allí. Si coloca cualquier código antes de chain.doFilter, ese código se ejecutará primero. Si coloca algún código después de chain.doFilter, habrá un procesamiento posterior después de que se complete todo el ciclo de manejo, y regresará de la página al filtro para hacerlo después de la parte.

Ah, y si no llamas al doFilter aquí, bueno, rompería la cadena. Así que no estás progresando aquí, que en realidad es algo que tal vez desees hacer muy conscientemente. Tal vez su filtro estaba manejando algún tipo de seguridad alrededor del servlet y descubrió que no era satisfactorio. Así que decidió no continuar con el servlet de destino en absoluto. Así que no llames a un doFilter. Tal vez, en cambio, produzca algún tipo de error para el cliente. Eso es muy posible.

Una cosa más sobre estas dos páginas que acabamos de leer. Los filtros y despachadores de solicitudes. Todos comparten el mismo objeto de solicitud. Entonces, si está asignando un bean CDI que está dirigido, el alcance, el alcance de la solicitud, el bean CDI dirigido a la solicitud, entonces todos los filtros en una cadena, todos los servlets que se usaron en reenviar o mostrar el despachador de solicitudes reenviar o el despachador de solicitudes incluyen operaciones, en realidad están mirando el mismo objeto de solicitud. De modo que comparten instantáneamente esa instancia de bean de código de solicitud entre ellos.

Lo cual es bastante útil si desea un filtro para incluir información en una solicitud. Bueno, usa el bean de código de solicitud. Si desea que un servlet coloque información para el próximo servlet al que se reenvió la solicitud para usar esa información. Nuevamente, si eso está en la misma progresión, se está manejando un mismo objeto de solicitud en estas ocasiones, entonces cualquier CDI beans que haya apuntado a ese alcance obviamente será visible para todos los componentes en esa progresión.

Una cosa más. El orden de los filtros se controla a través del archivo web XML. Entonces, debe decidir allí en qué secuencia se aplican los filtros.

Ahora, cambio de tema. Las próximas dos páginas, esta y la siguiente, hablan sobre los servlets asincrónicos y el uso de la API NIO. Son temas muy ligados.

Los servlets asíncronos. Bueno, mira, el protocolo HTTP es inherentemente síncrono. Entonces, realmente, ¿qué es un servlet asíncrono? ¿Cuál es la naturaleza de esto? Porque entonces el cliente crea una solicitud HTTP. El cliente espera algún tipo de respuesta. puede que no sea todo el documento. Podría ser solo el código de estado. Pero aún así, el cliente espera la respuesta del servidor. Entonces, ¿cómo es que puedes tener un servlet asíncrono? ¿Qué significa?

Se trata de mejoras en la utilización de recursos en un servicio. Es un truco para mejorar el rendimiento. La idea es que cuando el cliente llama al servidor, un servidor debe asignar un hilo en el que se procesará la solicitud del cliente. ¿Qué pasa si lleva mucho tiempo procesarlo? Entonces este hilo no se puede usar para otro cliente. Está ocupado con el manejo de esa solicitud en particular.

Entonces, la idea es que lo que puede hacer es hacer que su servlet sea asíncrono. Bueno, el soporte asíncrono es cierto. Y luego, básicamente, cree un nuevo hilo, un nuevo objeto ejecutable, que comience a ejecutar a través de un objeto de contexto asíncrono. Y ese subproceso, ese nuevo objeto ejecutable, estaría allí para llevar a cabo ese progreso lento y prolongado en el que un servidor podría tener que estar involucrado. hora de ejecutar, de llevar a cabo.

Y eventualmente, cuando termine ese procesamiento, informará al contenedor de servlet, eso es todo. Se hace. Ahora puede informar al cliente sobre los resultados de ese procesamiento.

Pero lo más importante, durante ese período de tiempo en el que estaba haciendo ese procesamiento lento aquí, lo estaba haciendo en un hilo separado. Entonces, el subproceso que se usó inicialmente para aceptar la llamada del cliente no estaba ocupado haciendo ese procesamiento. Fue lanzado para manejar alguna otra llamada de cliente en ese momento en particular. Por lo tanto, mejora las capacidades generales del servidor para manejar una cantidad significativa de llamadas simultáneas, aunque algunas de estas llamadas toman mucho tiempo para procesarse.

Muy cerca de esta idea, tenemos otra idea de I/O sin bloqueo. I/O sin bloqueo, de nuevo, se trata de asincronía en los servlets. Por lo tanto, la API de NIO implica que todavía está utilizando un servlet de soporte asíncrono. Puedes usar la anotación para eso. Puede usar un archivo XML web para describirlo como un servlet asíncrono.

La I/O sin bloqueo que usa un servlet está tratando de resolver un problema ligeramente diferente. El ejemplo anterior que vimos, resolviendo un problema de un servidor que tarda mucho tiempo en procesar las cosas, lo hacemos en un hilo separado. Esta vez, estamos resolviendo el problema de un cliente que carga mucha información, posiblemente a través de una conexión que es bastante lenta. Por lo tanto, la capacidad del cliente para cargar datos o descargar datos de nosotros podría verse comprometida. Idea de red tan lenta.

Y lo que estamos tratando de hacer es procesar esa carga o descarga en otro subproceso, sin bloquear el subproceso principal de un contenedor de servlet, liberándolo para manejar llamadas de otros clientes. Entonces, todavía está usando un servlet asíncrono en este caso. Por ejemplo, para manejar la entrada del cliente, obtiene el flujo de entrada del servlet y registra el ReadListener.

ReadListener tiene tres métodos, onDataAvailable, onAllDataRead y onError. La lógica es bastante sencilla. A medida que el cliente carga cosas, las procesa en onDataAvailable. Cuando se reciban todos los datos del cliente, se invocará el método onAllDataRead. Así que sabes ese punto en el tiempo, bueno, eso es todo. Has manejado toda la carga. O si tuvo un problema, ha habido un error, entonces se invocará el método onError.

Si está tratando con cosas que desea descargar del servidor al cliente, simplemente invierta InputStream a ServletOutputStream, ReadListener a WriteListener. Así que es prácticamente un espejo del código. Y luego dos métodos que necesita sobrescribir, onWrite y onError para manejar el contenido que puede ser bastante grande y puede tardar un tiempo en descargarse al cliente, o tal vez la red del cliente no es tan buena, en otro subproceso sin bloquear el subproceso principal del grupo de subprocesos principal en un tiempo de ejecución de servlet.

Ambos son trucos para mejorar el rendimiento. No cambian la naturaleza de la interacción de un cliente con un servlet. El cliente no ve ninguna diferencia. El cliente no sabe si está llamando a un servlet asíncrono o síncrono. El cliente no tiene ni idea. El cliente llama a la URL como de costumbre. El cliente obtiene la respuesta como de costumbre. Se trata de la distribución de subprocesos dentro del contenedor de servlet de Java EE, cómo está sucediendo eso para mejorar el funcionamiento interno de ese contenedor, obviamente por razones de mejora del rendimiento.

Bueno, nos queda una cosa más para discutir aquí, y es la forma en que producimos y manejamos los errores en un contenedor web. Un servlet podría simplemente devolver un código de error al cliente como este. Enviarme error 404, por ejemplo. ¿Por que no? Envíame eso. Página no encontrada.

Bueno, es solo un código de estado. Código de estado HTTP. Si va a lanzar una excepción, eso básicamente producirá uno de los 500 códigos. Entonces, desde la perspectiva del cliente, eso significa que su servlet realmente falló. Pero aún así, puedes si quieres.

Y luego, cuando produce el error, le dice al cliente cuál es el código de estado y cuál es el mensaje de error. En realidad, puede crear un controlador de errores personalizado, y eso se puede registrar en un archivo XML web. Para cada código de error, para cada tipo de excepción, puede registrar una página de controlador o un servlet. Entonces, cuando arroja ese error, el control se pasará a esa página, a ese servlet.

Dentro de ese servlet, obtiene acceso completo a todos los atributos de ese error. Están disponibles en el objeto de solicitud, por lo que puede averiguar cuál fue el código de estado, cuál es la hora de inicio, cuál es el mensaje, qué servlet lo envió. Toda esa información está disponible, por lo que puede producir la salida informativa para el usuario, diciéndole qué salió mal, qué está pasando en su aplicación. Eso es mostrar eso en una página de error designada.

Supongo que eso es todo. Así que hemos visto muchas cosas relacionadas con el tiempo de ejecución del servlet. Pero no se trata solo de servlets, en realidad, porque observamos todo el contenedor web, la forma en que manejamos las llamadas de protocolo HTTP. Por supuesto, observamos la forma en que se escriben los servlets, cómo manejan diferentes métodos HTTP, cómo producen diferentes tipos de contenido, cuál es su ciclo de vida, métodos de ciclo de vida.

Usamos algunos frijoles administrados, frijoles CDI. Utilizamos filtros web, despachadores de solicitudes. Intentamos mejorar el rendimiento con el uso de servlets asincrónicos y API NIO. Y finalmente, analizamos las formas de manejar los errores.

En términos del ejercicio práctico de este capítulo, lo que está a punto de hacer es crear una aplicación web que buscará y mostrará productos. Tendrá solo un formulario de búsqueda HTML simple donde se supone que su usuario debe ingresar criterios y ejecutar una consulta. Serán manejados por un servlet, y luego el servlet generará algún tipo de salida, que será la lista de productos como resultado de esa consulta.