# REST

Echemos un vistazo a las formas de implementar servicios REST con la API JAX-RS. En este capítulo, podemos tratar de comprender primero la forma en que funcionan los servicios REST (convenciones de transferencia de estado representacional) y luego implementar controladores Java para los componentes REST, y luego también ver cómo invocamos los servicios REST desde el cliente.

Echemos un vistazo a las convenciones y recursos del servicio REST. DESCANSO: en primer lugar, no es un estándar. Es solo una convención de codificación. Entonces, puede encontrar servicios REST que realmente no siguen ese patrón, pero, bueno, la convención dice que deberían hacerlo. Se trata de manejar diferentes tipos de recursos en un servidor. El recurso es un objeto del lado del servidor. Y luego lo asignamos a la URL y manejamos el acceso a ese objeto a través de una variedad de métodos HTTP: GET, POST, PUT, DELETE.

Supongamos que el Recurso representa algún tipo de objeto comercial y queremos realizar operaciones como consultar, insertar, actualizar, eliminar alrededor de ese objeto. Asignamos estas operaciones a métodos HTTP. Esperamos que el método GET realice la consulta. Esperamos que el método POST cree un nuevo tipo de objeto, una nueva instancia de objeto. Esperamos que el método PUT lo actualice, DELETE para eliminarlo, para eliminarlo. También son posibles otras operaciones.

Pero debido a que no es un estándar, realmente no podemos garantizar que algunos servicios REST, escritos por otras personas, siempre se adhieran a esa convención en particular. Esperamos que lo sean, pero de nuevo. A diferencia de los servicios SOAP, que están codificados muy estrictamente, un estándar muy rígido (se trata de archivos y esquemas WSDL), los servicios REST no lo son.

No están obligados a proporcionar ningún descriptor formal. No están obligados a seguir esa convención, aunque, bueno, realmente deberían hacerlo, ¿sí? Bien, es solo una colección de diferentes recursos a los que se accede a través del protocolo HTTP.

Ahora, echemos un vistazo más de cerca al modelo de comunicación REST real. Los datos que transporta el protocolo REST pueden ser cualquier cosa. Puede ser XML. Puede ser JSON. Puede ser el formato que quieras. No existe ningún requisito para que la información que intercambiamos entre un cliente y el servicio REST siga un formato en particular.

JSON es la notación de objetos de JavaScript. Por lo general, lo utilizan las aplicaciones de JavaScript. Y XML, bueno, obviamente, eso es lo que usan otros servicios, como los servicios SOAP. Pero no hay restricciones en cuanto al formato exacto de información que se supone que debe usar el servicio REST. Lo que se supone que debe hacer, se supone que debe reaccionar a los métodos HTTP: GET, POST, PUT, DELETE, HEAD y OPTIONS, reaccionar a estos métodos y proporcionar el manejo de un recurso en particular.

Muy bien, el cliente típico que invocará el servicio REST es probablemente una especie de cliente de JavaScript en un navegador, o tal vez una aplicación móvil. Pero podría llamarse desde Java, ya sabes, desde cualquier lugar que desees.

Cómo se implementan los servicios REST utilizando la API JAX-RS: bueno, antes que nada, debe crear un componente de aplicación REST y asignarlo a la URL HTTP. Este es el primer ejemplo: la ruta de la aplicación asigna el componente de la aplicación a la URL. Volveré a esta clase de aplicación en un momento. Hablaremos de ello en un momento.

Pero luego quiero resaltar este registro de Recursos. Creas una clase que representa tu recurso. Por ejemplo, digamos que el servicio REST debe manejar algo sobre un Producto, ¿sí? Este es un recurso de producto.

El Recurso aprovisionará el manejo de las operaciones. GET, POST, PULL, DELETE, eso aún está por venir. Eso no está en este PowerPoint. Agregaremos ese código un poco más tarde. Y un recurso también es un mapa de la URL, así que aquí está: mapa de los productos de la URL.

Ahora, cómo se registra el recurso con la aplicación: volvamos a la clase de la aplicación. Necesitamos tomar el recurso y agregarlo a la lista de recursos admitidos para esa aplicación. Entonces, anular el método llamado obtener clases, ¿verdad?

Hay otro método en la aplicación que podría haber usado y que registra singletons. Así que hay clases de obtención, y luego también hay un registro de singletons, así que un par de métodos, ¿sí? Pero en este ejemplo, le muestra el registro de la clase en lugar de un singleton.

La URL resultante será la URL base más la URL del recurso. Entonces será la URL completa que apunta a ese Recurso. Un Recurso puede entonces registrar más Sub-Recursos. Entonces, por ejemplo, un Producto tiene un Sub-Recurso, que es Descuento. Eso simplemente se agrega a esa URL base como un subrecurso adicional.

Para mapear Recursos y Sub-Recursos, usa la notación llamada Ruta. Para mapear la URL base, utiliza la notación denominada Ruta de la aplicación. Entonces, la URL base es para la aplicación y luego las rutas son para los recursos.

¿Qué recurso podría ser? Bueno, podría ser un Enterprise JavaBean. Podría ser un frijol CDI.

Podría ser simplemente un POJO. Bueno, esto es solo un POJO. Es solo un objeto Java normal. No hay nada especial al respecto.

Si se trata de un bean sin estado singleton, se registra con un método diferente en la clase de configuración de la aplicación. Pero de lo contrario, solo se trata de obtener la clase y registrarla.

De acuerdo, y mapearlos a URL: mapear recursos a URL es un proceso bastante interesante, porque aparentemente, también puede parametrizar estas URL. Y este es un ejemplo interesante. Así que ahora tenemos... por cierto, ese es el primer ejemplo de la creación real de un recurso que maneja un método HTTP específico. Ves la anotación GET, ¿sí?

Por lo tanto, está diciendo que este recurso, recurso REST del producto, reaccionará cuando el cliente lo llame a través del método HTTP GET, ¿de acuerdo? Y esperará que la URL contenga algunos valores extra interesantes. Por ejemplo, aquí hay un código de producto.

Ha notado que Path contiene una expresión regular que describe la forma de ese parámetro, ¿lo ve? Y esto es lo que esperas, que el código tenga una forma particular. También podría usarse para describir cosas como, por ejemplo, valor desde y hasta, como un rango de valores, ¿no? Entonces, los parámetros podrían contener diferentes tipos de partes, si lo desea.

Ahora, cada parámetro tendría un nombre, así, y luego la anotación PathParam se referiría a ese nombre. Así que eso es bastante sencillo, ¿sí? El nombre asignado a la anotación PathParam dentro de un método. Y así es como su código Java dentro de ese método obtiene acceso a los parámetros enviados a través del método GET aquí. Son solo parte de la URL.

Otra cosa interesante que sucede en este ejemplo es este par de anotaciones, que se llama Produces y Consumes. Las anotaciones Produce y Consumes se utilizan para describir qué tipo de estructuras de datos: ¿con qué tipo de datos trabajaría este servicio REST en particular? En este caso particular, su aplicación es de tipo JSON. Pero como digo, podría ser XML, podría ser HTML, podría ser texto sin formato, cualquier cosa.

REST Resource podría admitir varias operaciones diferentes, no solo GET. Entonces, este ejemplo muestra más operaciones: GET, POST, PUT, DELETE. Comencemos con OBTENER.

Convencionalmente, recuerde, ese no es un estándar estricto, sino una convención general, espera el método GET, que no tiene parámetros, GET sin parámetros, espera que le devuelva una lista de componentes del tipo de ese recurso.. Así que lista de Productos, en este caso.

Si el método GET contiene parámetros, por ejemplo, contiene un parámetro de ID de producto, entonces espera que no devuelva una lista de productos, sino un solo producto. Pero de cualquier manera, se mire como se mire, generalmente se dice que se usa para leer datos. El método GET está ahí para admitir operaciones de lectura.

Echemos un vistazo a POST. POST podría usarse para crear un nuevo objeto: crear una nueva entidad. PUT: la diferencia entre POST y PUT, verá, es que PUT contiene una identificación del objeto: la identidad como parámetro de ruta. Y eso significa que probablemente sea más adecSin embargo, debo decir que hay un comentario en la parte inferior de la página que a veces, PUT o POST pueden usarse indistintamente por un servicio REST. Recuerde, ausencia de estándar, es solo una convención. Entonces, hay servicios REST que pueden usar POST para actualizar y PUT para crear, y no al revés. Eso depende.

Supongo que la razón es esta. Si está creando una nueva entidad y no está generando la ID a partir de la secuencia, puede designar la ID como parámetro de ruta. Y si tiene medios para identificar qué registro actualizar sin pasar un parámetro de ID, bueno, supongo, entonces, el PUT podría usarse para crear en lugar de actualizar. No sé. POST podría usarse para actualizar en ese caso.

Bueno, depende de ti, de verdad. Pero en términos generales, normalmente usaría POST para crear y PUT para actualizar. DELETE para eliminar, esperaría-- Operación DELETE para aceptar la identidad del registro como un parámetro de ruta para identificar qué registro, qué objeto desea eliminar.

Se supone que el método HEAD HTTP convencionalmente hace exactamente lo mismo que GET, excepto que no devuelve un objeto real. Solo devuelve encabezados. El caso de uso típico para HEAD es comprobar si la operación GET recibida anteriormente (el contenido de GET) sigue siendo válida. Entonces puede, por ejemplo, al llamar al método HEAD, verificar si este contenido ha cambiado, como, ya sabes, si aún se puede almacenar en caché, si los encabezados siguen siendo los mismos, entonces es posible que no tenga que descargar todo el contenido. de nuevo a través del método GET.

El método OPTIONS generalmente se usa para devolver información sobre otros métodos admitidos por ese recurso REST en particular. Entonces, si su cliente desea averiguar si el Recurso es compatible con GET, POST, PUT, DELETE, de cualquier manera, llamar al método OPTIONS devolverá esa información.

La API de JAX-RS proporciona una implementación predeterminada para los métodos HEAD y OPTION. Realmente no es necesario escribirlos. El JAX-RS ya le brinda estos métodos listos para usar. Pero si quiere, por ejemplo, poner algo de lógica adicional en el método HEAD, ciertamente puede hacerlo.

Hasta ahora, tenemos servicios REST que producían y consumían solo un tipo de contenido. Pero pueden producir y consumir diferentes tipos. Y puedes mezclarlos en cualquier forma y forma que quieras. Entonces, en este ejemplo, verá que, en general, esperamos que este recurso de servicio produzca objetos JSON. Pero luego, algún método podría producir XML, y otro método podría producir texto sin formato, y luego otro podría consumir XML o JSON.

Por lo tanto, se trata de cuán flexible desea que sea su servicio. ¿Quieres que tu servicio permita esta multitud de formatos diferentes de información? ¿Está bien que un cliente le envíe objetos XML y objetos JSON? Supongo que si desea admitir diferentes tipos de clientes, eso podría tener sentido. Correcto, eso se logra con las anotaciones Produce y Consumes.

Un poco más sobre los parámetros. Hasta ahora, hemos visto ejemplos que se usaron en este primer estilo de parámetro llamado Parámetro de ruta. Eso estaba en las páginas anteriores. Así que barra y luego el parámetro.

Alternativamente, podría usar otros estilos de parámetros. Un parámetro de consulta como un par de nombre y valor, como puede ver. Parámetro Matrix: bueno, la diferencia es que Matrix usa este punto y coma en lugar de un signo de interrogación como separador de parámetros. Los parámetros podrían no ser parte de la URL, sino cookies o encabezados HTTP. Así que podrías usarlos para la parametrización. Un elemento en el formulario HTML, application/x-www-form-urlencoded, que también podría ser el parámetro. Eso es FormParam.

Bien, eso es lo que normalmente usaría en sus métodos de manejo de recursos. Ah, también, si el parámetro falta, puede predeterminar su valor. Así que esa es esta anotación justo en la esquina de la pantalla, por defecto el valor del parámetro. Se puede usar con cualquier estilo de parámetro: ruta, consulta, matriz, cualquiera.

Correcto, ¿recuerdas Bean Validation? Hablamos sobre esta API de validación de beans en el capítulo que trata sobre la persistencia de Java. Y aparentemente, mencionamos allí que Beam Validation se puede usar no solo en la API de persistencia de Java, sino en todas partes, en cualquier lugar, en realidad, incluidos los servicios REST, ¿ves? Esas son todas las mismas restricciones de validación de Bean: NotNull, sí. Era un min max decimal, ¿sí?

Y, por lo general, solo puede usar esta anotación @Valid, anotar un parámetro y validar cualquier usuario que se lo haya enviado. Así que esa es exactamente la misma API de validación de Bean, sin ninguna diferencia. Utiliza el mismo paquete de recursos para las propiedades del mensaje de validación, para especificar mensajes de error. Usa las mismas anotaciones que usó antes en la parte JPA del curso. Es exactamente lo mismo.

Bien, ahora, si quisiéramos producir errores de servicio web desde dentro del servicio REST, supongo que es bastante simple así. Debe enviar al cliente un código de estado HTTP relevante.

Hay varias formas en las que puedes hacerlo. Entonces, cualquier cosa del rango de 400 o 500 códigos de error HTTP básicamente indicará errores al cliente. Así que suponga que, en este caso particular, podría usar un objeto ResponseBuilder para establecer el estado con un código de respuesta relevante; bueno, NOT\_FOUND indicará un error.

Alternativamente, puede simplemente lanzar la excepción. Este ejemplo de lanzamiento de WebApplicationException con exactamente el mismo error NOT\_FOUND, por lo que pretende que no se encuentra el recurso al que el cliente intenta acceder. Sí, ese es tu código de error, 404. Bueno, podría ser cualquier otro código de estado que quieras.

Alternativamente, podría lanzar una de las excepciones que en realidad son subclases de WebApplicationException. NotFound es en realidad una subclase de WebApplicationException con código de estado precodificado. Así que no tienes que analizarlo como un argumento. Es solo que, para cada código de estado HTTP, ya hay una subclase disponible para WebApplicationException que puede lanzar.

Si desea lanzar cualquier otra excepción personalizada suya, deberá crear un mapeador, un mapeador que mapee su excepción al código HTTP que desea devolver al cliente. Las notas del curso contienen el ejemplo.

Lo siguiente interesante que vamos a ver es la capacidad del servicio REST para manejar nuestras solicitudes de manera asíncrona. Recuerde, el protocolo HTTP sigue siendo sincrónico. No hay nada que puedas hacer al respecto. Se trata de una optimización de la utilización de recursos en un nivel de servidor.

Entonces, en este caso particular, está diseñando el servidor: un servicio web, un servicio REST, que usa este parámetro interesante, AsyncResponse, con anotación suspendida. Y lo que nos permite hacer es crear un controlador para la respuesta asincrónica, y nos permite definir una lógica de qué quiere hacer con esa respuesta como otro hilo que hará el manejo, así que esa es la lógica real. - pero también asocie ese manejo con un tiempo fuera.

Puede establecer la duración del tiempo de espera, en este caso particular, 20 segundos, y luego puede configurar el controlador de tiempo de espera como una pieza de código que desea invocar si se produce el tiempo de espera, antes de que realmente lograr producir ese resultado de servicio en particular. Así que te has dado 20 segundos para hacerlo. Y el manejo de la llamada real está en un nuevo objeto ejecutable. Así que eso es básicamente un hilo nuevo.

Si se trata de un servicio REST que se implementa como un Enterprise JavaBean, en realidad es bastante fácil de hacer, porque los Enterprise Java Beans ya son capaces de manejar llamadas asincrónicas. Así que todo lo que necesitas hacer es poner una anotación asíncrona, y eso es todo, de verdad. No es necesario que crees este hilo nuevo.

No, solo parámetro suspendido. Es lo mismo que en el ejemplo anterior. Y simplemente marque el método como asíncrono, y eso es todo. Escriba la lógica de negocios que realiza el manejo de la llamada de forma asíncrona. Por lo tanto, es mucho más fácil de hacer si está utilizando Enterprise Java Bean, porque está utilizando sus funciones de manejo asíncrono, su capacidad de manejar solicitudes de manera asíncrona.

Cómo invocar el servicio REST desde el cliente: ahora, esta página en particular le muestra cómo puede llamar al servicio REST desde el cliente JavaScript. JavaScript es un lenguaje de programación diferente. No está cubierto en este curso. Está cubierto en otro curso. JavaScript y HTML5, desarrollo de aplicaciones web: tenemos el curso para ello, que detalla exactamente lo que hace en JavaScript aquí.

Sin embargo, debido a que los servicios REST a menudo se invocan desde JavaScript, es una razón para mostrarle al menos el ejemplo de cómo se hace, tal vez sin una explicación elaborada de un capítulo, que es donde puede encontrar información en ese curso. Pero al menos una idea de lo que está pasando en el cliente de JavaScript.

Los clientes de JavaScript pueden manejar estilos sincrónicos o asincrónicos de comunicaciones HTTP. En esta etapa, se pondrá un poco extraño. Para hacer esa comunicación con un servicio REST, debe crear un objeto XMLHttpRequest.

Bueno, tan pronto como piensas en JavaScript, las cosas no son como se llaman. Un objeto XMLHttpRequest no tiene que representar XML. De hecho, puede representar cualquier dato que desee. Podría ser JSON, podría ser XML, podría ser texto sin formato. No importa. Sé que se llama XML, pero eso no significa que en realidad sea XML.

Entonces, en este caso particular, está creando un XMLHttpRequest, y luego lo que hace es que... bueno, obtiene algunos parámetros. Supongo que su página tiene algún valor de ID de producto, por lo que es un elemento en su documento HTML, supongo. Usted apunta a la URL de su servicio REST, por lo que el paradero de su servicio. Esa es la URL que contiene este ID de producto al final.

Y luego lo abre con una solicitud HTTP, ábralo con el método GET. Puede ser una llamada síncrona o asíncrona. Este ejemplo particular es asíncrono. Se está registrando en la función readystatechange, lo que básicamente significa que está buscando una devolución de llamada en un punto en el que se completará el manejo de esa llamada.

Tan pronto como cambian, estás rastreando si terminaste esa llamada. El readyState es igual a 4. Hay varios readyStates en JavaScript. Cuando envías la solicitud, cuando empiezas a recibir... cuando subes información al servidor, entonces empiezas a recibir respuestas y finalmente, obtienes la respuesta. Así que readystatechange 4 es que tienes la fase de respuesta. Llegaste al final.

Y luego verifica el código de estado. Bueno, eso es solo un estado HTTP. Supongo que si es como 400, entonces eso es un error. Si es 200, todo está bien.

Y mira lo que haré a continuación. Espero que este servicio devuelva un objeto JSON. Así que estoy analizando el valor JSON devuelto, y lo estoy usando de alguna manera dentro de la página, bueno, falta cómo se usa exactamente en este ejemplo en particular. Pero esta es la interacción.

Así que estás diciendo que termino con un código 200, en cuyo caso usaremos el contenido producido por el servicio REST, o terminé con algún otro código, que supongo que es un error. Como digo, hay más información al respecto en el curso JavaScript y HTML5, Desarrollo de aplicaciones web, que le recomiendo encarecidamente que también tome si está interesado en escribir aplicaciones usando JavaScript.

¿Qué pasa con el cliente Java? OK, bueno, esa es la pieza de código de Java que llama al servicio REST. La API que está utilizando para esa invocación se llama Jersey, JAX-RS Jersey. Está básicamente en la biblioteca de implementación de referencia detrás de la API JAX-RS.

Bien, es bastante sencillo. Primero, debe apuntar a su servicio REST, la URL base. Cree un nuevo objeto Cliente. Construya cualquier URL adjunta a la URL base, bueno, la ruta a su Recurso. Eso es Recursos de productos, es una especie de ID de producto adjunto a eso.

Entonces estás registrando con el objetivo web qué tipo de respuesta se supone que debes obtener. Por lo que está diciendo, creo que recuperaré un objeto Producto codificado como JSON-- Objeto Producto codificado como JSON. Y para recuperarlo, necesito invocar el servicio. Así que termina construyendo el método GET: buildGet, POST, PUT, DELETE, cualquier método HTTP que desee usar e invoque ese servicio.

El servicio se invoca desde el objeto Respuesta. Usted lee ese Producto. Por lo tanto, el objeto JSON se convertirá en objeto Producto para usted. Ahí tienes Cierre el cliente cuando termine con una interacción, cuando termine con esa interacción.

Este es un ejemplo ligeramente diferente de un cliente Java que llama a un servicio REST. Y la diferencia es que esta vez es una llamada asíncrona. Entonces agrega estas operaciones asíncronas. Inicializas el cliente en un objetivo web exactamente de la misma manera. Pero antes de realizar la solicitud, asegúrese de que sea una solicitud asíncrona. Así que agregaste ese bit.

Y luego registra InvocaciónCallbackListener. InvocaciónCallbackListener tiene dos métodos, completado y fallido. Si realmente obtiene una respuesta del servicio REST, se invocará el método completado. Si el servicio REST devuelve un error, se devolverá el método fallido, se invocará por usted. Por lo tanto, podría poner lógica en estos métodos sobre cómo le gustaría manejar la respuesta correcta o con errores del servicio REST.

Supongamos que es algo bastante sencillo. Entonces, antes que nada, recuerde, usted forma la aplicación REST, mapea a la URL base, luego forma una cantidad de recursos que reaccionan a GET, POST, PUT, DELETE, cualquier operación, decide cómo les pasa parámetros, decide qué tipo de tipos de datos que consumen o producen. Dicho esto, agregue un poco de lógica comercial y estará listo para comenzar. Y luego decida, ¿qué tipo de cliente le gustaría tener? Tal vez JavaScript, tal vez Java.

Y también, hemos mencionado las capacidades asíncronas de los servicios REST y sus clientes. Y el patrón de todas las llamadas asincrónicas, probablemente ya lo haya notado, es mediante el registro de oyentes que buscan que la llamada se complete, o tal vez con errores.

¿Qué tal una práctica? En este ejercicio, creará un nuevo servicio REST que realiza una comprobación muy sencilla. Mira si el descuento para el Producto está disponible o no.

Luego, escribirá dos tipos de clientes para este servicio REST. Uno está en JavaScript y formará parte de su aplicación basada en navegador, y el otro está en Java, solo para ver cómo podríamos llamar al servicio REST desde el cliente Java también.