Desarrollo avanzado de aplicaciones II

Tema Nº6:Transacciones con Spring Data JPA.

Indicador de logro Nº6:

Gestiona las transacciones con el componente Spring Transaction para operaciones de entidades con cabecera y detalle.

**TEMA 01 Teoría de los**

Imagen que contiene Icono

Descripción generada automáticamente

**TEMA Nº6:**

**Transacciones con Spring Data JPA.**

**Subtema 6.1:**

Utilizando el gestor transaccional (JTA) con Spring Transaction.

**Ejemplos:**

Una de las razones más convincentes para utilizar el marco Spring es el apoyo de transacción global. El marco de Spring proporciona una abstracción coherente para la gestión de transacciones que ofrece los siguientes beneficios:

* Proporciona un modelo de programación consistente a través de las API de transacciones diferentes, tales como JTA, JDBC, Hibernate, JPA, y JDO.
* Apoya la gestión de transacciones declarativa.
* Proporciona una sencilla API para la gestión de transacciones programática de una serie de API de transacciones complejas tales como JTA.
* Se integra muy bien con las abstracciones de acceso a datos.

Tradicionalmente, los desarrolladores de J2EE han tenido dos opciones para la gestión de transacciones: las transacciones globales o locales. Las transacciones globales son gestionadas por el servidor de aplicaciones, utilizando el API de transacciones Java (JTA). Las transacciones locales son recursos específicos: el ejemplo más común sería una transacción asociada con una conexión JDBC. Esta elección tiene profundas implicaciones. Por ejemplo, las transacciones globales proporcionan la capacidad de trabajar con múltiples recursos transaccionales (bases de datos relacionales y, normalmente, las colas de mensajes). Con las transacciones locales, el servidor de aplicaciones no está involucrado en la gestión de transacciones y no puede ayudar a asegurar la corrección a través de múltiples recursos.

**Transacciones globales**

Las transacciones globales tienen una desventaja importante, en ese código tiene que usar JTA, y JTA es una API difícil de utilizar (en parte debido a su modelo de excepción). Además, un UserTransaction JTA normalmente tiene que proceder de JNDI. Por lo tanto, tenemos que utilizar tanto JNDI como JTA. Obviamente, todo uso de las transacciones globales limita la reutilización de código de la aplicación, ya que JTA normalmente sólo está disponible en un entorno de servidor de aplicaciones. Anteriormente, la mejor forma de utilizar las transacciones globales fue a través de EJB CMT (transacción administrada por contenedor).

CMT es una forma de gestión de transacciones declarativa (a diferencia de la gestión de transacciones programática). EJB CMT elimina la necesidad de transacción relacionados con búsquedas JNDI aunque, por supuesto, el uso de EJB requiere el uso de JNDI. Se elimina la mayor parte de la necesidad (aunque no del todo) de escribir código Java para el control de las transacciones. El inconveniente importante es que la CMT está ligada a JTA y un entorno de servidor de aplicaciones. Además, sólo está disponible si se opta por aplicar la lógica de negocio en EJB, o al menos detrás de una fachada de EJB transaccional. Lo negativo en relación con EJB es que, en general, no se trata de una propuesta atractiva, especialmente en la cara de las alternativas de peso para la gestión de transacciones declarativa.

**Transacciones locales**

Las transacciones locales pueden ser más fáciles de usar, pero tiene desventajas importantes: no pueden trabajar a través de múltiples recursos transaccionales. Por ejemplo, el código que gestiona las transacciones utilizando una conexión de JDBC no se puede ejecutar en una transacción JTA global. Otra desventaja es que las transacciones locales tienden a ser invasoras en el modelo de programación.

Spring resuelve estos problemas. Permite a los desarrolladores de aplicaciones para utilizar un modelo de programación coherente en cualquier entorno. Debe escribir el código una vez, y pueden beneficiarse de diferentes estrategias de manejo de transacciones en diferentes entornos. El marco de Spring proporciona la gestión de transacciones declarativas y programáticas. La gestión de transacciones declarativa es la preferida por la mayoría de los usuarios y se recomienda en la mayoría de los casos.

Con la gestión de transacciones programática, los desarrolladores trabajan con la abstracción de la transacción Spring Framework, que puede correr sobre cualquier infraestructura de operación subyacente. Con el modelo de declaración preferido, los desarrolladores suelen escribir poco o ningún código relacionado con la gestión de transacciones y, por lo tanto, no dependen de la transacción de Spring Framework.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

**Subtema 6.2:**

Comprender las funciones básicas de Spring Transaction.

**Ejemplos:**

* Aislamiento: el grado de aislamiento de esta transacción sobre otras transacciones. Por ejemplo, ¿esta transacción puede verse comprometida por la escritura de otras transacciones?
* Propagación: en general, todo código que se ejecuta dentro de un ámbito de transacción se ejecutará en esa transacción. Sin embargo, hay varias opciones que especifican el comportamiento si se ejecuta un método de transacción cuando el contexto de la transacción ya existe: por ejemplo, sólo tiene que seguir corriendo en la operación existente (el caso más común), o la suspensión de la operación existente y la creación de una nueva transacción. Spring ofrece todas las opciones de transacción de propagación familiares de EJB CMT.
* Tiempo de espera: ¿cuánto tiempo tiene la transacción para ejecutarse antes del tiempo de espera (y automáticamente se deshace de la infraestructura de transacción subyacente).
* Estado de sólo lectura: una transacción de lectura única no puede modificar ningún dato. Las transacciones de sólo lectura pueden ser una optimización de utilidad en algunos casos (como cuando se utiliza Hibernate).

**Transaction Propagation**

La propagación define el límite de las transacciones de nuestra lógica empresarial. Spring logra iniciar y pausar una transacción de acuerdo con nuestra configuración de propagación.

REQUERIDO es la propagación predeterminada. Spring comprueba si hay una transacción activa y, si no existe, crea una nueva. De lo contrario, la lógica empresarial se agrega a la transacción actualmente activa:

Texto

Descripción generada automáticamente con confianza media

Para SOPORTES, Spring primero verifica si existe una transacción activa. Si existe una transacción, se utilizará la transacción existente. Si no hay una transacción, se ejecuta de forma no transaccional:

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

Cuando la propagación sea OBLIGATORIA, si hay una transacción activa, se utilizará. Si no hay una transacción activa, Spring lanza una excepción:

Texto

Descripción generada automáticamente

**Transaction Isolation**

El aislamiento es una de las propiedades comunes del ACID: atomicidad, consistencia, aislamiento y durabilidad. El aislamiento describe cómo los cambios aplicados por transacciones concurrentes son visibles entre sí.

Cada nivel de aislamiento evita cero o más efectos secundarios de simultaneidad en una transacción:

* Lectura sucia: lee el cambio no confirmado de una transacción concurrente
* Lectura no repetible: obtenga un valor diferente al volver a leer una fila si una transacción concurrente actualiza la misma fila y se confirma
* Lectura fantasma: obtenga diferentes filas después de volver a ejecutar una consulta de rango si otra transacción agrega o elimina algunas filas en el rango y confirma

READ\_UNCOMMITTED es el nivel de aislamiento más bajo y permite el acceso más concurrente.

Como resultado, sufre los tres efectos secundarios de concurrencia mencionados. Una transacción con este aislamiento lee datos no confirmados de otras transacciones concurrentes. Además, pueden producirse lecturas fantasmas y no repetibles. Por lo tanto, podemos obtener un resultado diferente al volver a leer una fila o al volver a ejecutar una consulta de rango.

Podemos establecer el nivel de aislamiento para un método o clase:

Texto

Descripción generada automáticamente

El segundo nivel de aislamiento, READ\_COMMITTED, evita las lecturas sucias.

El resto de los efectos secundarios de la concurrencia aún podrían ocurrir. Por lo tanto, los cambios no confirmados en transacciones simultáneas no tienen ningún impacto en nosotros, pero si una transacción confirma sus cambios, nuestro resultado podría cambiar si volvemos a consultar.

Aquí establecemos el nivel de aislamiento:

Texto

Descripción generada automáticamente

El tercer nivel de aislamiento, REPEATABLE\_READ, evita lecturas sucias y no repetibles. Por lo tanto, no nos afectan los cambios no comprometidos en las transacciones simultáneas.

Además, cuando volvemos a consultar una fila, no obtenemos un resultado diferente. Sin embargo, al volver a ejecutar las consultas de rango, es posible que obtengamos filas recién agregadas o eliminadas.

Además, es el nivel más bajo requerido para evitar la pérdida de actualización. La actualización perdida ocurre cuando dos o más transacciones simultáneas leen y actualizan la misma fila. REPEATABLE\_READ no permite el acceso simultáneo a una fila en absoluto. Por lo tanto, la actualización perdida no puede ocurrir.

A continuación, se explica cómo establecer el nivel de aislamiento para un método:

Texto

Descripción generada automáticamente

SERIALIZABLE es el nivel más alto de aislamiento. Evita todos los efectos secundarios de concurrencia mencionados, pero puede conducir a la tasa de acceso concurrente más baja porque ejecuta llamadas concurrentes secuencialmente.

En otras palabras, la ejecución concurrente de un grupo de transacciones serializables tiene el mismo resultado que ejecutarlas en serie.

Ahora veamos cómo establecer SERIALIZABLE como nivel de aislamiento:

Texto

Descripción generada automáticamente con confianza media

**Subtema 6.3:**

Realizar prueba de confirmación de transacción

**Ejemplos:**

Spring crea un proxy o manipula el código de bytes de la clase para administrar la creación, confirmación y reversión de la transacción. En el caso de un proxy, Spring ignora @Transactional en las llamadas a métodos internos.

En pocas palabras, si tenemos un método como callMethod y lo marcamos como @Transactional, Spring ajustará algún código de gestión de transacciones alrededor del método invocation @ Transactional llamado:

Imagen de la pantalla de un celular de un mensaje en letras blancas

Descripción generada automáticamente con confianza baja

**Subtema 6.4:**

Realizar prueba para deshacer una transacción.

**Ejemplos:**

La anotación @Transactional son los metadatos que especifican la semántica de las transacciones en un método. Tenemos dos formas de revertir una transacción: declarativa y programática.

En el enfoque declarativo, anotamos los métodos con la anotación @Transactional. La anotación @Transactional utiliza los atributos rollbackFor o rollbackForClassName para revertir las transacciones, y los atributos noRollbackFor o noRollbackForClassName para evitar la reversión de las excepciones enumeradas.

El comportamiento de reversión predeterminado en el enfoque declarativo revertirá las excepciones en tiempo de ejecución.

Veamos un ejemplo simple que usa el enfoque declarativo para revertir una transacción para excepciones o errores en tiempo de ejecución:

Imagen que contiene Texto

Descripción generada automáticamente

A continuación, usaremos el enfoque declarativo para revertir una transacción para las excepciones marcadas enumeradas. La reversión en nuestro ejemplo está en SQLException:

Texto

Descripción generada automáticamente con confianza media

Veamos un uso simple del atributo noRollbackFor en el enfoque declarativo para evitar la reversión de la transacción para la excepción enumerada:

Captura de pantalla de un celular

Descripción generada automáticamente

En el enfoque programático, revertimos las transacciones mediante TransactionAspectSupport:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**Actividad:**

a) CUESTIONARIO TÉCNICO

* ¿Para qué sirve la anotación @Transactional?
* ¿Qué tipos de transacciones existen?
* ¿En qué ocasiones se utiliza la anotación @Transactional?
* Cree un nuevo proyecto con Spring Boot y realice dos operaciones utilizando @Transactional.
* En el mismo proyecto agregue la propiedad roolback en cada una de las operaciones.

b) CONCLUSIONES DE LA EXPERIENCIA

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_