BOLETÍN 2

APLICACIONES DISTRIBUIDAS

ECPaquita74

Ejercicio 1

Para resolver este ejercicio añadimos las dependencias necesarias para el uso de MongoDB y usamos la anotación `@BsontId` al `idActividad` de la clase `Actividad`. Además, tenemos dos clases nuevas dentro del paquete aadd.mongo.dao: *`MongoCodecDAO`* y *`ActividadCodecDAO`*. Haremos uso de un codec que nos permite mapear una clase Java a los documentos recuperados de una colección de la base de datos, la primera clase se dedica a un DAO genérico, usando un codec por defecto. En la segunda clase, que hereda de ésta, creamos dos métodos, uno para registrar las actividades en la base de datos (de nombre ecpaquita74 en la collection 'actividad'), que lanza una excepción si existe ya una actividad registrada con ese nombre, y otro para devolver una lista de todas las actividades que están registradas. También tenemos dos clases sin uso del codec, pero no se usan para el ejercicio.

De esta forma, para las consultas, tendremos que invocar al método `findActividades` de `ActividadCodecDAO`, que nos devuelve un array de objetos de tipo Actividad desde donde poder acceder más fácilmente a los atributos que necesitamos comparar desde el `ServletGestionarCalendario`, ya que tratamos directamente con objetos `Actividad`, por ejemplo, para consultar las actividades por meses y año.

El método de registro ayuda a que podamos hacer la comprobación de identificadores duplicados en la base de datos, lanzando una *`MongoWriteException`* si hay ya un objeto con ese _id en la collection, que se trata desde el servlet para que se comunique la información del registro en cada caso al usuario.

Para hacer la gestión de reservas, añadimos otra función en la clase `ActividadCodecDAO`, la llamaremos `updatePazas`. En ella, con un Filter y un Update, hacemos la búsqueda de la actividad que se le pasa como parámetro en Mongo y actualizamos el valor de las plazas a 1 menos ($`inc`: -1).

Ejercicio 2

Añadimos una clase en la carpeta de test del proyecto, que llamaremos `TestConsultasActividad` para probar las dos consultas que se nos piden. Para ello, borraremos las actividades que había ya registradas en la base de datos para certificar el buen funcionamiento de los tests que vamos a crear (buscando obtener el resultado que describo en los comentarios del código de cada uno). Para conseguir esto, he creado un test (`testIntroducirActividades`) en esta clase que introduce 8 actividades en la base de datos y que **debe ejecutarse antes**. Estas actividades nos ayudarán a comprobar el funcionamiento correcto de cada uno de los filtros que aplicaremos para las 2 consultas.

Para la primera consulta que se nos pide, tenemos ahora el método `getActividadesDisponibles` en la clase `ActividadCodecDAO`, que probaremos en el test de la clase anterior `testActividadesDisponibles`. El método se encarga de hacer un filtro de las actividades de la collection por fecha y plazas, donde el primer parámetro debe ser

mayor que la fecha actual (`LocalDate.now`) y el segundo mayor que 0. De esta forma, rescatamos todos los elementos que coincidan con un `find` y los metemos en una lista que devolvemos. Si esta lista está vacía, se devuelve null. El test de esta función tiene que devolver un valor distinto de null (ya que sí hay actividades disponibles entre las introducidas previamente), además, imprimimos en consola una dupla de id y nombre de aquellas actividades que se hayan recogido.

Para la segunda consulta, usaremos la clase `ActividadDAO`, donde creamos el método `getTotalPlazasMesTipo`. Aquí creamos dos variables para el inicio y el final del año 2022 en un objeto tipo `LocalDate`. Este método crea una lista de documents resultado de la agregación de la colección donde las actividades cumplen (*matchStage*) tener una fecha que se encuentre entre el inicio y final del año 2022 previamente declarados, que agrupará (*groupStage*) por mes (valor `$month` dentro de la fecha) y tipo de actividad, y hace un sumatorio de plazas (*accumulator*) por cada grupo, además de incluir con `push` (otro *accumulator*) un array con todos los nombres de las actividades que forman parte del mismo. En el test que prueba esta consulta (`testAggregatePlazasMesTipo`) sólo tenemos que ejecutar el método anterior, que es un void, ya que éste también se encarga de hacer la impresión por consola con un `forEach` de los documentos de la agregación, aplicándoles el método `toJson`.

Ejercicio 3

Para modelar un objeto de tipo reserva asociado a una actividad tendremos que incluir un id único de la reserva, la fecha en la que se hace la misma (que se pasará como LocalDate.now() al hacer el set de este field), el nombre de usuario (que podría rescatarse de la sesión actual del navegador o solicitarse en el formulario de reserva) y su teléfono (que habrá que consultarlo en el formulario de reserva), además de una referencia a la actividad en la que se ha hecho la reserva, que contendrá un documento con los atributos correspondientes a ésta, en el ejemplo incluiremos solamente su id (por ejemplo). Nuestros documentos JSON para modelar un objeto `Reserva` tendrían la siguiente forma:

```
{
    "_id" : ObjectId("01.11.Fotografia90"),
    "fecha" : ISODate("2022-11-01"),
    "usuario" : "MawiMorenoPS",
    "telefono" : "601234567",
    "actividad" : {
                    idactividad: "6Nov2022",
                   }
}
```

Los 3 archivos `*.html` que se adjuntan en la carpeta junto con esta memoria y el proyecto usados para los formularios se han de encontrar en la siguiente carpeta para que la aplicación funcione como es debido:
.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\ECPaquita74

La base de datos debe tener el nombre *ecpaquita74* y la colección de actividades tendrá nombre *actividad*