Intro to the Music Project

The Music Project DB

Look at MusicProjectSchema.html, a portable schema.

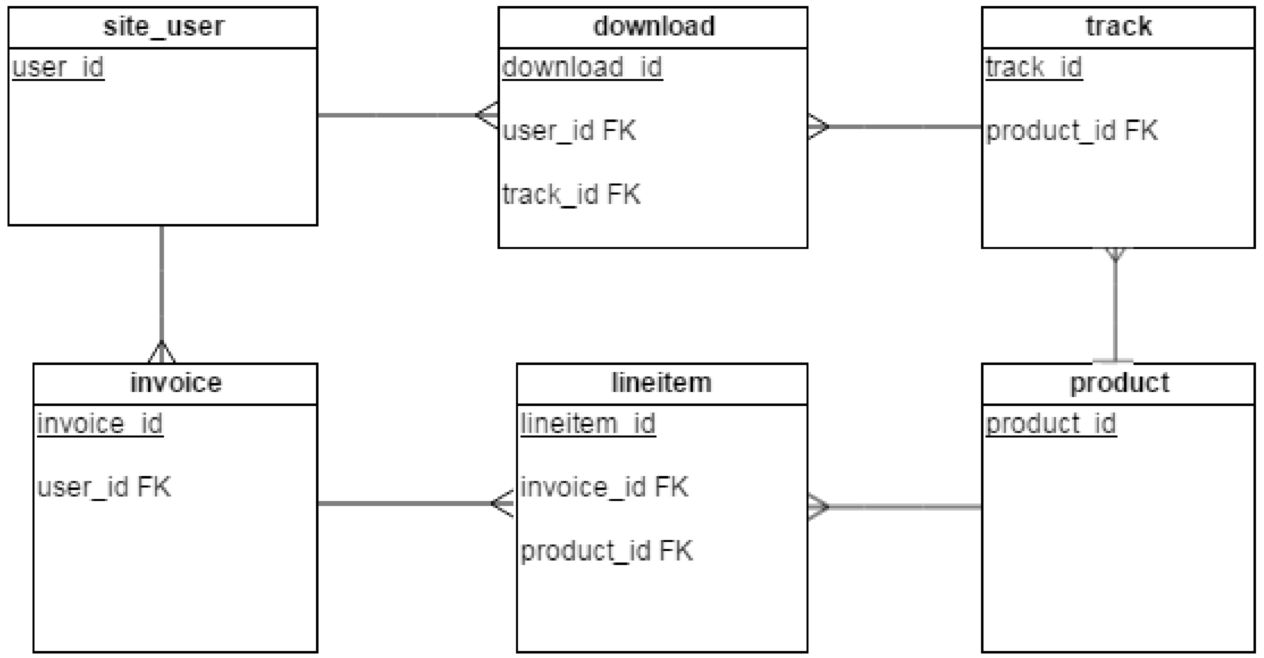
Also, look at Murach's MySQL-specific script, with its mysql-specific constructs.

In Murach's script, skip over the database creation of the DB named "murach" and look at the music database created second in this script.

See mysql extensions of SQL (FYI, not officially covered in this class):

- Auto_increment (a nice feature, but not standardized in SQL92, or supported by Oracle)
- Case-sensitive database ids: table name LineItem, column UserID
- Datatypes DATETIME, bigint, tinyint, (SQL does have smallint), int(11)
- Enum, multiple inserts in one statement
- KEY for suggesting index on column

The E-R diagram in crow’s foot notatin



- This diagram shows the FKs determining how the tables are related.
- It shows a "crows foot" at the “many” end of the one-to-many relationship, where we would put a star in the UML-style diagram. This is another common notation.

Notes on the music database setup

- Note no N-N relationships, so no “extra” join tables
- like Murach, pg. 663, except for relationship notation and additional track table.

The FKs define the one-to-many (or many-to-one looking the other way) relationships, and their constraints help keep the database consistent.

- But FK constraints do cause problems for load and drop database scripts : we need to order the create tables and drop tables to avoid FK violations.
- First table create : need to find a table with no FK columns. Choice of site_user or product. createdb.sql uses site_user. Once site_user is created, we can create tables with FK to it. And so on.
- First table drop : need to find a table with no incoming FK constraints to it. Ex : download. With download gone, can drop track, and so on. See dropdb.sql.

Notes on the music database setup

Tracks vs. whole CDs (Products)

- Users can “download” a sample of a song to listen to it. Note that a user downloads an mp3 file for a *certain track* of a CD, not for the CD in general, so the download row should reference a row of track, not product.
- On the other hand, a user *purchases* a whole CD, not a single track, so the lineitem references a certain product, not a certain track.

Relationships in DB and among Objects

Consider a table with a FK to another entity table. The corresponding domain object often has an object reference to the related domain object, but not always, because we only implement refs between objects that we actually plan to use.

In the pizza project, the pizza_orders table has a FK to pizza_sizes, and the PizzaOrder object has a ref to a PizzaSize object.

However, the pizza_toppings table has a FK to pizza_orders, but there's no ref from PizzaTopping to PizzaOrder. This is because we never need to navigate from a PizzaTopping to its order. Instead, objects are connected the other way: the PizzaOrder has a Set<PizzaTopping>, just the details of the order.

In the music project, the Download table has a FK to Track, and a Download object has a ref to its Track object, to allow us to navigate from a Download to the Track to see its details.

Relationships in DB and among Objects

A Product has many Tracks, details of it like toppings for a pizza order

So Product needs a Set<Track> as an instance variable.

A Track object has a reference to its Product (so we can find the Product for a download by object navigation)

So we've found a bidirectional object relationship between Product and Track!

But a Track does not have Set of Downloads --no action requires this lookup!

An Invoice has a set of lineitems, much like the toppings for a pizza order, so the Invoice object has a Set<LineItem>. We probably don't need to navigate from a LineItem to its Invoice.

As in the pizza project, we need instance variable (fields) of domain classes to hold inter-object refs, or Sets of them.

Review of Object variables: fields (instance variables) vs. local variables

Consider this code:

```
public class InvoiceDAO {
    private Connection connection;   <-- field (instance variable)
    private LineItemDAO lineitemdb;  <-- another field
    ...

    public void insertInvoice(Invoice invoice)throws SQLException{  <--method header
        Statement stmt = connection.createStatement();  <--local variable, inside method
        int invoiceID =  getNextInvoiceID();   <-- another local variable
        invoice.setInvoiceId(invoiceID);
```

- fields (instance variables) live longer than local variables--as long as their object survives
- local variables live only for the current method (or code block) execution, then disappear
- DAO, service objects live as long as the app runs, so fields of these objects also stick around
- domain objects usually have much shorter lives, just long enough to transport DB data to the service layer for example...

Architecture Definitions and Principles

Definition: Domain Data.

Domain data is app-specific data, data that is processed or used in the app and is or could be kept in a database. UI details like HTML don’t count as domain data. Domain data may or may not be held in app-defined Java objects.

for pizza: pizza order number, PizzaOrder object, row, student room number, PizzaTopping object, topping row, current day. Note that the current day is held in an integer variable--domain data doesn't have to be held in objects.

for music: User/site_user, Product/product, etc., Not in DB but still domain data: mp3 files, Cart

Not “domain data”: the service objects, DAOs: these are infrastructure things

Shared vs. private data

Many users of our application can be clients at the same time, coming in from different client machines.

Private Data. The presentation layer runs on behalf of a certain user session, so its domain data collection is private to that user. For example, the Toppings chosen by a user who is thinking about making an order. A current user's room number (our simple notion of user identity).

Shared Data. Data that can be accessed by multiple users is shared data. To protect it, we make sure all changeable shared data resides in the database. Our static web pages are shared, but are considered unchangeable.

Idea of a Stateless Service Layer

A stateless service layer has a stateless Service layer API, a set of stateless methods.

A stateless method is a method that is self-contained in the same way that web services are self-contained, so you are getting experience easily transferred to that realm.

We have already discussed how the service layer calls represent what the system can do. Each call represents some particular action.

Stateless: from http://whatis.techtarget.com/definition/stateless: Stateless means there is no record of previous interactions and each interaction request has to be handled based entirely on information that comes with it. Or long-term knowledge in the database.

What’s good about stateless?

It offloads the job of holding state from the Java core program environment. This is particularly important once that core code is running inside the web server. It is also more robust—if a message gets lost, it doesn’t change the *meaning* of the current request.

HTTP is stateless: each GET must say exactly what URL on the server it is asking for, for example. HTTP service is so simple that printers and other apparently non-computer devices can provide it.

- **Aside (FYI):** Other important stateless services: IP, NFS (Network File System, the distributed file service on UNIX/Linux), web services.
- The Windows distributed file protocol, SMB (Server Message Block Protocol), is stateful. A client can open a file on the server, and the server may deny other clients access to that file until the client closes it See StackOverflow 5436069
- **End Aside**

Stateless Service Layer Rule

No domain data is saved in the service layer (or DAO layer) between calls to the service API. Instead, each call works entirely from its arguments plus data from the database obtained via DAO calls.

Note that this is why StudentService.java has no fields holding domain data, just the infrastructure references to the DAOs. And similarly the DAO objects. And that is why StudentService can be a singleton, since if we had two of them they would be identical, just a waste of bits.

How does this make calls self-contained? They can't know things from past calls, unless that info was put in the database or held by the presentation layer (where it is private to that user) and sent back down through an argument.

This is another way of saying that we are completely dedicated to using the database to hold changeable shared data. And we like to get the user-private data stored in variables in the UI code. So the code in core of the app ends up not having to store any domain data for a significant time, just for a moment in local variables during the service call.

Example of non-self-contained call (not in use, of course)

```
void studentService.addToppingToOrder(Topping t) throws …
```

- This assumes there is a "current order" being built up, known to the service layer code, but this is not allowed. That current order would have to be held in a field of StudentService, and that is not allowed by the stateless rule.
- If we allow toppings to be added to orders already in the DB, then we need to pass the order ID in the call. We need orders to have another status, say "Incomplete".
- However, this complicates the system, because now there are incomplete orders to deal with in the database while the user makes decisions about what toppings should be there. We'll keep things simple by using makeOrder(...), which has arguments for everything needed in a pizza order.

Example of *temporary* data held in local variables while we're computing in the service layer.

- Here temporary means a lifetime contained within a single service call.

```
public void markNextOrderReady() throws ServiceException {
    try {
        PizzaOrder order = pizzaOrderDAO.findFirstOrder(PizzaOrder.PREPARING);
        if (order != null)
            pizzaOrderDAO.updateOrderStatus(order.getId(), PizzaOrder.BAKED);
     } catch (SQLException e) {
         throw new ServiceException("Error in marking the next order ready",e);
     }
}
```

Local variable order

This still satisfies the rule, because this object's lifetime is all inside one service call. It doesn't provide information to another service call.

We can't split this up into two service calls, however, without breaking the rule or putting this code in the presentation layer, which is not a good idea since it's a core action of this app.

Example of longer-term data in presentation layer

Consider roomNo in TakeOrder. It is held across several calls to the service layer.

Here is a run of TakeOrder, done after SystemTest has been run, so there is a pizza size and a topping.

```
Possible Commands
O: Order
S: Status Report
R: Receive Order (acknowledge receipt)
Q: Quit
Please Enter the Command:
o
Please Enter the room Number:
5                                ← program gets room no. into a variable
Basic Pizza: tomato sauce and cheese
Additional toppings:             ← program calls getToppingNames, getSizeNames
 Pepperoni
Sizes:
  small
…
 Thank you for your order        ← program calls makeOrder(…) using room no.
```

The presentation layer is stateful for a certain user

- Here the room number is held in the presentation layer, in a variable, across several calls to the service layer.
  - That's fine because it's user-private. The presentation layer is private to one user.
  - If there are two current users of the app, each has their own client process with their own presentation variables.
- We see it is OK to hold domain data in variables across calls to the service layer in the presentation code. The room number is saved across service calls, as shown above. Also, getToppingNames() and getSizeNames() get domain data in Strings, then use UI for user choice (subset of Toppings, certain PizzaSize, specific to one user), then makeOrder using these ints and Strings.
- But don't keep whole domain objects around in the presentation layer between calls to the service layer. An exception can be made for immutable domain objects such as MenuTopping. But mutable domain objects are just scratch copies of database data at one point in time, so can go stale. And we expect the service layer to be in charge of any changes in them.

Architecture Principles and Assumptions

1. **Pull UI.** We are planning to implement a “pull UI”, that is, all info provided to the user/client is a response to a user request, i.e., all info provided by the service API to the presentation code is via returns from calls.

There is no notification to the user generated by the system (email can be an exception here), for ex., to notify the user that their pizza order is ready.

This means we can use a simple call-down layering: no upcalls are needed (an upcall would be a call from service code to presentation code).

Also note that JDBC follows the pull model, that is, it can’t notify its calling code about changes in the database, just respond to explicit requests for database state.

A web app using HTTP is using a pull UI, and this is our goal.

2. **Stateless service layer and service API**. Each service call is self-contained in sense of knowing what to do based only on its arguments and its access to database state via the DAOs. The DAOs are also stateless in the same sense of not holding domain data separate from the DB, but this follows naturally from their job as a veneer over the DB.

3. **Repository for shared domain data.** All changeable domain data held between service calls that is shared among users resides in the database. The database is the expert in handling shared data, especially shared data that changes over time.

Note: the database doesn't have to be relational, but our three databases are relational.

Interdependence of rules/assumptions

1. **Pull UI.**
2. **Stateless service layer and service API**
3. **Repository for shared domain data: the database.**

Note that 3 follows from 2: a shared datum x is not allowed to be saved in the service layer or DAO, and the presentation layer is private to a user, so by elimination it must be in the database. On the other hand, the database is allowed to hold user-private data, like shopping carts.

But 2 does not follow from 3, since 3 does not cover user-private data. So a system that follows 1 and 3 allows user-private data held in the service layer, making a heavyweight system that is hard to analyze. We'll stick to the stateless service layer model.

Another assumption: only one database

- Working with multiple databases is extremely difficult, “distributed databases”, so we are assuming only one in use.
- It’s better to stretch your database system than to duplicate it.
- Use a DB server with 40 or even 100 CPUs if you need to. If your site is bigger than that, you can afford consultants that can know how to use distributed databases.
  - If using the AWS cloud, consider using Aurora, which acts like a traditional database but internally stores data across multiple servers, allowing much greater scale-up without explicit distributed database work.
- Later, we will consider splitting the database into two parts, the "sales" db and the "catalog" db, as a step towards “microservices”.
  - This still means that each *table* belongs to only one database, the essential simplifying assumption.

Another idea: **Thin presentation layer.**

- The presentation layer should do the minimal work for UI. No business decisions, no domain object changes or creations. Just displaying data to the user and finding out what they want.
- If a domain object needs to change, name the action and put it in the service API.
- However, we have seen that the presentation layer code may *hold* domain data (as ints, Strings, etc., or possibly immutable domain objects) across calls to the service API.
- By having all the real actions of the app coded in the service layer, and expressed in the service API, it is possible to implement multiple presentation layers for different ways of using the app.
  - For example, a web app for big-screen clients, vs. for smart phones. Or as part of another app, receiving direction via web services.

Monolithic code

- We are implicitly saying that all this code is in one project, although it's not necessary to the basic principles.
- If it is all built together, it is called **monolithic code**. If a bug shows up in one little part, the whole thing has to be replaced.
- So when the codebase gets large, it's time to break it into **microservices**.
- Actually it's the number of programmers working on it that mattes most. If that number exceeds 75-100, it's a real problem. We'll cover this later. The microservices still hue to the basic principles, and each has its own little database.