Data Access and Integrity

Our class schedule shows two more topics this week. The first one is Authentication and Authorization, and the second one is Tokens. This set of slides and notes cover both of these so, I'm sorry to say, it's going to be on the long side.

We'll start with Authentication and Authorization which I've also referred to as data access. Let's go right to the first slide and see what the difference is between authentication and authorization. Simply put, authentication is about who you are, and authorization is about what you can do. Whenever we access any protected data or physical site, both of these aspects have to be considered. When we are doing this remotely as with a software application, it can be a challenge to be sure that the user is who they say they are and to be sure they only do what they are allowed to do. Needless to say, this is something that hackers spend a lot of time focusing on. Let's move to the next slide and begin to take a deeper look at authentication.

Let me preface what I'll be saying in the next several slides with the statement that I am a developer, not a security expert. What I am going to tell you is from a developer's point of view and represents a developer's knowledge of security. If I were working with a very secure system, I would not trust my own level of knowledge. I would enlist the help of a security expert. You should do the same.

With that disclaimer aside, let's get keep going. When it comes to authentication, security experts consider the ideal situation to be one where the user needs present something they know, something they have, and something about themselves. This is known as 3-factor authentication.

As a user, you commonly need to supply both your user id and a password. This is using two examples of *something you know* and is referred to as single factor authentication. The fact that you need to supply two forms of *something you know* is irrelevant, it is still only single factor authentication. It is likely that to get into your home, dorm room, or apartment, you simply need to insert a key to gain entry. That is also single factor authentication, in this case it is *something you have*. There are many software packages, particularly the more expensive ones that serve niche markets, that require you to insert some sort of a dongle into your USB port. This is also a form of single factor authentication, where the vendor is ensuring that you are the person who purchased the software by virtue having the dongle in your possession, *something you have*.

Think about some of the rooms in Golisano. You may have noticed that to get into the open lab, you simply have to swipe your ID card. This is yet another example of single factor authentication based on *something you have*. On the other hand, to get into our computer classrooms, your professor needs to swipe his or her ID card **and** enter a pin. This is 2-factor authentication requiring *something they have* and *something they know*. Some software systems also require 2-factor authentication. RIT has implemented this for MyCourses and SIS. They refer to it as multifactor, but it really is only 2-factor. It requires that you enter your login

ID and password, *something you know*, and then respond to a challenge on your phone, *something you have*. Other systems implement 2-factor authentication by requiring a pin and then sending a code to a device you have, perhaps your phone or an RSA stick.

Regardless of the number of factors required, there's a great deal of room for a variety of approaches. For example, I'm sure you are very familiar with the expiration rules on passwords as well as the rules for the strength of a password. These rules are an attempt strengthen the single-factor authentication. Again, we are the developers not the experts, but when devising these rules, one must consider the human element as well. Always design systems to encourage desired behavior on the part of the user, make it easy for them to do the right thing.

Using biometric factors, the *something about you* factor, is becoming more common, but usually as a substitute for one of the other factors. For example, many laptops will now accept a finger swipe instead of entering a password to unlock the machine, swapping *something you know* for *something about you*. I have only seen all three factors implemented in science fiction movies.

Moving to the next slide, the question that is posed is who should be doing the authentication, the system or the application. This is a question that comes up often in many aspects of software applications. Do I use services that already exist or do I write my own? And the trade-off is always the same. If I use existing services then, in theory, I should have a more robust service available to me. It would be a service that has been tested and used quite a bit and is, therefore, reliable. On the other hand, if I write my own, I can customize it for my situation. I can make it efficient and specific.

Which do I use? Just like when we are choosing a database driver, your best bet is just to go with the existing, general purpose solution unless you have a very specific need to do something else. So, in our case, it is generally a better idea to let the built-in security of our environment handle authentication. Here at RIT, that means using LDAP or Active Directory to authenticate our users. This also makes life easier for our users because they will not need to remember different sets of credentials or different processes for logging in.

Let's move to the next slide and the topic of authorization. When we talk about authorization, the assumption is that the user has been authenticated and now it is a matter of constraining what this particular person can do. When dealing with database applications, we can set this up at different levels. One level is within the database itself. I'll explain this, but I strongly recommend that you also investigate the options on your own. You can do this from the command line, but it's more convenient to use one of the many graphical interfaces that exist. Many of you have already used MySQL's Workbench. That's a nice tool, but I prefer Heidi SQL for a PC or SQLPro on a Mac. I am sure there are plenty of other great tools out there as well. Using one of these graphical interfaces will allow you to explore what can be done at the database level. Again, you could certainly do all of this at the command line using the grant and show grant commands of SQL, but it'll get pretty messy and I prefer the graphical interface, personally.

If you use one of these graphical tools and click on its option for user management, you will see that you can specify which databases that particular user has access to and, within the database, what operations they can perform. For example, you have been using some databases that I have on my servers. I set up a user ID for you and assigned very specific rights to that ID--you can do a SELECT, but you can't do an UPDATE or DELETE. If I wanted to, I could even define those at the table level. It is really very powerful.

For our purposes, I was able to define a single id and get that out to all of you. That is not what we would want to do with the database application. Instead, I would need to set up ids for each one of you and assign rights to each one. Of course, only the DBA would be allowed to do that. For my servers, I am the DBA so it is up to me whether I want to take on that work. However, in a production setting, it is a very rare situation that you will find a DBA who wants to do user-level management.

So, that brings us to application level authorization. In this case, the authentication might still have been done by the system, but now it's up to the application to determine what that person can do. And while doing things at the database level can be very powerful, doing them at the application level actually gives us additional flexibility. For example, a common situation is that there'll be several people who "own" their data that is in a given table and need rights to edit it, but should not be editing anyone else's data that is in that table. That type of granularity is not possible at the database level. But, at the application level, we can use appropriate code to ensure that any given user is editing only data for which they're authorized. So, using an application level authorization approach puts extra work on the developer, but it also creates a great deal more flexibility, and that is often necessary.

Let's go to the next slide and take a deeper look at application level authorization. There are several ways which I can do it. The first way, and I mention this only for completeness, is treating all users the same. This is a bad idea. There may be certain situations where it's appropriate, but by and large you are essentially removing all security with this approach.

A much more secure approach would be to provide authorization at the user level. For every user, you indicate what they can and cannot do. In order to implement this, you need to find all the possible tasks that might be done. Let's go to the next slide on task level authorization to take a better look at that.

Think about your CRUD operations: create, read, update, and delete. You have to build those methods for every table. You can think of each of those as a task the user may want to do. That means for every table there are at least four tasks. And I can tell you right now that you'll need an even finer grain of task definition than that. As you can imagine, you will have a very long list of tasks. Essentially, what you'll do is set up a table of these tasks, and then, of course, your users table, and an associative table between the two where you can specify which users can do which tasks. I had to build a system using this approach one time. The users insisted on it. However, when we went into production, they realized how difficult it was to simply add a new user and go through and check off all the boxes indicating what a user could or could not do. I

ended up adding another feature where, when adding a new user, they could say they wanted to copy the permissions from a different user. That saved them from the tedium of specifying each capability for this new user, but it didn't help maintenance. If it was decided that a bunch of people should be given an additional privilege, they still had to go through and manually add that privilege to each of the people, hoping they didn't miss someone. So again, while the task level authorization gets you a great deal of granularity, it can be very difficult to develop a system that makes it easy to use and maintain the information.

That takes us to the next slide and the most common form of authorization, role based. This is similar to group policies, perhaps you've dealt with those. In essence, we're going to define classes of users, referring to these classes as roles. We assign privileges to each role, and roles to each user. As long as we keep the number of roles to a reasonable number, this becomes a very easy to maintain setup.

Most database applications can be thought as a means of controlling access to a repository of information. There will be people who put information into the repository, people who look at the information that's in the repository, and people who oversee the operation of the repository. Within that, these people may have constraints on which information they can see or add. This broad classification gives rise to the roles I have listed on the slide.

Working from the bottom up, the public role typically has read only access and only to certain information. The general user typically has read and write access to data that they "own." An editor is someone who has read and write access to a much larger amount of data. And finally, the admin has access to all the data and also to any configuration data. Now, not all database applications will have these four roles. And, there may be other roles that are more appropriate for a particular application. But these are very common and you should get comfortable with them.

Keeping the number of roles down yet still providing sufficient granularity is a balancing act. SIS is an example of where, in my experience, the balance is tipped too far toward minimizing the number of roles. You have your role as a student. There is no role for a department head. Instead, I have something that is called "Level 1 Access". When I log in to SIS, I see about 40-50 options, the vast majority of which I have no need for. On the other hand, there are functions I need to do and can't—I have to ask one of the staff who work for me, and have "Level 2 Access", to do it. Defining roles is a balancing act.

Role-based authentication is managed with a combination of data and code. A user's role is data and is typically stored in the user table. On the other hand, whether that role is actually authorized to perform a particular operation is determined in code. So, for example, in a method that would update the configuration table, an if statement in your code would check to see if the user was in the Admin role before proceeding.

The next slide lays out the general considerations when doing application level authorization. Whether it is task oriented, or role based, or some other method, we need to be sure that the

current user is authorized to perform the requested function. Not only do we need to ask the expected questions regarding who they are and what are they entitled to do, but also, if they are somewhere where they should not be, how did they get there and what went wrong earlier in the security protocol. Typically, that latter issue will be dealt with in our log files. That is, it's not just exceptions that write to our log files. If the logic of our program detects an abnormal situation, it should also write out something to our log file.

The next slide deals with business rules. I don't want to spend too much time on this, but you should understand the difference between what happens in the data layer and what happens in the business layer. Business rules are coded in the business layer. Sometimes they are rather simple, sometimes they are nonexistent, and sometimes they're very complex. Some basic questions business rules address include: who has access, what do they have access to, what time of day or day of week do they have access, and from where do they have access? I've already talked quite a bit about the Who and What. The time is often a factor if you work for the defense department or some other highly secure organization. You may find taht you are only allowed to access the system during normal business hours and, perhaps, only from on-site. Even here at RIT, several of our systems are accessible only when you are on-site or using a VPN so that it looks like you're on-site.

I have to tell you an amusing story about this. Do you remember the ear infection system I talked to you about? Prof. Bogaard and I developed it for doctors at Rochester General Hospital. Because it was dealing with patient data, it needed to be audited to ensure it was following the prescribed regulations. One of the regulations was the need to maintain a white list of IP addresses. That is, we had to ensure that whenever a user was entering the system, they were coming from an IP address that we had already approved. Prof. Bogaard and I tried to convince the auditor that this was silly because it was very easy to spoof an IP address, but the government auditor didn't care and insisted that we put the in the feature. So, we did.

About a week later I got a frantic phone call from one of the doctors. She asked me if the system was down because she couldn't get it. I checked and everything looked to be running just fine. She then went on to explain that she was at another doctor's office trying to demonstrate this system to them and she was unable to get in. Do you see what happened? Because she was at another doctor's office, her IP address was new and was not recognized by the system. So, like many overzealous security practices, instead of keeping out the bad guys, all this did was make life difficult for the good guys. The story does have a happy ending. I showed her how to use "whatismyipaddress" and then I added it to the system and she went on with her demonstration.

That's about it for business rules. I feel bad having just this one short slide on the topic; it really is very important. If you're going to live your life strictly as a data layer developer, then you probably don't need to know much more about it. But, if you are going to be a full stack developer or any sort of application developer, you need to appreciate the importance of the business rules. And I'll tell you right now, based on many scars, you need to do a very thorough job upfront when you're gathering requirements in order to find out what all the business rules

are. People always think in terms of functional requirements. When you are talking to new users about building a system for them, they'll tell you all about the things they need it to do. That's great. But, they will forget tell you about the business rules because they are just part of their lives and things they take for granted. So, if you are in the overall process of application development, for your own sake, pay careful attention to the business rules.

Okay let's go to the next slide and get into some of the details about checking whether a user is authorized. There are two basic methods. The first method is the most secure and the most expensive in terms of resources. What you do is the following. Every time a particular operation is invoked, get the user's id and query the database to determine whether they have the right to invoke that operation. If they do, great, go ahead. If they don't, then you exit the process, perhaps with some error message. This is a secure approach because we're not passing any information other than the current user and it is dynamic in that if the DBA changes the user's rights while they're in the system, the changes will apply. Of course, there is no free lunch. This approach can be expensive because with every operation you're making an additional call to the database.

The second approach less secure, but far more efficient. When the user is authenticated, they're issued a "ticket" or a "token", a string of characters in which their rights are embedded. Every time the user tries to perform an operation, this ticket is passed along. The receiving method, instead of making a call to the database, just needs to look inside the ticket and determine whether the user has the right to perform the operation.

This is less secure because we are sending valuable information with every call. If a hacker gained access to the data stream, they might be able to use it to break into the system or do some other form of damage. It also establishes the user's rights when the user logs in and they will retain those rights throughout their entire session regardless of any changes that the DBA might make. On the other hand, it is very fast and efficient.

Which approach you use depends on the needs of the system. Most often, this second approach that favors efficiency over security is what systems use. But that means the security of that ticket becomes very important—we want to be sure it cannot be reverse engineered. Let's look at the next slide to get into this a bit more.

Security experts may have a more precise definition, but for our purposes we're going to say a token and a ticket are the same thing, a string of characters that contains secure information and is passed between methods or systems. You may already know the difference between hashing and encrypting, but for those who don't I'd like to go spend just a moment on it. A hash is a one-way conversion of characters. I am sure you are familiar with SHA1 and SHA2, these are examples of a hash. Once a word or phrase has been hashed, the original word or phrase cannot be retrieved. To make it even more secure we can add "salt" characters and we can hash it several times. These additional steps are used to prevent forward guessing.

Given the horsepower of today's computers, it is possible to take many phrases and hash them according to a specific algorithm, for example MD5, and generate what are known as "rainbow tables". A rainbow table allows a hacker to do a reverse lookup to determine what your original password or pass phrase might have been. Adding salt or hashing multiple times is generally sufficient **today** to prevent the use of rainbow tables. That will probably change!

Encryption, while also a conversion of characters to secure the original word or phrase, is specifically meant to be undone. It allows us to pass a secret message to the receiver and have them decode the encrypted string to determine the secret message. Because it can be undone, it is inherently less secure then a hash. But, it does allow you to convey information; the hash does not.

As an aside, think about your car or your parent's car if you don't own one. What extremes do you take to prevent it from being stolen? Do you walk away from it with the keys still in the car? Do you always take the keys and lock the door behind you? Do you have an alarm system on the car? Do you have an armed guard that you pay to stand watch over the car? Your answers to those questions probably depend on the value of your car.

The same is true for your software applications, in my opinion. If you work in financial institutions or some other organization with a big target on your back or with very high stakes, then I am sure you'll want to use the most secure approach possible. On the other hand, if your company is just one of many nondescript organizations and you were building an internal application, then your security level can probably be less.

This attitude brought me to develop an encryption algorithm for tokens that I have used for years and I will share with you here. To the best of my knowledge, none of my clients have ever had their application broken into. And if they did, it would mean a minor disruption to their business and then life would go on. Before I can describe the algorithm, I need you to be aware about the concept of bases in number theory. You know that we work in a decimal system that is a base 10 system. You also know that the binary system of computer is called a base 2 system. You probably also know that when we look at a dump on the computer it is shown in a hexadecimal format or a base 16 system. There's nothing magical about the systems. I could have a base 25 system, or base 17 system, or any other system if I chose.

Okay, with that in mind I can now tell you about the algorithm. The first step is to determine what information you want to store in the token. This depends greatly on your business rules. Let's say you need store the user ID, what time they entered the system, and their access level.

The next step is to convert each of these pieces of information into a string of numbers. Where there are characters, you can simply use the ASCII code. You then need to pad each of them so that they are some fixed length. For example, the user ID always be 15 characters, and time should always be 12 characters, and so on. Now convert each of them from base 10 to some other base, using a different base for each piece of information. For example, convert user ID to base 17, convert time to base 21.

Finally, break each of them into pieces, scramble the pieces according to a specific set of rules, and insert the pieces into your token string. So, the first two characters of time might go in position 18 and 19 of the token. And the next two characters of the time might go into position 5 and 6 of the token. You can also insert random characters at various spots in the token.

Once you have done that, you have your encrypted token that you can pass around. You create it when a user first logs into the system, it is stored in a cookie on the client machine (assuming this is a web application), and it is passed back with every method call. When the application receives the token, it decrypts it and extract the necessary information. I typically create an object called Token that has all the necessary methods for doing the encrypting and decrypting.

So that's my encryption algorithm. Feel free to use it.

Let's go on to the next slide and talk a bit about storing credentials. Some of you have asked me about this already. I'm going to talk about it a little bit now and then will talk about it more in another week or so. I'm going to talk about two different sets of credentials and we want to keep them straight because our handling of them is it different. The first type are the credentials a user needs to access the system. The second type are the credentials that the system needs to access the database.

First let's talk about the credentials the user needs to access the system. As I said earlier, the best situation is to use LDAP, or Active Directory, or some other function of the operating system to authenticate the user. But, if you want the application to authenticate, then you need to store the user ID and password in the database. This is often the case when you are developing a web-based application for a wide audience. Your project for the course, the conference submission system, is an example of this. When you store user credentials in the system, you need to take full responsibility. Obviously, you need to hash the password and store it in the database. But you also need to provide a facility in case the user has forgotten their passport. And you also may want to guard against hackers and be able to lock the account after some number of tries. These are not difficult features to implement, but they take some time and some thought. I am anxious to see what you come up with for your project.

Credentials for the system to access the database can be handled in a few different ways. Which way you choose, as with other security aspects, depends on the nature of the application and your environment. The easiest way is to simply include the credentials right in your code where you connect to the database. Since you are adopting style of having a separate database class it is easy to include a couple of constants in that class that represent the ID and password for the application to connect to the database.

This approach exposes the credentials to the developers and anyone else who might see the code. Is that a problem? The answer, of course, is it depends. The developer has access to the database via the application. Even if we hid the password from the developer, odds are the developer could still utilize the application code and write any SQL statements to be executed against the database. Seeing the credentials is not an in issue, in my opinion.

Now, who else might see that code? That becomes a matter of the security of your overall environment. So, you need to give that consideration in determining whether you store credentials right in the code. One could also make the case that storing credentials in the code prevents me from showing the code to others. Consider your homework that you submit. Do you include in your code the credentials to access MySQL on your computer? If so, you're telling me and the TA what your credentials are. That's not good.

So that gives rise to a second method of storing our credentials—putting them in a separate file. Taking our credentials out of the code allows us to share the code with anyone we want without worry about the divulging any secure information. But now, who has access to the file where those passwords are stored? Again, this depends on how your environment is set up for security. This is, however, a preferred approach to storing database credentials right in the code. In another week or so we will talk about larger systems and another way to store the database credentials. Until then, we'll need to stick with these two methods.

The next slide, titled SOA considerations, is about larger issues. Think way back to the beginning of the course when I first drew the SOA architecture on the board. There was the presentation layer, the business layer, and the data layer. Recall that I showed I could have several different databases involved with several different data layers, and several different business layers, all feeding different applications. When you have a situation like that, the passing of establishing and maintaining access privileges gets much more complicated. I have no set answers for you. It all depends on your situation. The key is for you to be aware that it needs to be managed. I may very well have multiple layers with authentication happening at each layer. It's all a matter of how distributed, or federated, my system is. I just want you to be aware of what you might find yourself dealing with.

Let's change gears a little bit now and talk about data integrity. We are going to start with the slide titled Auditing the Database. Auditing the database is primarily about who I blame for bad data. If you're working in the healthcare system or the financial system, any change in the data needs to be documented. Once the data has been entered, it is considered sacred and should not be changed without a good reason.

On the slide, I say that tracking is important for things such as connections, queries, errors, and performance, but the reality is we typically use it to document who made a change and why they made the change. That aspect is delineated in the second bullet. Depending upon how thorough your audit requirements are, you will need to record some or all of these items every time a change is made. This change information might be stored in the same table along with the data, or it might be in a separate tape.

The next slide shows two common methods for doing an audit. These terms, "snapshot" and "full", are my terms and not necessarily industry standard terms. To be honest, I do not know if there even are any standard industry terms. The snapshot is simply a recording of a point in time, without any history. Usually what we will do is add a few additional fields to the table in

order to record metadata about the change: a user ID, a time stamp, and perhaps a reason. Then, anytime any data in that record is updated, or inserted, we also update these extra fields. If I go in and change the value of some field, then my name, the current time, and a reason are stored also. If two seconds later, you go in and make a change to any field in that same record, my information is replaced with your id, time, and reason—there will no longer be any record that I had made a change. The data that I changed will still be there, but no one will know that I did. That's why this is called a snapshot; it is only effective on relatively stable databases

The "full" audit is much more thorough and requires the use of an additional table. In this method, when someone makes a change to a field, the table is of course updated, but we also enter into a separate audit trail table the following information: the table that was updated, the name of the primary key field, the value of the primary key, the name of the field that was updated, the old value, the new value, and all the information about the user that we stored with the snapshot approach. We are effectively building a complete record of changes so that we could roll back the database to any point in time, as well as knowing who was responsible for each change along the way. You can imagine that this is a lot of information to store. Think about inserting a new record—there will be an entry in the audit table for each field of the new record. This audit trail table will grow very long, very quickly. That's okay, your database system can handle it.

When you implement this, you will need to use a transaction. You will need to:

a. Start the transaction
b. Get the "old" values
c. Insert the "new" values
d. Insert "old" and "new" values (and metadata) into the audit trail table
e. Commit the transaction

Let's move on to the next slide, data integrity. These next few slides take a look at the data that a user is entering into the system. Now while much of this should be dealt with in the presentation layer, it is still our job in the data layer to support that effort.

This first slide has a very common phrase, "garbage in equals garbage out". The best way to be sure we have good data in the system is to be sure that only good data gets into it in the first place. The name of our school is a great example of what can go wrong. RIT could be written as Rochester Institute of Technology, Rochester Inst of Tech, just RIT, or some other arrangement of letters. This is why, in my opinion, text boxes are evil. Giving users a free form vehicle to enter information means that they can enter anything they want.

Why do I care? Well later on when I want to know something about a particular company, I want to be able to search on that company and be certain I'm getting all the information about it. If RIT were entered in the database in multiple ways, when I searched for it by one of the names, I would only get a fraction of the information about it. This could have serious financial implications. When if there are volume discounts that need to be considered? Even though we might be purchasing large quantities historically, that will not be obvious.

The best way to handle this is by providing drop downs or select boxes for the user. Go to the next slide where I talk a bit about form validation. Again, it's a matter of choosing the right tool for the job. You will spend much more time in some of your other courses discussing all the options available to you. I just want to encourage you to be sure to use them.
But it's not enough to check all this just at the form level in the presentation layer. You want to be checking things all along the way because you don't know where it may have come from. Remember the situation where I may have multiple presentation layers, multiple business layers, and multiple data layers. Trust no one.

Moving on to the next slide, what about data modifications? The most important thing for you here is to maintain data consistency. You can use the built-in cascade feature of your database to be sure that parent-child relationships are enforced. Just as with our earlier discussions, using built-in features is the easiest and most robust way to proceed, but sometimes may hamper our flexibility. There will be times when you will want to turn referential integrity off and handle everything in code. That's OK, but be aware that you have taken on a very serious responsibility. Be absolutely certain that whatever code you write is fail safe.

I want to explain a common situation to you and how best to handle it. You are familiar, of course, with the one-to-many relationship. There will be many times in a database application where you need to display that one-to-many relationship to the user. A very common layout is the have the top half of the screen be about the parent, with all the fields from that record, and the bottom half the screen be a list of several child records. What happens when the user wants to make a change to some of the information listed in the child records?

There are 2 ways to go about this. One way is to require them to first open that child record so that is all they are looking at, and have them make the changes on that screen. The data is then sent to the back end and the update is made in the database. But sometimes there's a need to make a change on the first screen that is showing the one-to-many relationship. It could be that I wanted to delete one of the child records, or I want to add a new child record, or I want to change some of the information that is displayed. How do I tell the back and what it should do? I could write a very sophisticated routine that monitors old and new values that are displayed and then just sends the appropriate updates to the back end.

But a much easier and more common way is to send all the child records that are displayed to the back end. There the system will first delete the child records that are in the database, and then it will add these new ones. This means even that if I only changed one small piece of information, I am still deleting all the child records and then re-inserting all the child records. This may seem like a lot of work, but as long as we're talking about a reasonable number of records, say 5 to 20, it really is very quick and very efficient. **However**, this is a perfect example of needing to use a transaction. I certainly wouldn't want the system to fail in between the delete and the insert. So, to implement this I would first start a transaction, then delete the child records, then insert the new child records, and finally commit the transaction.

Another comment about deletion… In many production systems we rarely delete records. Instead we mark them as deleted, archived, not active, or something else along those lines. The reason we do this is to maintain historical records. Adding an extra field to your table to hold this status information is easy, but it is often the source of bugs in queries. Now, whenever you design a query, you need to think about whether you want to select all the records, just the "active" records, or maybe just the "inactive" records. This is not hard to implement, it's just one more WHERE clause; but it can be hard to remember! Often times, people will forget to include this filter and will get far more records than they anticipated!

And, finally, let's go to the last slide. In some applications the accuracy of the data is critical. In these cases, we may use a dual entry approach. Two different data entry clerks will enter the information. The information is then compared. If it's the same, everything is great, if not the discrepancies are examined and corrected. There are two different ways to implement this; which is better depends on the database throughput. The first way, for low volume systems, is that user one enters the data into the table and a status field (like the active/inactive field) is set to "unverified". When the system receives the information from the second user, it "sees" that this data has been entered and compares the two. If they are the same, the second user's data is discarded and the status field is set to "verified". If different, the second user is presented with both sets of information and asked to check it. Their second submission is then written into the database and the status field is set to "verified". This approach is less resource intensive than the next approach I will describe, but it gives a great deal of authority to that second user. Some systems will require a third person, a supervisor perhaps, to approve (with a key phrase) the changes before they will be accepted.

The second approach is to put both sets of inputted information into a separate "holding" table where they are reviewed by a third person. When that third person verifies the information, it is inserted into the main table. This approach is better for high-volume systems because it minimizes time for a table to be locked and it also allows a more batch-oriented approach, one that lends itself to a room full of operators.

While dual entry still exists in many areas, it's primary use was for keying in survey results (e.g., US Census). This has largely been replaced by online and ICR technology today so you will see this less often. Nonetheless, it is an example of some of the thinking you will need to do when building a database application.

That's it for this week.