Database.com for Architects Series White Paper

Understanding Database.com system and data access controls

WHITE PAPER

Contents

Overview

The *principle of least privilege* is a basic concept that most information technology professionals readily understand. In the context of a software application environment, the principle of least privilege means that you grant system administrators and application users no more privilege than what is absolutely necessary to perform their specific jobs.

Database.com makes it easy for you to implement a security policy of least privilege for all types of users. Every Database.com org is locked down tightly when you first provision it. This paper explains how to use various features such as users, profiles, roles, groups, and sharing rules to progressively open up access so that just the right users have access to just the right data at just the right time.

Here's a preview of how it's done:

1. **Create profiles and permission sets** – Identify the different types of users you need for your application, based on the different functions each type needs to access. Create a base level *profile* for each type of user such that each profile has only the permissions required for that type of user to perform these functions. Then create *permission sets* to handle exceptions—situations in which a user may need a few more permissions.
2. **Create users** – For each person who needs database access, create a *user*, assigning the user to the appropriate profile and permission sets.
3. **Set sharing models** – For each object, set the *organization-wide default record sharing model* to determine whether the records that each user owns are public or private.
4. **Share private records** – Use *roles*, *groups*, *record sharing rules*, and other means to share private records with other users.

Example app and security policy

This paper uses an example app and corresponding security policy to illustrate the concepts and functionality of various features in Database.com's comprehensive security model. Here's a quick summary of the example scenario you'll be reading about (and with which hopefully all U.S. residents are familiar).

- The app manages information about bills for members of the U.S. government, specifically legislators in the House of Representatives and the Senate who vote on bills, the Speaker of the House and Senate Majority Leader who approve bills that were voted on, and the President who signs approved bills into law.
- The app lets a legislator and fellow committee members create, read, update, and delete (CRUD) bills that correspond to the committee they sit on.
- The app prohibits non-committee members from reading and voting on a bill unless its status is "For Review."

Behind the scenes, we'll have three different types of administrators: a super-user, an administrator whose only job function is to manage change sets (see the paper "Architecting Database.com apps: a design primer" for more about these roles), and an administrator whose only job function is to manage users.

Profiles and licenses

Before creating users, it's best to create one or more profiles. A *profile* is a collection of functional permissions and settings that control what a user can do with a Database.com org. For example, profiles control:

- *system-level access*, including time- and IP-based login restrictions as well as permissions that apply to different functions within an organization such as the ability to manage users
- *object-level access*, including permissions to CRUD records for each object in the database

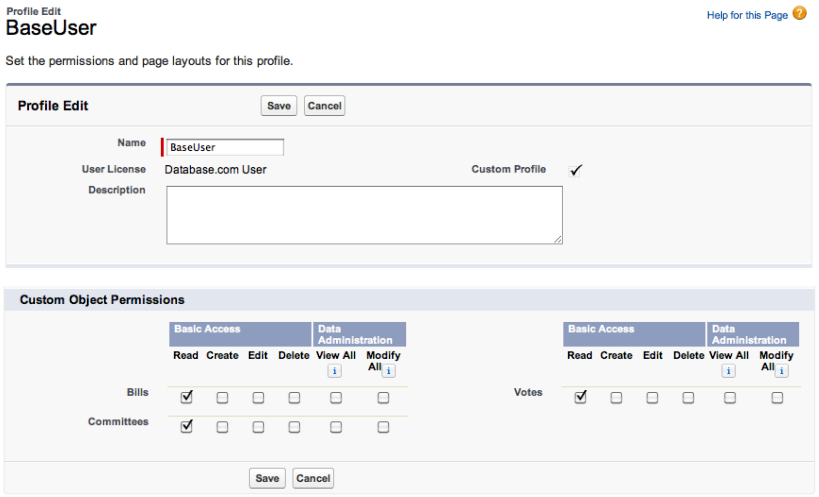
- *field-level access*, including the ability to read or edit fields in objects
- *access to invoke Apex classes and custom logic*

The available permissions you can configure for a profile depend on the *user license* you associate with the profile. For example, system administrator profiles use the *Database.com Admin* license, the only type of license that provides access to powerful, system-level access controls. Non-administrator profiles use the *Database.com User* license or the *Database.com Light* license—the difference between these two user license types is explained later in this paper after introducing some other concepts and features that help to differentiate them.

Example

To get started building our example security policy, you would do the following:

- Create a profile called BaseUser for all types of app users. The profile uses the Database.com User license, permits assigned users to login during normal business hours from IP addresses within the organization's secure network, and permits assigned users to read records in all objects (Bill, Vote, and Committee).
- Create a profile called BaseAdmin for users who will act as administrators of varying types. The profile uses the Database.com Admin license, with permissions that let assigned users login any time of the day or night, from any IP address. That's it. The profile doesn't provide access to any other system-level operations, and it doesn't provide access to any data whatsoever. Remember, least privilege.



Creating profiles is easy using the Database.com Console, as the previous figure illustrates.

Permission sets

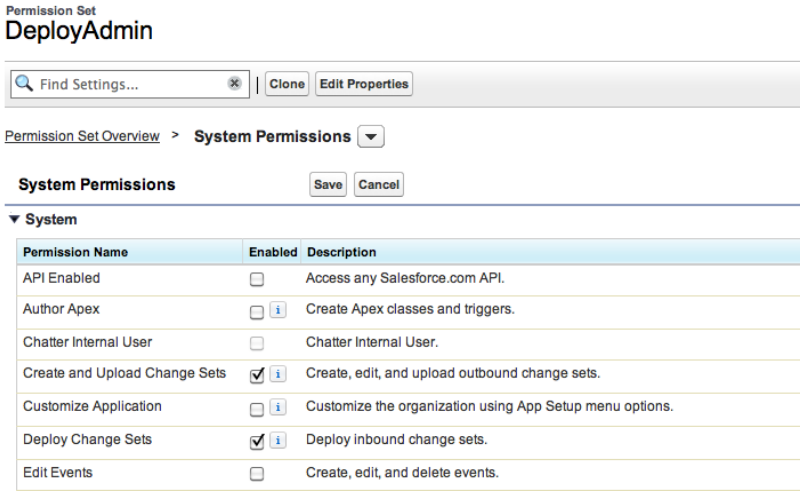
Suppose you need to implement a security policy that has many types of users with similar yet varying privilege requirements. Rather than building and managing many profiles that differ only slightly, it's more efficient to build one profile to manage the common permissions and then use Database.com *permission sets* to manage other specific sets of permissions. Similar to profiles, a permission set's associated user license determines what types of configuration options are available for the permission set.

Example

Continuing with our example, permission sets make it easy to handle the varying requirements for the different types of administrators and users in our org. To handle this scenario, you would create the following permission sets with the Database.com Admin license:

- DeployPerms for BaseAdmins who require the ability to manage change sets

- SecurityPerms for BaseAdmins who require the ability to manage user accounts and transfer record ownership
- BillManager for BaseUsers who require the ability to create and manage bills (legislators)
- BillApprover for the two BaseUsers who can approve bills (the Speaker of the House and the Senate Majority Leader)



- BillSigner for the one BaseUser who can sign bills (the President)

Similar to profiles, it's easy to create permission sets using the Database.com Console.

Users

Once you have profiles and permission sets in place, you can turn your attention to users. Every new Database.com org starts with a super-user administrator account that you can use to manage everything in the organization, including profiles, permission sets, and users.

You can create other user accounts declaratively or programmatically. For example, to get started, you might create administrative users with the Database.com Console. But when you build your app, the app can leverage Database.com's metadata API to let users create their own user accounts from a "Sign Up" page.

Record ownership and sharing

Inherent in the design of Database.com's security model is the notion of *record ownership*, which helps to simplify the implementation of row-level least privilege data security policies. The creator of a record owns the record after creation and has full access—the owner can read, update, delete, and transfer ownership for the record.

Various data access controls determine whether org users can access records they don't own. These controls include an object's sharing model, role hierarchies, groups, and sharing rules.

Sharing models

Each object has a *sharing model*, also known as an *organization-wide default (OWD)*, which governs the default org-wide access levels users have to each other's records in the object.

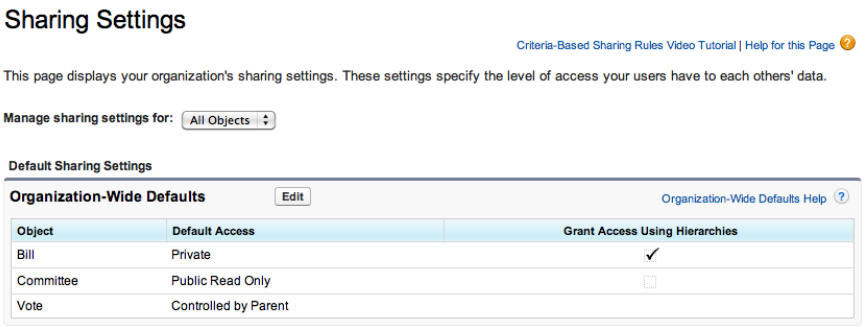
- With an object that uses a *private sharing model*, the record owner can read and manage a record, assuming that the user's profile provides object-level access. Other users can work with records they don't own only by other record sharing means.
- With an object that uses a *public read-only sharing model*, any user can read all records in the object, assuming that the user's profile provides the Read permission and field-level access for the object.

- With an object that uses a *public read/write sharing model*, any user can read and write all records in the object, as permitted by the object- and field-level permissions in each user's profile.

An object can have different sharing requirements based on the user context, so it's very important to consider this fact when setting its OWD. A good rule of thumb is to set each object's OWD to be as strict as necessary for the most strict user requirement, and then use sharing rules to open up access, as required.

Example

Our app has an object called Committee that serves as the basis for a lookup field in the Bill object. It makes sense to give the BaseUser profile the Read permission for the Committee object and then set the Committee object's OWD to public read-only so every BaseUser can read all the records in this lookup table when working with the app.



Now consider the Bill object your app uses to store information about bills. A user should be able to read and manage his/her bills, but may not necessarily want other users to have access to their bills. Thus, we need to set the OWD for Bill to private and then use other controls to share private records with other users.

One more thing: When two objects are in a master-detail relationship, the OWD of the master object determines the record sharing default for the detail object. In other words, a user who can access a master record can automatically access all associated detail records as well. This natural progression of record access makes it easy to manage access to related records because you only have to consider sharing defaults for master objects. In the previous figure, notice how the OWD for the Vote object is "controlled by parent," which is the Bill object.

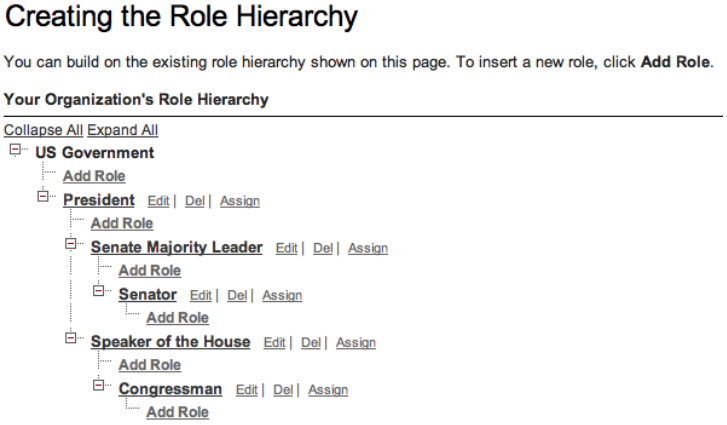
Role hierarchies

With objects that use a private record sharing default, an easy way to share the object's private records with other users is with a role hierarchy. A *role* is a group of users who all live at the same level of an organizational hierarchy in terms of access privilege requirements.

Example

Our app can use the following role hierarchy to easily facilitate shared access to private bills.

Once you implement a role hierarchy with Database.com, private record sharing automatically goes *up* the nodes in the hierarchy, but not *down*. Going up means that House bills are automatically accessible to the Speaker of the House, and then the President, but not necessarily accessible to Senators and the Senate Majority Leader.



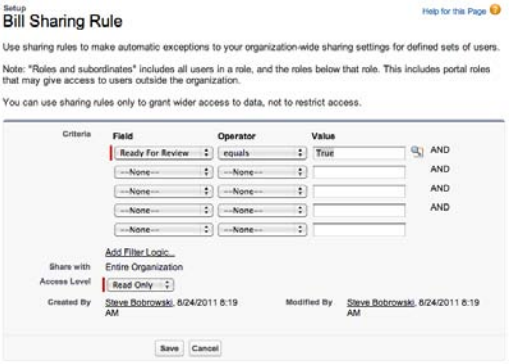
Groups and sharing rules

Organizations often need to share sets of private records that are related by ownership or other criteria with particular users. For such requirements, you can use Database.com *groups*. All that you need to do is create a group and your sharing rules using the Database.com Console with a few mouse clicks.

Examples

For our app, we want to automatically share CRUD access to bills among the members of the same committee. To implement this requirement, we would create a group for every committee in Congress, add users to each group, and then declare a sharing rule for Bill that implements the automatic record sharing to the group. This type of declarative ownership-based sharing rule is easy to create without any code, as the first figure to the right illustrates.

You can also create sharing rules based on criteria. For example, the second figure to the right illustrates a declarative, criteria-based sharing rule that facilitates read access to all bills ready for review.



When declarative sharing rules are not flexible enough to enforce your business rules, you can turn to programmatic, *Apex-managed sharing rules*. This feature lets you share records that correspond to specific business processes.

Note: You can also create sharing rules with other targets, such as roles within a hierarchy.

More about standard and light user licenses

Now that you understand the various Database.com features you can use to control system and data access with Database.com, let's revisit our earlier question: *What's the difference between the Database.com User and Light Database.com User licenses?*

- A standard *Database.com User* license is subject to all Database.com security controls, including authentication, profiles (system and object-level data access controls), and record sharing/row-level data access controls. When you want to leverage Database.com's supported, scalable, fine-grained data access controls for record sharing, use this type of user license for your app users.
- A *Light Database.com User* license is subject to Database.com authentication and profiles, but not to record-level data sharing controls. When your app doesn't require record-level data access controls, or your team is comfortable with building and maintaining such controls in the application itself, then consider lower-cost light users.

In summary, there's a tradeoff between convenience and cost when it comes to license types.

Database.com access control internals

To appreciate more about the implementation of a row-level security system, let's explore the related internals of Database.com. There are several primary components that make Database.com's record sharing possible and scalable.

To begin with, every object has an internal system-managed *sharing table*. A sharing table has a row to keep track of every instance of record sharing in the associated object. In our example app, for a given

record in the Bill object that's owned by a Senator, the associated sharing table contains the following rows:

- A row for the owner of the record, in this case, a Senator
- A row for the Senate Majority Leader role, which extends access to all members of the role
- A row for the President role, which extends access to all members of the role
- Rows that correspond to various sharing rules

As this simple example demonstrates, a sharing table can have many more rows than the corresponding object, depending on the complexity of the security policy and the security controls in use.

When it's necessary to change an object's sharing defaults, change the owner of a record, or any number of other operations, Database.com "recalculates" the sharing records in the corresponding sharing table. Database.com has a bulk processing mechanism that operates asynchronously in the background to make this recalculation process as efficient as possible and not affect ongoing operations.

Besides controlling what rows a given user has access to when using an application, an object's sharing table is also useful to Database.com's internal query optimizer. The optimizer uses each object's sharing table to gain insight into the selectivity of various query execution plans on a per-user basis, and then uses this selectivity factor as criteria for picking the most efficient execution plan for any given query by a given user at runtime. You'll see why this information is important in the next section.

What would it take to implement row-level security yourself?

Hopefully, you now have a full appreciation of Database.com's powerful, built-in record sharing model and its inherent focus on record ownership. But just in case you don't, let's think through what it would take to hypothetically create a similar row-level security system using a traditional relational database.

Traditional relational database management system (RDBMS) *object privileges* let an administrator control object-level access to data structures. For example, most RDBMSs let you grant INSERT, SELECT, UPDATE, and DELETE table privileges to a user that provide the user with the ability to CRUD rows in a table, respectively.

It's important to understand that with a traditional RDBMS, there's no ingrained concept of record ownership. Consequently, object privileges grant CRUD access to *all* rows in a table. So in our example app, every AppUser in the system can manage every bill, vote, and committee because there's no enforcement of row-level security.

To implement a record sharing model similar to the one that comes with Database.com, you'd have to build your own security sub-system to manage data record sharing among users. For example, after programmatically implementing your record sharing system, you might create a sharing table for each base table that keeps track of record shares. You'd have to carefully identify and deal with all edge cases such as situations where the ownership of a record must change. Overall, the process would be arduous, to say the least. And even if you were successful at building the above type of sub-system for managing row-level security, your database's query optimizer wouldn't be aware of the selectivity of rows based on your system's record shares—unless you plan to maintain a forked version of an open source RDBMS, there's no workaround for this shortcoming.

In conclusion, building your own record ownership-based system using a traditional RDBMS can be quite complex and difficult to maintain. With Database.com, all this powerful functionality is included so you can focus on building your app, not a security model.

Summary

Most database application security policies require that system administrators and end users have no more privilege than what is absolutely necessary to perform their specific jobs. Database.com is inherently equipped with a fine-grained security model that makes least privilege security policy implementations easy to establish, maintain, and scale, even in large and complex organizations.