

calculating the permissions for the `user_level` field. In this case, it returns an associative array that sets the `edit` permission to 0. When this is combined with the permissions returned on line 13, it will result in the user having read only access to the `user_level` field.

Cascading Permissions vs Overriding Permissions

There is an important difference between the way that field-level permission rules are *overlaid* on the table permission rules and the way that table permission rules *override* application permission rules. If any non-null permission set is returned for table-level permissions (i.e. from the table delegate's `getPermissions()` method), then these permissions completely override the rules that are defined in the application delegate class. Xataface never even checks with the application delegate class if it finds permissions defined in the table delegate class.

Permissions returned from the `fieldname__permissions()` method (i.e. field level permissions) are merged with the record-level permissions (as returned by the `getPermissions()` method of the table delegate). This means that any permission that is not explicitly defined in the permission set, is just taken from the table level permission set.

Figure 12.10 demonstrates the wrong way to override permissions at the field level. The problem is that the `READ_ONLY()` method grants the permissions that allow users to read information, but it doesn't explicitly disallow the other permissions. It just omits them. Therefore the effective permissions for the `user_level` field in this example would be **ALL** permissions.

Figure 12.11 shows the correct way to override permissions. The difference is that it starts out with the zero vector (i.e. it loads `NO_ACCESS()` as the base level permissions). Then it applies the `READ ONLY` permissions on top. The method used in figure 12.9 is also a perfectly valid way to handle this problem. The key is that permissions returned from the `fieldname__permissions()` method are overlaid on top of the table-level permissions. They do not override them entirely.

12.7.1 Default Field-Permission Rules

Imagine you have a table where you only want a user to be able to edit one field. All of the rest of the fields you would prefer to make read only.

Your first instinct might be to define `READ ONLY` permissions at the table level, and then allow the edit permission only on the single field similar to the strategy depicted in figure 12.12. This won't work because the edit action only checks the table-level permissions to decide if the user is allowed to edit a record to begin with. It never even proceeds to check the individual field permissions if the record only grants read-only permissions. Therefore, if the user tries to access the edit form for a record in this scenario they would just see a "Permission Denied" message.

```

1  <?php
2  class tables_users {
3      function getPermissions($record){
4          return Dataface_PermissionsTool::ALL();
5      }
6      function user_level__permissions($record){
7          return Dataface_PermissionsTool::READ_ONLY();
8      }
9  }

```

Figure 12.10: An example table delegate class that attempts to override the permissions for the user-level field to make them read only. This method does not work because the `READ_ONLY()` method doesn't explicitly set the edit permission to 0. So the edit permission as returned by the `getPermissions()` method will still be used - and it is set to 1.

```

1  <?php
2  class tables_users{
3      function getPermissions($record){
4          return Dataface_PermissionsTool::ALL();
5      }
6      function user_level__permissions($record){
7          $perms = Dataface_PermissionsTool::NO_ACCESS();
8          $perms = array_merge($perms, Dataface_PermissionsTool::READ_ONLY());
9          return $perms;
10     }
11 }

```

Figure 12.11: The correct way to override permissions at the field level. Permissions need to be explicitly disallowed.