WineWizard – “Phase 2 Project”

Design concepts enabling flexible hierarchical structures

Background:

Innomark Consulting has reviewed the new requirements placed on the WineWizard system as a part of the Phase 2 Project. The following is a report from Jay Dean of Innomark on plans to accommodate complex and variable hierarchical structures.

This report is informal and informational only. Until the redesign is complete and has been tested, Innomark cannot address the performance of the overall system nor any element of the design. WineWizard will be responsible for developing the solution to this challenge using the tools, platform, and design of its choosing.

The Challenge:

WineWizard customers organize their users and their organizational roles into hierarchical structures, also called “tree structures” or “trees”. Such structures are similar in many ways but are varied in detail. Some will contain more levels than others or may include alternate roll-ups and other variations. To store all these hierarchies in a single table requires a very adaptable table structure, and query code that can correctly interpret these varied hierarchies.

Recommendation #1: Separate the User and Role concept.

Each user within the WineWizard system is mapped to a specific organizational role Conceptually, a ‘user’ and a ‘role” are related but still separate things and should be stored in independent tables.

The Users table should hold a complete list of application users, along with Company ID and personal information such as Name, UserID, email address, etc. There is no intrinsic hierarchy to this table.

The Roles table (which could be called “Position” or “Organization”) stores organizational roles or job positions, like “Regional VP”, “Wine Dept. Manager”, or “Unit Manager”. These records, also tagged with the Company ID, are naturally hierarchical. For example, Unit Manager roles will typically report to (or “roll-up to”) a Region Manager.

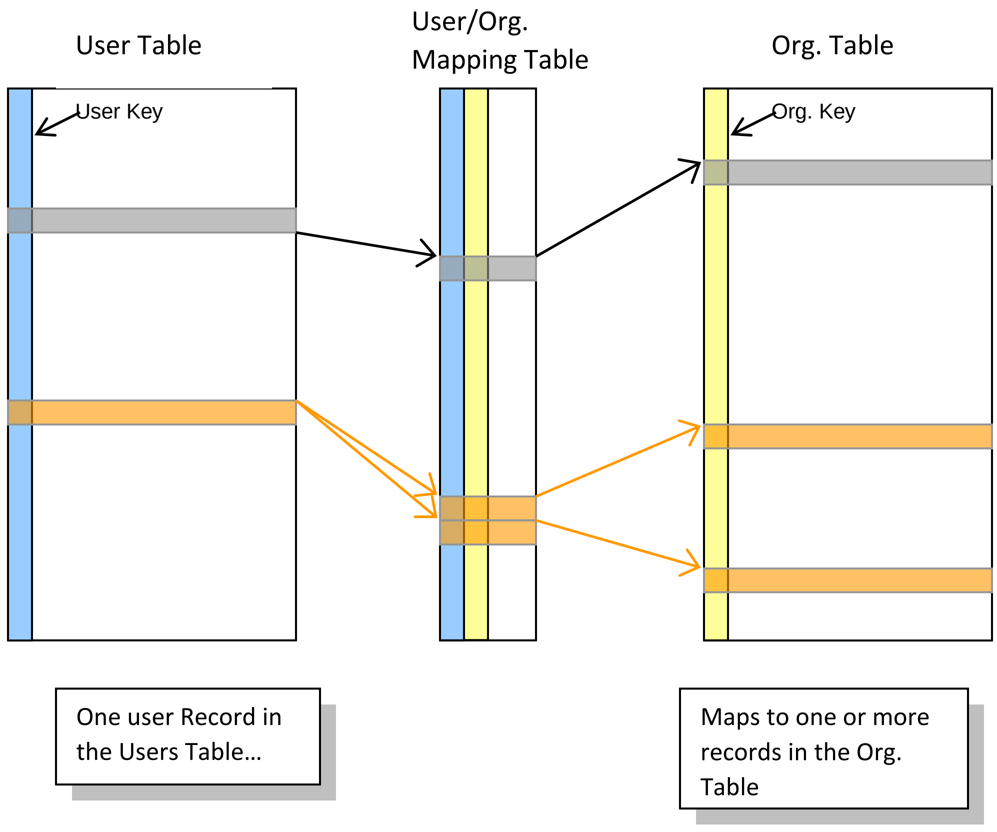
Important: Roles must be linked to a CompanyID

As mentioned above, each record must include the Company ID. This is true throughout the recommended data structure; all records should be linked to a CompanyID (the key field from the Company table).

The application is storing the organization roles for all customers in a single table and must be able to resolve the structure for a specific customer. When this document says “Role” or “Organization” read it as “Company/Role” and “Company/Organization”. **Even if roles in different organizations are similar or use the same title, each role should be separately defined for each CompanyID.**

A User-to-Org Mapping Table

We recommend employing a third table to link the Users and Roles tables. Even if most users are mapped to only a single role, this linking table will allow users to be mapped to more than one role when that is required.


User Table
User/Org.
Mapping Table
Org. Table
User Key
Org. Key
One user Record in the Users Table…
Maps to one or more records in the Org. Table

Recommendation #2: Use a “Parent/Child” table structure for the Organization table (aka the “Roles” table)

A Parent/Child (P/C) table is a very simple and flexible way to store hierarchical information. You will be able to store a variety of hierarchies with different structures in a common table.

The structure of a Parent/Child table is simple. Each element in the hierarchy is a “node”. Each node corresponds to a record in the table. This means that each node, in this case a “Unit” or “Region” or “Dept.”, etc., can be assigned properties, rights and restrictions like any other table record. The *relationship between nodes* is defined by storing, with each node, the key (or ID) of its “Parent” node. (For the moment assume that each node will have only one parent, but each parent can have many children. We’ll cover exceptions below.)

The most basic P/C table has just three fields:

1. Node Key (the unique ID number for the node)
2. Node Description (The name for the node)
3. Parent Key (The ID number for this node’s parent)

The Parent Key is not a Foreign Key

Note that the Parent Key is a reference to the first field in *this* table, not a foreign key to another table. This is what makes a P/C table different; it is a *recursive* table. The power in this structure is that it does not pre-define the generations (or levels) within the hierarchy, the tree structure can extend as many levels are needed. The structure can even be “ragged”, extending more levels in one “branch” than in others.

The lack of pre-defined generations is also the main problem with a P/C structure. It can be hard to know how to query this table and reconstruct the full hierarchy. There are, however, several ways to address this issue.

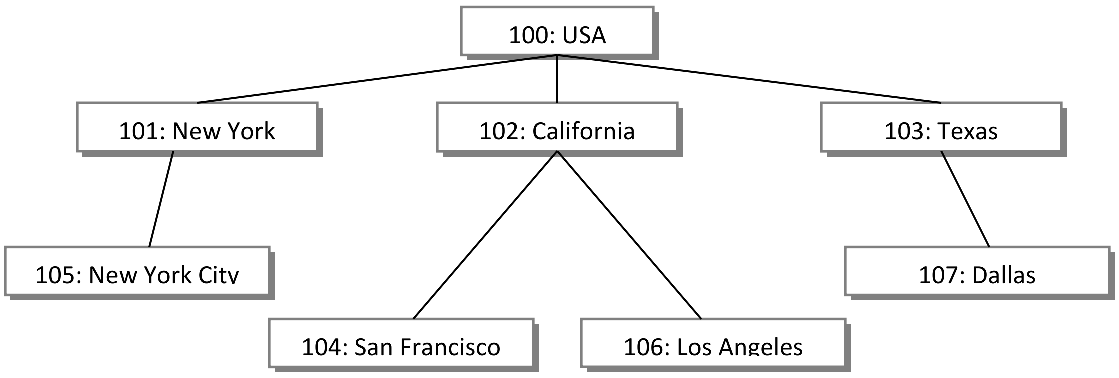
Example P/C table:

Node Key	Node Desc.	Parent Key	Type	Attrib. 2	Attrib. 3
100	USA	*{null}*	Country		
101	New York	100	State		
102	California	100	State		
103	Texas	100	State		
104	San Francisco	102	City		

105	New York City	101	City		
106	Los Angeles	102	City		
107	Dallas	103	City		

The Type field and the other attribute fields are optional parts of the table. If the generations of the hierarchy can be categorized, as they are above, I recommend doing so. While you can define the hierarchy without this field, it is very helpful and will make things much easier.

This table builds this tree:



The terminology used to describe these structures may be based on familial relationships, as in “Child Node”, “Parent Node”, “Ancestors”, or “Descendants”, or tree-based, such as “Root Node”, “Leaf Node”, or “Branch”.

In the illustration above, the tree metaphor is inverted. The Root Node (U.S.A.) is at the top, the Leaf Nodes at the bottom. The leaf nodes *always* represent to lowest level of detail in the structure. “Parent” nodes represent aggregations of lower-level detail nodes, and the Root Node is the top-level aggregation of all the nodes.

Querying the Parent Child Table:

Since the P/C table references itself, we must refer to the same table multiple times in the query while meaning different things. You do this with aliases.

For example, the following query against the sample table above…

```
SELECT  city.NodeKey as CityID, city.NodeName as CityName,
state.NodeName as StateName, country.NodeName as CountryName

FROM   tblGeography city, tblGeography state, tblGeography
country

WHERE city.ParentKey = state.NodeKey AND state.ParentKey =
country.NodeKey AND city.Type = “City”
```

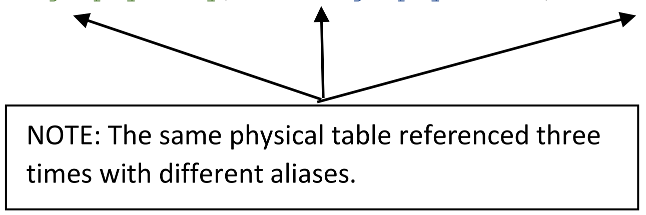
…generates the following result:

CityID	CityName	StateName	CountryName
104	San Francisco	California	U.S.A.
105	New York City	New York	U.S.A.
106	Los Angeles	California	U.S.A.
107	Dallas	Texas	U.S.A.

…which is essentially a traditional generational table.

In this example the same physical table is referenced three times under different aliases. The database will treat each as a separate table.

```
FROM   tblGeography city, tblGeography state, tblGeography country
```



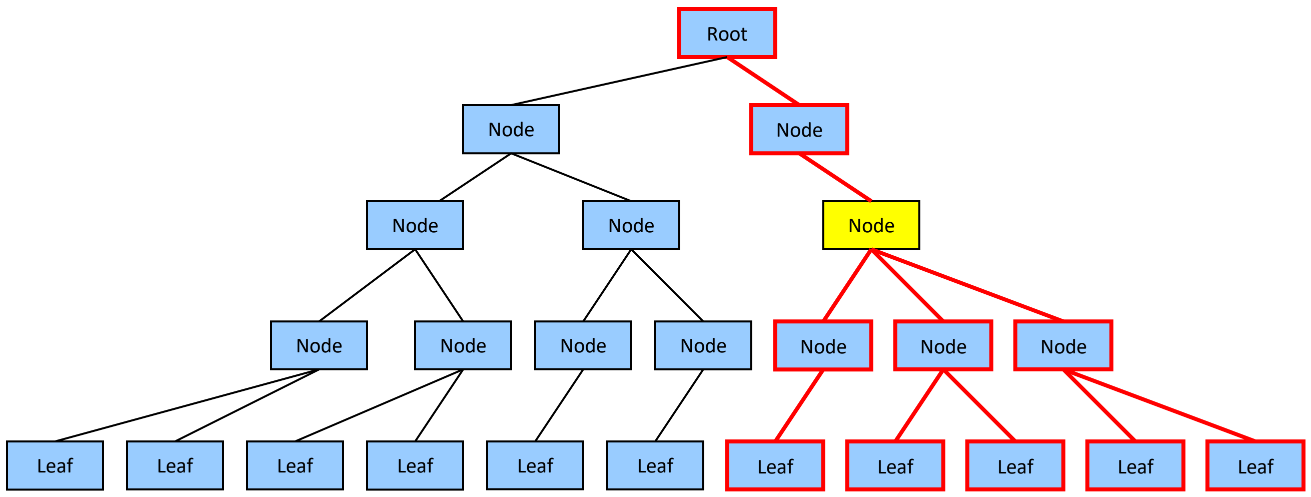
NOTE: The same physical table referenced three times with different aliases.

Data sets you will need to be able to resolve:

Based on our experience with applications using hierarchical structures, you will likely need queries to resolve the following data sets within your application.

For any given Node Key:

1. The name and all other attributes for that node
2. Parent Node
3. All Child Nodes
4. All Ancestors
5. All Descendants
6. The node that is n levels above the specified node
7. Those descendants that are n levels below the specified node
8. All leaf-level descendants below the node



In the above illustration…starting from the yellow node, your application code should be able to identify the nodes outlined in red.

- Information about the specified node itself can be directly queried from the table in a normal manner.
  - `SELECT * FROM tblOrg WHERE NodeId=2023`
- Parent and Child nodes are easily identified though simple queries against the table.
  - `SELECT * FROM tblOrg WHERE ParentId=2023` (returns all children)
  - `SELECT * FROM tblOrg WHERE NodeID=(SELECT ParentId FROM tblID WHERE NodeID=2023)` *(This use of a sub-query is described in a MySQl context in the following article)*
  - http://www.devshed.com/Server_Side/MySQL/MySQL_Subqueries/page1.html

- If the number of generations is known, recursive queries, where the same table links to itself as above, can be exploited. If the number of generations is not known, a query with more joins than are required will usually work. This means that the query asks for more generations than are likely to exist. The "excess" joins and query fields will return nulls.
- The Root Node is distinctive and easy to identify because it does not have a parent (ParentID is null.)
  - One can define a hierarchy with multiple "root nodes". In these cases the root for a specific node can be discovered through iterative queries, to

resolve the complete list of Ancestor nodes. The node in that list with a null value on the ParentKey field is the Root for the specified node.

- Calculating and storing several “helper fields” in the Parent/Child table can dramatically simplify queries and improve performance. Examples include the key for the Root node (when there are multiple roots) or the hierarchy level (number of steps from the root) for each node. Having such fields can be very helpful but introduces the problem of maintenance and accuracy in a frequently updated table.

- The leaf nodes are also distinctive in that they have no children. The easiest way to find them is to filter the query for keys that do not appear in the ParentID field.

  -
    ```
    SELECT * FROM tblOrg WHERE NodeId NOT IN (SELECT
    DISTINCT ParentID From thlOrg)
    ```

All these questions, however, can also be resolved with a series of iterative queries. A solution that uses a layer of programming code (Perl, Java, C#, Python) to unwind the relationships and store the results is often the best option.

This code layer is best organized as a “business object” tier. This set of objects should reflect the business structure of the data rather than the tables and fields of the database storage structure.

The Business Layer / Object Layer

Overview:

There are many ways to implement a Business Layer in a tiered application, with varying benefits, limitations, and implications, so we cannot make a specific recommendation. The optimal solution will be matched to the specific situation and needs of WineWizard, the code platform you choose, and the application complexity you are willing to manage.

It is possible, and some developers prefer, to generate a very elaborate and “pure” n-tier object layer, but this is a lot of work and almost certainly more than you will require. We see no reason to take a purist approach, rather, we suggest adopting basic principals and a simple and consistent approach.

NOTE: All discussion of objects must be adjusted according to the particular object platform used. Object in Perl are subtly different from similar structures in Java and other OOP languages. There are ways, however, to get equivalent functionality in most cases

What do we mean by "Object"?

In this context, an "object" is just a bit of code, organized in a specific way. When we speak of one object calling another object, we are referring to one bit of code referencing another. In some platforms, the way objects reference and call one another is important, and there can be performance costs (and benefits too!) to having many code elements calling across process boundaries.

In a web-based application there are always questions of state and session persistence, and web server performance issues. No doubt you have run into those in your development to-date. A fine-grain object architecture can help, by allowing each specific task to be handled in an optimal way.

A data-access layer, for example, can serve many instances of mid-tier objects, each representing a current user of the application, from the results set of a single query. This can eliminate repetitive queries on the database. Other objects must maintain their state, but others can be made stateless, which can allow better use of server resources.

Background on "n-tier" architecture:

The reason to create a layered or "tiered" architecture is to isolate code around specific tasks. A typical design is a three-tier structure:

- **Top Tier- Presentation Layer**: The top-level tier is the user interface elements of the client application. This code is concerned with display of information and collection of input from the user. This code is aware of interface elements like list-boxes and input fields. If you decide to change the way a particular screen looks and operates, this will be the only tier revised. If you decide to offer multiple user-interfaces, perhaps one for web browsers, one direct to MS Office, and one for mobile devices, you will construct multiple clients connecting to the common mid and bottom tier. The lower tiers are isolated from concerns over the presentation.
- **Mid-Tier – Business Layer**: This code layer encapsulates the business logic within the application. The Presentation Layer will interact with this tier to accomplish business tasks. This is generally where the specific "work" of the application is accomplished. This should be where data is assembled, manipulated, transformed, and otherwise altered.
- **Bottom Tier – Data Storage**: This is the database and associated code; that manages the persistence of data and its recovery. In a very pure tiered design, one could pull out one database platform and insert another and revise only this layer. One can make rather dramatic changes in how data is stored without needing any revision to the business or presentation layers.

Frequently, people refer to “N-Tier” architectures. In practice many applications go beyond strictly three tiers. One can generate many tiers in order to isolate code so that each element is concerned with only one part of the process.

Data Access Layer:

Both the bottom and top tiers are easy to split further into multiple layers. The data storage tier can employ an RDBMS for storage of the data, and a data-access tier to query the DB at the request of the mid-tier, keeping all knowledge of the data storage (including SQL queries) out of the mid-tier. Applications which make use of varied database platforms or data storage technologies usually take this approach.

One might use this layer to hide detail of the actual physical tables. If one is using a linking table as described above, or if one is using some platform specific query syntax, this can be hidden in this data-access layer. Importantly, one can also isolate knowledge of *where* the data is stored. In your business case, it’s very possible that the data server will be relocated or transferred to a cloud platform. If all database connections are established and managed by the data access objects, this can be a relatively painless thing to accomplish.

The data structures returned from this layer become an abstracted data interface that is not linked to any one data storage technique. Rather dramatic changes in the way data are stored and accessed, including splitting the datastore into multiple and heterogeneous physical systems, can be entirely hidden from the upstream tiers.

The Business Layer:

The Business Tier takes in relatively raw, unprocessed data from the data storage tier and passes business-relevant, structured information to the presentation layer. There might be several steps and some complex branching involved in this process, but all that complexity should be isolated within this layer.

Encapsulation:

The business layer objects should use local variables only, that is, local to the object instance. Data is passed into the object through the defined interface (properties and methods of the object), and it goes out the same way.

Breaking the “Rules”

As mentioned above, one can build these object layers very strictly, or take a more expedient approach. The example above takes a few subtle short cuts. The “GetCompleteWineList” method builds a data structure that is specifically designed for presentation needs. The business object is designed around the needs of the user interface. If you change the interface, you might also need to alter the business object.

The purist would have had the OrgRole object offer a WineLists collection, from which the presentation layer could iterate through all the WineLists and extract the needed properties. This solution is very flexible, but the shortcut can work well if you are careful.

The purist approach can also have negative impacts on performance also. Marshalling the data into a single package for delivery is a perfectly acceptable alternative.

A Business Object to handle the WineWizard Organization Tree

The immediate goal of our engagement is to suggest a way to handle the Organizational hierarchy within the WineWizard app. We much prefer to handle hierarchical trees in business objects as described above, querying the Parent/Child table with a series of small and simple queries, rather than try to resolve all the tree structure in one pass.

Two Basic Approaches:

We see two very different ways of handling this tree. One can either retrieve the entire organizational hierarchy and resolve it into a complete tree structure of nested objects (covered in detail later), or get just that part of the hierarchy relevant to a single user. If you expect to need the whole tree structure at some point in a user session, it is best to retrieve it at user login and fill out a complete hierarchy object.

In WineWizard, however, only the org structure for a the one user is displayed, so you should need only a part of the organization tree during any one session. We suggest using the simpler approach (just the piece relevant to the user)

Briefly, the Complete Tree Approach

The complete approach is to create a Hierarchy object, perhaps called "Organization", that has a few properties of its own, but is primarily a container for a collection of "Node" objects (in this specific case we might call the nodes, "Roles".)

As your current codebase is based on Perl, note that in this language collection objects are called 'associative arrays'.

Each node object represents an element in the company organizational structure. It stores in its private variables all the information about that node, and references to other org objects (like its parent).

Within the Node Object:

- NodeKey Property
- NodeName Property
- CompanyID Property

- Hierarchy Property (returns a reference to the containing hierarchy object)
- Parent Property (Reference to Parent Node object)
- Children Collection (Contains child node objects)

The object might also contain:

- Type Property: (“Division, Region, Department, etc.)
- Root Property (Reference to the Root Node Object)
- Level Property (Integer, at what level in the tree is this node?)
- AltParent Property (Reference to an alternate Parent Node (dotted line relationships))
- Other properties like “Franchise”
- Shortcut methods/properties like “GetWineLists”, or “GetAllDescendantKeys”

(From a purist perspective, each node object that is instantiated should store all information in local variables. In practice, we have, at times, left that data in an array within the parent Hierarchy object and then stored a reference in each Node object to retrieve the relevant data when needed. This seems like a shocking violation of OOP principles to some, but it works well in our experience.)

The Partial Tree Approach

Based on our analysis of your business need, our recommended approach is to implement a Partial Tree for organization information.

The application will need a lot of organizational information for only a specific user (the one logged into that session), and needs it in relatively specific ways. One could create a Node object, like that described above, and allow code to explore the org tree starting there, but there’s a simpler approach.

Create a general “OrganizationElement” object, or perhaps, “User/Org” object, that contains all code required to resolve a user’s position within the customer’s company. Here’s how it would work:

1. Once a user is authenticated, instantiate this object, passing it this user’s ID.
2. The object resolves the user’s current role in the company through a simple query joining the User table to the Roles table.
3. Using that, the object can now query the ID’s and names of that user’s parent (boss), children (direct reports), all descendents, or ancestors as required. Another object reference is not required; your presentation-layer code should not be tasked with navigating up and down the object model. The Presentation Layer needs a simple list of Keys and Names for those other parts of the organization.

For example: You ask the object for the Descendents list (because a user can see all lists created by users lower in the org tree), and the object returns a comma separated list of Org ID's and Usernames. This list can be immediately passed to a control in the UI, such as a list box.

One could think of this as a "UserSession" object *(these are simple names to imply the function of the object. There's nothing special in these terms)* that handles all the business logic for that user.

What the page rendering code needs is that list of available Wine Lists (again using the example scenario I used earlier). The shortcut is to have your UserSession object manage all the data for that user, including the set of wine lists, and contain code to make edits, enforce security rules and keep it all straight.

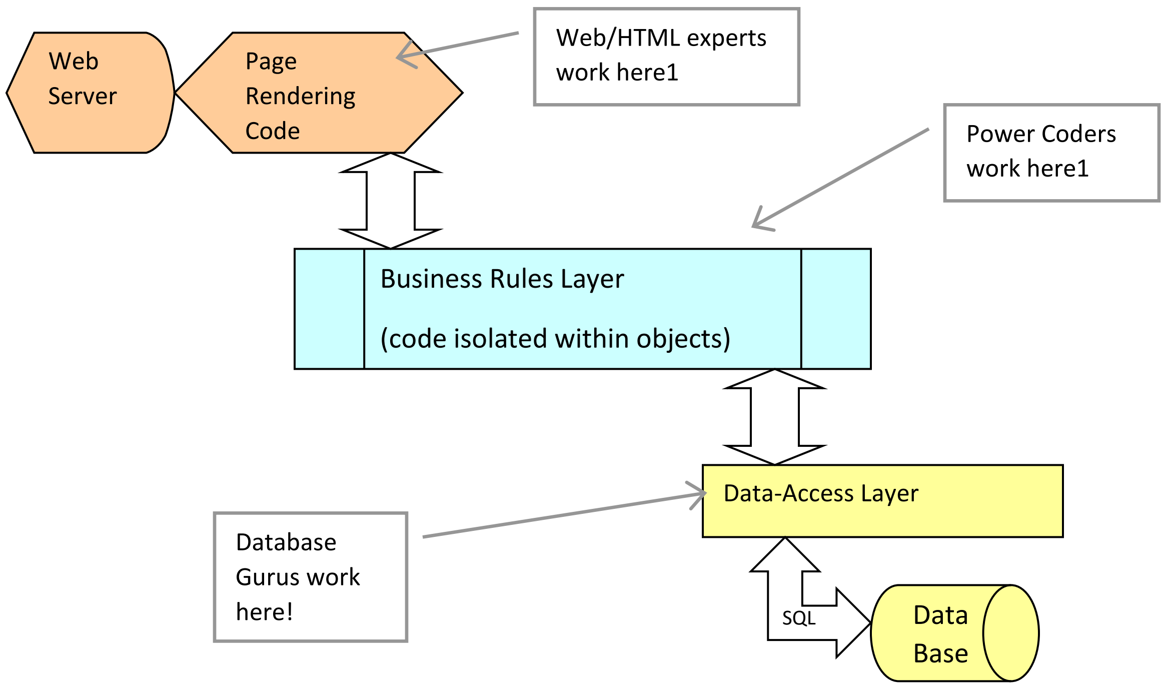
This is far from a "purist approach", but it should be the best approach for WineWizard in the near term.

Returning that list for display might require several separate queries, aimed a several different tables, but the code within the object should handle that.

A Use Case Scenario:

1. User 10025 logs in
2. Application instantiates a new UserSession object, sets UserID property to 10025
3. Page Rendering Code Calls *objUserSession.GetAllWineLists()*
4. UserSession Object gets from database (or data object) all children of User 10025 (Select * Where Parent ID = 10025)
5. UserSession object gets all children of the nodes returned in the previous step.(same query, excerpt WHERE clause is list of IDs returned in previous step)
6. UserSession object keeps extracting children until nothing is returned.
7. UserSession stores the list of children in case it's needed later.
8. UserSession gets from the DB the ID's, Names and other info for all Winelists attached to any of the Org IDs obtained above.
9. UserSession prepares the data and passes it on to the Page Rendering code
10. Page code displays the list to the user for selection.

As mentioned above, this is a “bare-bones” approach, but it keeps elements isolated in important ways. The page code must be written and maintained by someone who knows web page creation and design. The Data Access layer needs to be written and maintained by someone who knows SQL and the database platform. The Business Layer is all code and should be maintained by someone who well understands the business rules and application logic.



Comments on Design Methodology:

A few suggestions on how to apply this idea to your application.

1. The list provided of “the things you should be able to resolve from a given Org ID”, is based on my experience with hierarchical structures. You might not need all that. A good first step is to examine when you need to retrieve stored information based on a user’s organizational position. You should also know which situations generate which requirement. For example, if a user can view any WineList created by themselves or any user lower in the hierarchy, you must resolve the list of users lower in the hierarchy before you can display the user’s personalized list.

2. Look over the app and list all the information you may need to display related to that user. One example is, “the list of all winelists this user can view” (or “can edit, which is possibly a different list.)

3. Think through how you want to receive that information so it can be easily integrated into the web page.
4. Design objects that provide “bundles” of data in convenient packages for the application. Generally, this means that objects should resemble the business elements they represent. In your case, elements like “User”, “Role”, “Winelist”, “TrainingCourse” all seem appropriate as business layer objects.
5. Don’t neglect opportunities to provide a “short-cut” property that eliminates a need to iterate through an extensive object model.
6. Keep your data local. Variables referencing outside the object should refer to another object, *not the data within the other object* (except for the sort of careful violation of this rule I discussed above.) The other object owns its data; other objects get access to it through that object’s interface (and yes, I do cheat at times, but carefully.)
7. Test the objects often, in realist situations
8. Finally, don’t get put off by the jargon and buzzword jungle that surrounds this subject. Remember, an Object” is just a “module” (that plays by object rules, see #6 above.) A “property” is just a “function”, and so on. OOP and N-Tier architecture is just a disciplined way of organizing code and the variables that store session data.