**Multitenancy** refers to a principle in [software architecture](http://en.wikipedia.org/wiki/Software_architecture) where a single instance of the [software](http://en.wikipedia.org/wiki/Computer_software) runs on a server, serving multiple client-organizations (tenants). Multitenancy contrasts with multi-instance architectures where separate software instances (or hardware systems) operate on behalf of different client organizations. With a multitenant architecture, a [software application](http://en.wikipedia.org/wiki/Application_software) is designed to virtually [partition](http://en.wikipedia.org/wiki/Partition_(mainframe)) its data and configuration, and each client organization works with a customized virtual application instance.

 The distinction between the customers is achieved during application design, thus customers do not share or see each other's data.

A multi-tenant architecture, would expect a software application to virtually partition its data and configuration so that each client organization works with a customized virtual application instance.

Economics of multitenancy

### Cost savings

Multitenancy allows for cost savings over and above the basic economies of scale achievable from consolidating IT resources into a single operation.[[6]](http://en.wikipedia.org/wiki/Multitenancy#cite_note-6) An application instance usually incurs a certain amount of memory and processing overhead which can be substantial when multiplied by many customers, especially if the customers are small. Multitenancy reduces this overhead by amortizing it over many customers.

### Release management

With the multitenant model, the package typically only needs to be installed on a single server. This greatly simplifies the release management process.

At the same time, multitenancy increases the risk of applying a new release version and the effects of that. As there is a single software instance serving multiple tenants, an update on this instance may cause downtime for all tenants even if the update is requested and useful for only one tenant.

Requirements

### Customization

* Branding: allowing each organization to customize the look-and-feel of the application to match their corporate branding (often referred to as a distinct "skin").
* Workflow: accommodating differences in workflow to be used by a wide range of potential customers.
* Extensions to the data model: supporting an extensible data model to give customers the ability to customize the data elements managed by the application to meet their specific needs.
* Access control: letting each client organization independently customize access rights and restrictions for each user.

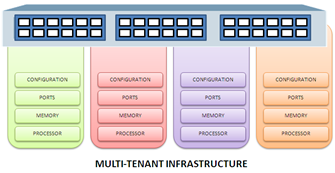
### Quality of service

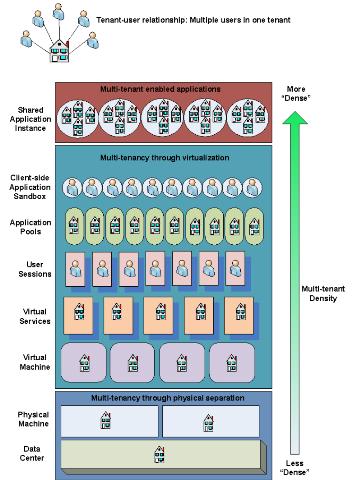
Multitenant applications are expected to provide adequate levels of security and robustness, which are provided by the operating system in the case of multi-instance applications.

In the case of SaaS (Software as a Service) multi-tenancy is almost always achieved via a database and configuration, with isolation provided at the application layer. This form of multi-tenancy is the easiest to implement and is a well-understood model of isolation.

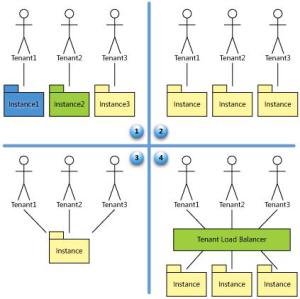
In the case of IaaS (Infrastructure as a Service) this level of isolation is primarily achieved through server virtualization and configuration, but generally does not yet extend throughout the datacenter to resources other than compute or storage. This means the network components do not easily support the notion of multi-tenancy. This is because the infrastructure itself is only somewhat multi-tenant capable, with varying degrees of isolation and provisioning of resources possible. [load balancing](http://www.f5.com/glossary/load-balancing.html) solutions, for example, support multi-tenancy inherently through virtualization of applications, i.e. Virtual IP Addresses or Virtual Servers, but that multi-tenancy does not go as “deep” as some might like or require. This model does support configuration-based multi-tenancy because each Virtual Server can have its own load balancing profiles and peculiar configuration, but generally speaking it does not support the assignment of CPU and memory resources at the Virtual Server layer.

One of the ways to “get around this” is believed to be the virtualization of the solution as part of the hardware solution. In other words, the hardware solution acts more like a physical server that can be partitioned via internal, virtualized instances of the solution**\***. That’s a lot harder than it sounds to implement for the vendor given constraints on custom hardware integration and adds a lot of overhead that can negatively impact the performance of the device. Also problematic is scalability, as this approach inherently limits the number of customers that can be supported on a single device. If you’re only supporting ten customers this isn’t a problem, but if you’re supporting ten thousand customers, this model would require many, many more hardware devices to implement. Taking this approach would drastically slow down the provisioning process, too, and impact the “on-demand” nature of the offering due to acquisition time in the event that all available resources have already been provisioned.





**Maturity model for Multi-Tenancy**

[](http://vikashazrati.files.wordpress.com/2008/06/multi-tenancy3.jpg)

* Ad Hoc/Custom- At this level, each customer has its own customized version of the hosted application, and runs its own instance of the application on the host’s servers.
* Configurable- the vendor hosts a separate instance of the application for each customer (or tenant), vendor meets customers’ needs by providing detailed configuration options that allow the customer to change how the application looks and behaves to its users.
* Configurable, Multi-Tenant-Efficient – the vendor runs a single instance that serves every customer, with configurable metadata providing a unique user experience and feature set for each one.  
  Authorization and security policies ensure that each customer’s data is kept separate from that of other customers.  
  A significant disadvantage of this approach is that the scalability of the application is limited. Unless partitioning is used to manage database performance, the application can be scaled only by moving it to a more powerful server (scaling up)
* Scalable, Configurable, Multi-Tenant-Efficient- the vendor hosts multiple customers on a load-balanced farm of identical instances, with each customer’s data kept separate, and with configurable metadata providing a unique user experience and feature set for each customer.  
  Appropriate data partitioning, stateless component design, shared metadata access are part of the design.
* Customization of application behavior, rather than being done by modifying application code, is done instead by configuring application metadata that is invoked on a user-by-user basis
* Instead of customizing the application in the traditional sense, then, each customer uses metadata to configure the way the application appears and behaves for its users.  
  The challenge for the architect is to ensure that the task of configuring applications is simple and easy for the customers, without incurring extra development or operation costs for each configuration.
* Metadata representation is extended to the deepest levels of database definition, custom business logic and fundamental design of user interface appearance and behavior.
* Data partitioning is most important. Is the data partitioned per tenant.
* Strong separation between the application logic and their datasets.
* Instead of hard-coding data tables and page layouts, developers on a multi-tenant platform define attributes and behaviors as metadata, which functions as the application’s logical blueprint.

**Supporting Multi-Tenancy at Presentation Layer**

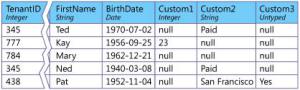
* Presentation services of a multi-tenant platform: Users of Tenant get dedicated entry point with unique branding.
* Tenant isolation ensures that the user populations of the individual tenant cannot access the business service and information of the other tenants of the platform.
* Branding ensures that when an end user accesses a channel destination or authenticates with a channel, the end user is presented with a look and feel specific to his or her subscriber.
* All presentation components must support tenant-specific settings in a range from name-and-value pair configurations, such as tenant IDs, tenant-specific look-and-feel, with settings to indicate which form fields or action buttons should be rendered to end-users.

**Supporting Multi-Tenancy at Business Layer**

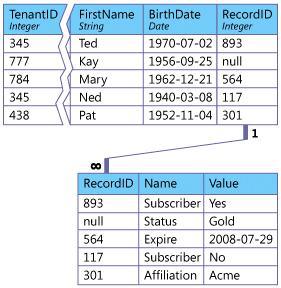
* Each tenant of a multi-tenant platform is identified by a unique, platform-wide tenant ID.  
  The implementation of the services at the business layer must use the tenant ID to provide variability for service subscribers and individual end-users.  
  How?
* Metadata: Features of an application which can be configured during runtime are defined in the form of Metadata
* Metadata Services: A set of programmatically accessible services are designed that provides an interface to access/modify the metadata.
* An Engine: The business layer of the application is an engine that runs the business processes defined on the basis of metadata.

**Supporting Multi-Tenancy at Data Layer**

* Dedicated Tenant Database
* Shared Database, fixed extension set

[](http://vikashazrati.files.wordpress.com/2008/06/multi-tenancy4.jpg)

* Shared Database, custom extension set (another table with key value pairs)

[](http://vikashazrati.files.wordpress.com/2008/06/multi-tenancy5.jpg)

**Concerns with Multi-Tenancy**

1. Security and inappropriate data access
2. Impact of other clients on their system performance
3. Inability to establish and pay for their own, higher service levels (common among large companies)
4. Being forced into an upgrade
5. Inability to support the level of client specific configurations and even customizations as necessary
6. Inability to obtain client level user acceptance testing
7. Inability to determine the precise production/go live dates for the system.
8. If the number of users in a tenant are large, e.g. >1000 users then the multi-tenant advantages start losing steam as they may need a separate hosting with separate maintenance etc.

 in order to take advantage of the benefits of SaaS, an organization must surrender a level of control over its own data, trusting the SaaS vendor to keep it safe and away from prying eyes

we'll look at the continuum between isolated data and shared data, and identify three distinct approaches for creating data architectures that fall at different places along the continuum. Next, we'll explore some of the technical and business factors to consider when deciding which approach to use. Finally, we'll present design patterns for ensuring security, creating an extensible data model, and scaling the data infrastructure.

**Three Approaches to Managing Multi-Tenant Data**



1. Separate Databases

Computing resources and application code are generally shared between all the tenants on a server, but each tenant has its own set of data that remains logically isolated from data that belongs to all other tenants. Metadata associates each database with the correct tenant, and database security prevents any tenant from accidentally or maliciously accessing other tenants' data.

(**D**) The "premium" approach, and the relatively high hardware and maintenance requirements and costs make it appropriate for customers that are willing to pay extra for added security and customizability. For example, customers in fields such as banking or medical records management often have very strong data isolation requirements, and may not even consider an application that does not supply each tenant with its own individual database.

1. Shared Database, Separate Schemas

Another approach involves housing multiple tenants in the same database, with each tenant having its own set of tables that are grouped into a schema created specifically for the tenant.

When a customer first subscribes to the service, the provisioning subsystem creates a discrete set of tables for the tenant and associates it with the tenant's own schema.

|  |
| --- |
| --- create a schema and authorize a user account to access it.  CREATE SCHEMA ContosoSchema AUTHORIZATION Contoso  --- create and access tables within the tenant's schema using the **SchemaName.TableName** convention  CREATE TABLE ContosoSchema.Resumes (EmployeeID int identity primary key, Resume nvarchar(MAX))  --- set as the default schema for the tenant account  ALTER USER Contoso WITH DEFAULT\_SCHEMA = ContosoSchema |

A tenant account can access tables within its default schema by specifying just the table name, instead of using the **SchemaName.TableName** convention. This way, a single set of SQL statements can be created for all tenants, which each tenant can use to access its own data

Tables are created from a standard default set, but once they are created they no longer need to conform to the default set, and tenants may add or modify columns and even tables as desired.

(**D**) Data is harder to restore in the event of a failure. If each tenant has its own database, restoring a single tenant's data means simply restoring the database from the most recent backup. With a separate-schema application, restoring the entire database would mean overwriting the data of every tenant on the same database with backup data, regardless of whether each one has experienced any loss or not. Therefore, to restore a single customer's data, the database administrator may have to restore the database to a temporary server, and then import the customer's tables into the production server.

(**A**) Appropriate for applications that use a relatively small number of database tables, on the order of about 100 tables per tenant or fewer.

1. Shared Database, Shared Schema

Using the same database and the same set of tables to host multiple tenants' data. A given table can include records from multiple tenants stored in any order; a Tenant ID column associates every record with the appropriate tenant.

(**A**)Has the lowest hardware and backup costs, because it allows you to serve the largest number of tenants per database server.

(**D**) However, because multiple tenants share the same database tables, this approach may incur additional development effort in the area of security.

**Choosing an Approach**

Economic Considerations

Applications optimized for a shared approach tend to require a larger development effort than applications designed using a more isolated approach (because of the relative complexity of developing a shared architecture), resulting in higher initial costs. Because they can support more tenants per server, however, their ongoing operational costs tend to be lower.

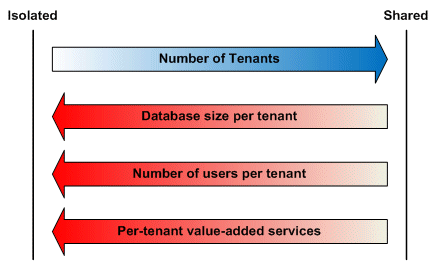
Security Considerations

A common misconception holds that only physical isolation can provide an appropriate level of security. In fact, data stored using a shared approach can also provide strong data safety, but requires the use of more sophisticated design patterns.

Tenant Considerations

The number, nature, and needs of tenants

* The larger you expect your tenant base to be, the more likely you will want to consider a more shared approach.
* If you expect some or all tenants to store very large amounts of data, the separate-database approach is probably best.
* The larger the number of concurrent end users, the more appropriate a more isolated approach will be to meet end-user requirements.
* Any per-tenant value-added services, such as per-tenant backup and restore capability services are easier to offer through a more isolated approach.



Regulatory Considerations

Companies, organizations, and governments are often subject to regulatory law that can affect their security and record storage needs.

## Realizing Multi-Tenant Data Architecture

### Security Patterns

A secure SaaS application is one that provides defense in depth, using multiple defense levels that complement one another to provide data protection in different ways, under different circumstances, against both internal and external threats.

* **Filtering:** Using an intermediary layer between a tenant and a data source that acts like a sieve, making it appear to the tenant as though its data is the only data in the database.
* **Permissions:** Using access control lists (ACLs) to determine who can access data in the application and what they can do with it.
* **Encryption:** Obscuring every tenant's critical data so that it will remain inaccessible to unauthorized parties even if they come into possession of it.

Trusted Database Connections

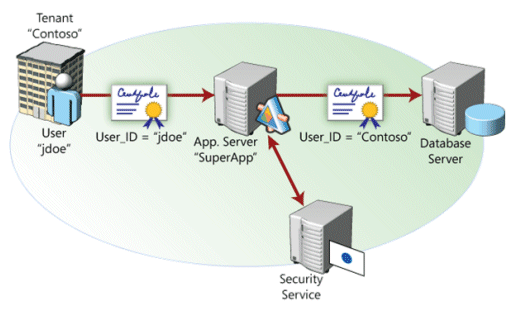
In a multi-tier application environment application architects traditionally use two methods to secure access to data stored in databases: impersonation, and a trusted subsystem account.

With the *impersonation* access method, the database is set up to allow individual users to access different tables, views, queries, stored procedures, and other database objects.

With the *trusted subsystem access* method, the application always connects to the database using its own application process identity, independent of the identity of the user; the server then grants the application access to the database objects that the application can read or manipulate. Any additional security must be implemented within the application itself to prevent individual end users from accessing any database objects that should not be exposed to them. This approach makes security management easier, eliminating the need to configure access to database objects on a per-user basis, but it means giving up the ability to secure database objects for individual users.

The tenant is an organization that uses the application to access its own data store, which is logically isolated from data stores belonging to any other tenants. Each tenant grants access to the application to one or more end users, allowing them to access some portion of the tenant's data using end user accounts controlled by the tenant.

In this scenario, you can use a hybrid approach to data access that combines aspects of both the impersonation and trusted subsystem access methods.



The database server does not distinguish between requests originating from different end users associated with the same tenant, and grants all such requests access to the tenant's data. Within the application itself, security code prevents end users from receiving and modifying any data that they are not entitled to access.  The application must *intercept* the query results and only present the user with the records that she is entitled to see.

Secure Database Tables

To secure a database on the table level, use SQL's GRANT command to grant a tenant user account access to a table or other database object.

If you use the hybrid approach to database access discussed earlier, in which end users are associated with the security contexts of their respective tenants, this only needs to be done once, during the tenant provisioning process; any end user accounts created by the tenant will be able to access the table.

This pattern is appropriate for use with the separate-database and separate-schema approaches. In the separate-database approach, you can isolate data by simply restricting access on a database-wide level to the tenant associated with that database, although you can also use this pattern on the table level to create another layer of security.

Tenant View Filter

SQL views can be used to grant individual tenants access to some of the rows in a given table, while preventing them from accessing other rows.

SELECT \* FROM Employees WHERE TenantID = SUSER\_SID()

This statement obtains the security identifier (SID) of the user account accessing the database (which, you'll recall, is an account belonging to the tenant, not the end user) and uses it to determine which rows should be included in the view. (The example assumes that the unique tenant ID number is identical to the tenant's SID. If this is not the case, one or more additional steps would be required to associate each tenant with the correct rows.) Each individual tenant's data access account would be granted permission to use the **TenantEmployees** view, but granted no permissions to the **Employees**source table itself. You can build queries and shared procedures to take advantage of views, which provides tenants with the appearance of data isolation even within a multi-tenant database.

This pattern is slightly more complex than the Secure Database Tables pattern, but is an appropriate way to secure tenant data in a shared-schema application, in which multiple tenants share the same set of tables.

Tenant Data Encryption

A way to further protect tenant data is by encrypting it within the database, so that data will remain secure even if it falls into the wrong hands.

Public-key cryptography requires significantly more computing power than symmetric cryptography. A better approach is to use a key wrapping system that combines the advantages of both systems.

With this approach, three keys are created for each tenant as part of the provisioning process: a symmetric key and an asymmetric key pair consisting of a public key and a private key. The more-efficient symmetric key is used to encrypt the tenant's critical data for storage. To add another layer of security, a public/private key pair is used to encrypt and decrypt the symmetric key, to keep it secure from any potential interlopers.

When an end user logs on, the application uses impersonation to access the database using the tenant's security context, which grants the application process access to the tenant's private key. The application (still impersonating the tenant, of course) can then use the tenant's private key to decrypt the tenant's symmetric key and use it to read and write data.

(**A**) The first line of defense, at the database level, prevents end users from accessing the private data of other tenants. If a bug or a virus in the database server were to cause an incorrect row to be delivered to the tenant, the encrypted contents of the row would be useless without access to the tenant's private key.

(**D**) Because you can't index encrypted columns, selecting which columns of which tables to encrypt involves making a tradeoff between data security and performance.

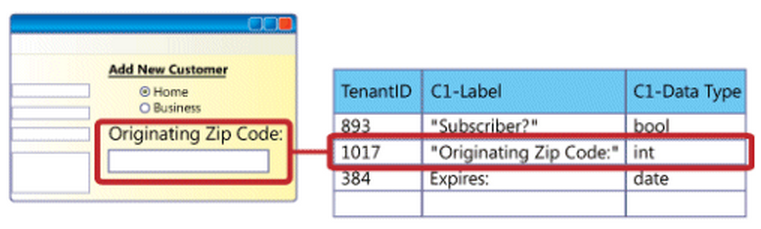
Extensibility Patterns

As designed, your application will naturally include a standard database setup, with default tables, fields, queries, and relationships that are appropriate to the nature of your solution. But different organizations have their own unique needs that a rigid, inextensible default data model won't be able to address.

* Preallocated Fields

One way to make your data model extensible is to simply create a preset number of custom fields in every table you wish to allow tenants to extend.

 Records from different customers are intermingled in a single table; a tenant ID field associates each record with an individual tenant. In addition to the standard set of fields, a number of custom fields are provided, and each customer can choose what to use these fields for and how data will be collected for them. Use the string data type for every custom field, and use metadata to track the "real" data type the tenant wishes to use.



When creating the text box, the tenant used validation logic (not shown) to require that the text box contain an integer. As implemented, this custom field is defined by a record in a metadata table that includes the tenant's unique ID number (1017), the label the tenant has chosen for the field ("Originating ZIP Code"), and the data type the tenant wants to use for the field ("int").

When an end user types a quantity into the field and saves the record, the application casts the value for Originating ZIP Code to a string before creating or updating the record in the database. Whenever the application retrieves the record, it checks the metadata table for the data type to use and casts the value in the custom field back to its original type.

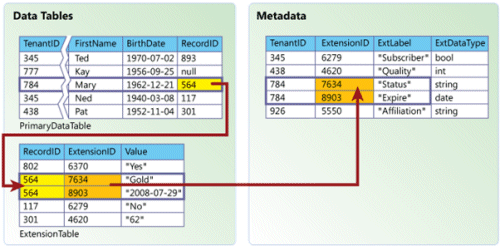
* Name-Value Pairs

Allow customers to extend the data model arbitrarily, storing custom data in a separate table and using metadata to define labels and data types for each tenant's custom fields.

Here, a metadata table stores important information about every custom field defined by every tenant, including the field's name (label) and data type. When an end user saves a record with a custom field, two things happen. First, the record itself is created or updated in the primary data table; values are saved for all of the predefined fields, but not the custom field. Instead, the application creates a unique identifier for the record and saves it in the Record ID field. Second, a new row is created in the extension table that contains the following pieces of information:

* + The ID of the associated record in the primary data table.
  + The extension ID associated with the correct custom field definition.
  + The value of the custom field in the record that's being saved, cast to a string.

This approach allows each tenant to create as many custom fields as necessary to meet its business needs. When the application retrieves a customer record, it performs a lookup in the extension table, selects all rows corresponding to the record ID, and returns a value for each custom field used. To associate these values with the correct custom fields and cast them to the correct data types, the application looks up the custom field information in metadata using the extension IDs associated with each value from the extension table.



#### Using Data Model Extensions

Whatever method you use to create an extensible data model, it must be paired with a mechanism for integrating the additional fields into the application's functionality. Any custom field implemented by a customer will require a corresponding modification to the business logic (so the application can use the custom data), the presentation logic (so that users have a way to enter the custom data as input and receive it as output), or both. The configuration interface you present to the customer should therefore provide ways to modify all three, preferably in an integrated fashion.

Scalability Patterns

#### Scaling Techniques

The two main tools to use when scaling out a database out are replication and partitioning. Replication involves copying all or part of a database to another location, and then keeping the copy or copies synchronized with the original.

Partitioning involves pruning subsets of the data from a database and moving the pruned data to other databases or other tables in the same database. You can partition a database by relocating whole tables, or by splitting one or more tables up into smaller tables: horizontally or vertically. Horizontal partitioning means that the database is divided into two or more smaller databases using the same schema and structure, but with fewer rows in each table. Vertical partitioning means that one or more individual tables are divided into smaller tables with the same number of rows, but with each table containing a subset of the columns from the original.

* Tenant-Based Horizontal Partitioning

A shared database should be scaled when it can no longer meet baseline performance metrics, as when too many users are trying to access the database concurrently or the size of the database is causing queries and updates to take too long to execute, or when operational maintenance tasks start to affect data availability.

The simplest way to scaleout a shared database is through horizontal (row-based) partitioning based on tenant ID.

If you're experiencing application performance problems because too many end users are accessing the database concurrently, consider partitioning the database to equalize the total number of active end-user accounts on each server. For example, if your existing database serves tenants A and B with 600 active users each, and tenants C, D, and E with 400 active users each, you could partition the database by moving tenants C, D, and E to a new server; both databases would then serve 1200 users each.

If you're experiencing problems relating to the size of the database, such as the length of time it takes to perform queries, a more effective partition method might be to target database size instead, assigning tenants to database servers in such a way as to roughly equalize the amount of data on each one.

* Single Tenant Scaleout

If some or all tenants store and use a large amount of data, tenant databases may grow large enough to justify devoting an entire server to a single database that serves a single tenant. If the database continues to grow, eventually it will no longer be cost-effective to move it to a more powerful server, and you will have to scale out by partitioning the database on to one or more additional servers. Scaling out a dedicated database is different than scaling out a shared one. With a shared database, the most effective method of scaling involves moving entire sets of tenant data from one database to another, so the nature of the data model that you use isn't particularly relevant. When scaling a database that's dedicated to a single tenant, it becomes necessary to analyze the kinds of data that are being stored to determine the best approach.

Scale out guidelines to consider:

* **Use replication to create read-only copies of data that doesn't change very often.** Some kinds of data rarely or never change after the data is entered. Other kinds of data are subject to active change for a defined period of time and then archived. These kinds of data are ideal candidates for one-way replication to any databases from which they might be referenced.
* **Location, location, location.** Keep data close to other data that references it. Consider the relationships between different kinds of data when deciding whether to separate them, and use replication to distribute read-only copies of reference data among different databases when appropriate.Try to find natural divisions in the data that will minimize the amount of cross-database communication that needs to take place.
* **Identify data that shouldn't be partitioned.** Resource data, are usually poor candidates for replication or partitioning. Use scaleout techniques to move other data off the server, leaving your resource data more room to grow. If you have moved all the data you can and still experience problems, consider scaling up to a bigger server for the resource data.

# Multi-Tenant Strategy for FIS System

## Considerations

### Security

* Being financial institutes, the clients will have very strict restrictions. Absolute no chance of mingling each other data has to be possible.
* Defense-in-depth has to be implemented, so checks have to be at multiple levels and not just at specific points.
* Each client can ask for a specific authentication and/or authorization mechanism to implement. Hence the architecture has to allow for support from any technology being followed while allowing the application requirements to be met.
* Once a user request is inside the active system, the credential will describe { tenant, task, claim }. The claim will be used to authorize the action to be performed. Any other information about the user will not be stored within the application boundary, but can be used to display breadcrumb information on application.

### Deployment

* The base system consisting of user application, data-system and configuration will remain same for all clients.
* Any optimization should be able to be performed tenant-wise. This includes partitioning, clustering, scale-out and scale-up and backup operations. Any strategy which hampers per-client task performance needs to be specifically handled to take appropriate steps to ensure that other clients are not affected by the operation.

### Customization

* Any client can demand different types of customization which will not be applied to other clients. So client specific meta-data has to be applied.

## Informal Notes:

### Security:

User will need to authenticate using the format <username>@<clientName> format. The clientName will be used to create the appropriate route for the client and use the correct strategy/factory as mandated by the client.

User gets authenticated from client’s own framework by redirecting from app’s login page. The authentication result must have appropriate claim from the client’s end.

Claim Structure: The claim will have the tenant token, generalUser / adminUser. The only other user information retrieved will be “Display Name” as chosen by user during registration with the specific client. This name will be indicative and may not indicate actual name. This will be just used for breadcrumb customization purpose and not persisted in application.

App audit will be done using a structure of datetime in UTC format, tenant token. This will not have any actual user specific information.

1. User from Starbucks requests the app through Starbucks corporate site. Url used: fis.com/prepaid/client/starbucks
2. Security Interceptor detects total absence of authentication token.
   1. Identifies client through Url parsing
   2. Loads authentication service abstract factory and asks for client specific authentication factory.
   3. Abstract factory checks mapping corresponding to Starbucks and returns OAuth as the method of authentication as demanded by Starbucks and redirects to the corresponding url.
3. User provides appropriate credentials. This can include username, password, mobile code, captcha etc as claim components.
4. The system does internal validation of user’s claim and return Boolean status and auth-token if successful. The token will contain user’s claim collection, user’s display name and validity criteria. This token will be sent to Security Interceptor.
5. Security Interceptor will create its own token containing client identifier (starbucks) and a hash code of user’s claim (end-user, admin, backup etc). It will store this hash and its mapped claim in its dictionary with validity set as per client. The hash will be returned back to request along with client identifier
6. The request reaches the app.
7. The app identifies the client being serviced through the client identifier token. The presence of token signifies authentication. App loads the application page along with styles and scripts for client. The user’s display name is added to breadcrumb (if present) and then removed from scope.
8. Audit is done through auth token and timestamp.

References: <http://msdn.microsoft.com/en-us/library/aa479086.aspx>