White Paper

Non Functional Requirements Check List

- Ramkumar R S

techcello

Contents

Introduction....4

Functional Vs Non Functional Requirements (NFRs)....4

Why NFRs are more important and critical? 4

1.0 Multi-tenant Application Architecture....5

1.1 Multi-tenant Application Architecture vs Multi-tenant Database Architecture....5

1.2 Support for all levels of Multi-tenancy....5

1.3 Who is a Tenant ?....5

2.0 Scalability....5

2.1 Data Connection Abstraction....5

2.2 Data Isolation....6

2.3 Distributed Caching and Stateless Design....6

3.0 Customizability and Configurability by Non IT Personnel....6

3.1 Themes and Logos 6

3.2 Associating CSS files to Tenants 6

3.3 Forms and Grids....6

3.4 Data model extension....7

3.4.1 Custom fields on Forms and Grids....7

3.4.2 Custom fields on Queries and Reports....7

3.4.3 Custom fields on Business Rules and Workflows....7

3.5 Business Rules Customization....7

3.6 Workflow Customization....8

3.7 Report Customization....8

3.8 Notification Template Customization....8

4.0 Configurable Access Control System..8

4.1 Tenant specific roles and role – privilege mapping..9

4.2 Dynamic Data scope policies..9

4.3 Single Sign on..9

4.4 Tenant – Sub tenant Hierarchy..9

4.5 Approval management for certain types of users and tenants..10

4.6 Tenant Configuration Templates..10

5.0 Other NFRs..10

5.1 Notification..10

5.2 Schedulers..10

5.3 Auditing..10

5.4 Metering 10

5.5 Usage Auditing..10

5.6 Module & Feature provisioning..11

5.7 Pre and Post Processors and Policy Injection..11

5.8 Logging and Exception Management..11

5.9 Master data and Pick up lists Management..11

6.0 NFR Stack Design..12

A Typical NFR Stack / Framework for a Multi-tenant SaaS Application..13

Abstract – Summary..14

About the Author..14

Introduction

An application that addresses a diverse set of stakeholders and multiple levels of user groups is more complicated and technically challenging than an application that is focused on a niche homogenous user base. The Non functional requirements (NFRs) of such applications are hence very critical and important.

Defining the NFRs independent of the functional requirements has many benefits. It leads to a focused effort on NFRs, a robust architectural framework, engineering excellence and consistency. The NFR stack when implemented as a loosely coupled framework, could also become a common stack for multiple applications / products, resulting in significant reduction in cost of development, support and maintenance (TCO).

A NFR Stack should contain all the plumbing, engineering and operational components in a ready to use form (APIs and Services), so that developers using them can focus only on building the application specific functionalities. While building or buying such a framework, care should be taken that the application owner / provider retains ownership and control over the entire application stack both technically and strategically. Vendor lock in or platform lock in should be avoided.

Functional Vs Non Functional Requirements (NFRs)

While building any application, the focus is often on the Functional requirements. These are capabilities and features that the users expect from the application. The functional requirements could vary from one group of users to the other and from one domain to the other. There can be multiple applications and multiple modules within an application to cater to various usage scenarios.

However, there are non-functional requirements for any application, many of which are latent and implicit. Some of the NFRs could be so common sense, one might wonder, whether it is really required to explicitly articulate it.

Why NFRs are more important and critical?

Let us take a simple example of “Performance Management”. It is a horizontal application that is needed by almost all parts of any organization. The functional requirements of each organization and the various divisions, departments and outside agencies using the system will have overlapping commonalities as well as unique variations. Trying to build, maintain and support a customized system for each group is costly. At the same time, a standard application cannot be forced on everyone.

So how to build and deliver a common software system, while still retaining the flexibility required to configure and customize to suit different user groups, is itself a Non functional Requirement. Like this there are many more NFRs. Lets us discuss them :

1.0 Multi-tenant Application Architecture

1.1 Multi-tenant Application Architecture vs Multi-tenant Database Architecture

When multiple levels of user groups from diverse set of organizations / stake holders are going to use an application, it is not advisable to customize the application at the source code level for each organization / user group. A meta data approach to building a configurable application is the right approach.

When the meta data of each user group / user organization is stored separately in the database and the application's views and behaviors are modified depending on the context of the user, then such an application is said to have a "Multi-tenant Application Architecture". Such an application can continue to have a separate transaction database for each tenant / user group.

This is different from the concept of a "Multi-tenant Database Architecture" where multiple tenant's data is shared in a single database.

"Multi-tenancy at the Infrastructure level" is another concept that deals with sharing of physical resources. Most Cloud providers and PaaS providers offer Multi-tenancy only at the Infrastructure level.

1.2 Support for all levels of Multi-tenancy.

An NFR stack should be able to support all levels of multi-tenancy – from shared nothing (dedicated server and database for each tenant / user group) to shared all (shared application and database) and the permutations and combinations in between (like shared application, dedicated database).

1.3 Who is a Tenant ?

A tenant in the context of SaaS, is usually a customer organization that pays and uses the software provided by the SaaS provider.

Whereas a Tenant in the context of other LOB applications could be any set of users grouped together by common and homogenous needs. One tenant (User Group) could be set up and managed independent of other tenants. The application could also be customized and configured to suit the requirements of each tenant.

2.0 Scalability

2.1 Data Connection Abstraction

In today's world, tying your code to any specific database is no longer accepted. Databases could be scaled out – partitioned vertically or horizontally as they grow. The data partitioning should be configurable during run-time. So it is important to abstract the data connection and pass all the Data access through an intelligent Multi-tenant DAL layer. This layer would do the connection string management during run time, based on the context of the user / tenant and based on the tenant configuration templates.

2.2 Data Isolation

Each tenant's data could be stored in a separate database, or in a shared database with separate schema / tables or in a fully shared table with a tenant ID. In all these cases the data model should be same for all tenants. (Customization of data model is discussed separately in a forthcoming section)

Developers should write code and queries as if they are writing for a single tenant. The intelligent Multi-tenant DAL layer should insert the tenant context during run time. This ensures that one tenant's data does not get mixed up with another tenant's data because of a developer's mistake.

2.3 Distributed Caching and Stateless Design

The NFR stack should use Distributed Caching and Stateless design, so that depending on the usage load, the server instances can be scaled up or down, without affecting user experience. By using a Load balancer and theoretically unlimited no. of Server instances, the application should be scalable to any no. of concurrent users.

3.0 Customizability and Configurability by Non IT Personnel

Customizability does not mean, that scripts and dedicated code are written and plugged in by IT personnel to meet the needs for each tenant. The aim should be to allow Power Users (Tenant level Administrators) to customize and configure the application to suit the requirements of their User Group, without having to depend on internal or external IT Staff. These admin screens should be accessible through a normal browser.

Customization and configuration should be supported for the following :

3.1 Themes and Logos

Each tenant should be configurable with their own Logos and Color Themes. All the UI screens in the application should reflect this.

3.2 Associating CSS files to Tenants

Many a times, the requirements of a particular tenant could be such that, changing the logo and the color scheme alone is not sufficient. The entire layout of the application pages and the graphical images used in them might have to be customized. The application should allow Power uses to create, or select from uploaded CSS files and associate them with individual tenants or for a set of tenants.

3.3 Forms and Grids

Display names / Column headings used in each form / grid, the order in which the fields / columns appear, the visibility / hide condition, whether the field is mandatory or not all these need to be customized by a power user. This feature may not be required for all forms and grids in the application, but will be required in at-least some of them.

3.4 Data model extension

When a tenant customizes the application to add one or more custom fields to an entity, the transaction database should not be disturbed. The extension fields should be stored separately in a separate table / database as key value pairs along with tenant ID and entity ID.

Data model extension should be possible through an administration GUI screen which will be used by Power users who have the necessary rights and privileges to do this.

While fetching data or saving data on an entity, the extension fields should automatically get appended depending on the context of the user and tenant.

Unlimited no. of custom fields should be supported for each entity. However certain entities that may not require customization can be excluded completely. (Data model extension feature will be disabled for those entities)

3.4.1 Custom fields on Forms and Grids

The power user should be able to add the custom fields (created using data model extension) in to any form or grid that uses the entity.

3.4.2 Custom fields on Queries and Reports

While creating Queries and reports that can be customized by a tenant, the power user should be able to view and include custom fields associated with that tenant.

3.4.3 Custom fields on Business Rules and Workflows

While customizing business rules and workflows, the power user should be able to view and include custom fields associated with that tenant, as part of the rule / work flow set up.

3.5 Business Rules Customization

Hard coding of business rules or using developer centric tools such as Microsoft Workflow Foundation is acceptable only for those scenarios where there is no end user level customization needed. But if there is a need to allow power users to customize certain business rules through the GUI, then we need to include an additional Business rule engine as part of the NFR. This engine will allow the developers to specify the business logic, variables, and other parameters during development and also expose them to the GUI for a power user to configure and customize. While such an engine should be used only for end user customizable scenarios, it should interoperate with other developer centric tools for more complex requirements.

3.6 Workflow Customization

This is similar to BR Customization. The developers should write the domain specific activities that are required in the application and expose these activities to a Workflow Designer. The GUI based workflow designer will be used by Power users to configure the sequence of activities with parallel and branching trees. The power users will also set up the start – stop and other conditions required for the execution of the workflow. The workflow steps could be manual or automatic. Manual steps will advance depending on the status returned from user driven activities. Automatic steps will advance once the business logic within them is executed.

The workflow customized by each tenant should be stored as meta data. During run time, the workflow engine, should load the appropriate version depending on the context of the tenant and execute the same.

Multiple workflows should be capable of running at the same time. Some of the workflow activities could be background jobs that can be scheduled and scaled out independent of the application.

3.7 Report Customization

3.7.1 Canned reports could be customized by tenants to change column headings, hide / show certain columns and add tenant specific custom fields if necessary.

3.7.2 Power users should be able to create Adhoc queries using a GUI based Query builder. They can drag and drop fields from entities that have been exposed to this tool during development. While doing this, tenant specific custom fields should be automatically shown in the respective entities.

3.7.3 The rights / privileges and data scope policies defined in the Access Control System should be enforced during Adhoc queries / reports, so that what users see is automatically filtered to match their data access privileges.

3.7.4 A Report / Chart designer could help power users to create their own reports and charts using queries created with the Adhoc Query builder.

3.8 Notification Template Customization

The notification templates used for each tenant should be customized through the UI so that the power user can set it up to suit the tenant organization.

4.0 Configurable Access Control System

Different tenants (User groups) might want to follow different policies on Access Control with respect to the data and features provided in the application. CRUD privileges related to database entities, named privileges associated to each entity, privileges defined at the field level, page level, service level and feature level – all these have to be configured for each tenant and user.

During development, the developer should focus on privileges and demand them when necessary. The roles and users could be set up during run time.

4.1 Tenant specific roles and role – privilege mapping

Tenants should be able to create custom roles for their organization and assign them to the users. They should be able to map the various privileges to the role and thus control "who can see what and who can do what" in the system.

These tenant specific access control policies should be remembered as meta data, which then should be enforced dynamically during run time, depending on the context of the user and tenant.

4.2 Dynamic Data scope policies

Often it will be needed to put a boundary on the data that can be accessed by a specific role. For example a Finance Manager should be able to see data only related to his division, or a HR manager should administer the application only for employees located in say 3 locations assigned to him.

In such scenarios the variable based on which the data scope is defined (Example division, location) could itself be a tenant specific master that the tenant could have created and customized for their organization. So the mapping of the Data scope policy should be done during run time by pulling up the tenant specific master data. The power user would be able to map them as one of the data scope conditions.

4.3 Single Sign on

Many tenants will have their own internal identity management and user authentication system. It should be possible to integrate the User authentication of the SaaS application with the authentication system of each tenant through federated authentication. This should follow the standards provided by the SAML protocol.

4.4 Tenant – Sub tenant Hierarchy

While each tenant is a group of users set up and managed independent of the other tenants, there needs to be a hierarchy among the tenants. For example a nodal agency / distributor could be a parent tenant and multiple agencies / dealers below it could be set up as child tenants. The child tenant in turn can have further smaller agents / contractors who could be sub-tenants. Customers and end users could be created within these sub tenants.

All the customization, configuration and access control settings done by the Power Users should be aware of the Tenant – Sub tenant hierarchy, and these settings / configurations should automatically roll down for the child tenants.

4.5 Approval management for certain types of users and tenants

Tenants at one level may require approval from their parent tenants to create certain types of sub tenants.

Users may also need to be created with access to multiple tenant's data. This is generally required by members of service teams that look at data across Tenants for an unified view or for reporting. Approvals and Permissions from the respective Tenants should be managed before activating such users.

4.6 Tenant Configuration Templates

Tenant specific configurations could be set up as a template and these have to be automatically rolled down to all the Child tenants. Child tenants could be allowed to selectively customize their own settings and these will be automatically rolled down to downstream tenants.

5.0 Other NFRs

Some of the other NFRs are mentioned below in brief.

5.1 Notification

Should support FTP and email notifications, both in instant and batch mode. The templates for notification (at tenant level) should be customized by the Power user through the UI.

5.2 Schedulers

Time and Event based schedulers should be available as services, which can be used in the business application.

5.3 Auditing

The system should support Auditing Services, where audit trail of any transaction or event could be activated by simply calling a service.

5.4 Metering

Metering of events and transactions will be required and this can be used to set usage quota for each tenant / user group. This should be tied in to the access control system.

5.5 Usage Auditing

This will be used to monitor the usage level / pattern of the various system components by different user groups. The data can be later used for Product Analytics.

5.6 Module & Feature provisioning

Not all modules and features should be accessible to all the user groups / tenants. A higher level administrator (SaaS Provider / Parent Tenant) should be able to control which modules and features of the application are accessible by which tenant.

5.7 Pre and Post Processors and Policy Injection

The system architecture and plumbing layers should support pre and post processors and policy injection mechanisms that are also tenant aware / tenant specific.

5.8 Logging and Exception Management

Tenant wise Logging and Exception management will help in tracing and troubleshooting tenant specific problems.

5.9 Master data and Pick up lists Management

Master data and Pick up lists created for each tenant should be rolled down to sub tenants, but should remain customizable by each tenant.

6.0 NFR Stack Design

A piece meal approach of building the NFRs described in the previous sections, will lead to inconsistency. Different developers will deal with these issues differently. Multiple applications requiring the same NFRs are often built ground up, following different approaches. Maintenance and Support of such applications also become expensive.

Traditional on-premise applications custom built for a specific user group have an NFR overhead of 10-20%. This means in a 100 people month project, 10-20 people months are spent on NFRs.

Whereas in modern Multi-tenant applications, the time and cost overheads associated with NFRs could be as high as 30-50% of the total development effort. That is, in a 100 people month project, 30-50 people months have to be spent on NFRs.

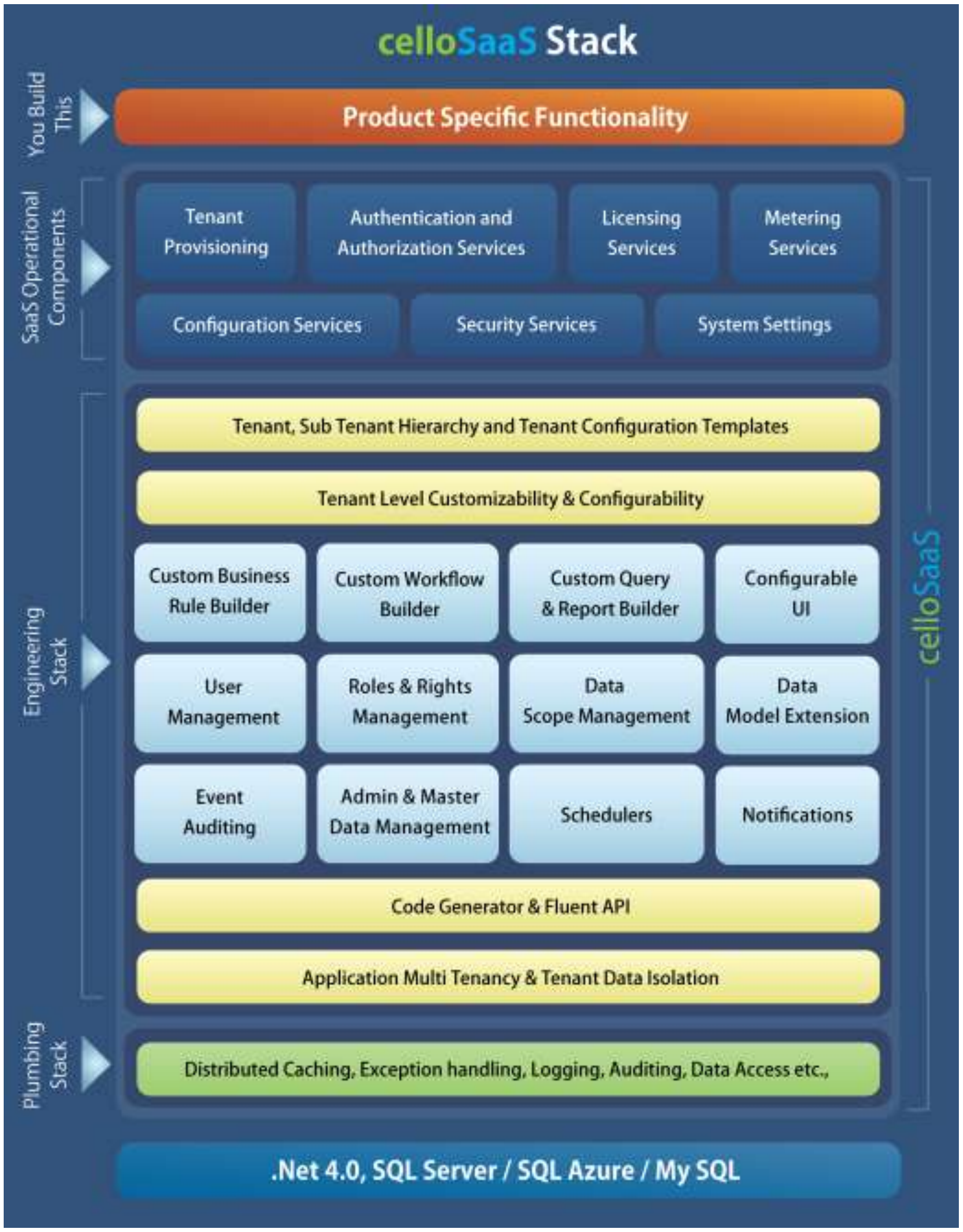
Ideally, NFRs have to be handled at the architectural and framework level. The NFR stack should be loosely coupled with the functionality layer. The application owner should be able to maintain and upgrade the NFR stack independent of the domain specific functionalities.

A Multi-tenant NFR Stack, should contain all the plumbing, engineering and operational components in a ready to use form (APIs and Services). Using this, developers can focus on building application specific functionalities instead of struggling with the plumbing and engineering aspects.

The application provider should also be able to re-use the NFR stack for other products and applications. This ensures consistency and quality across multiple applications while reducing overall cost of development, maintenance and support. (TCO).

While building or buying such an NFR Stack, care should be taken that the application provider retains ownership and control over the entire application stack both technically and strategically. Vendor lock in or platform lock in should be avoided.

A Typical NFR Stack / Framework for a Multi-tenant SaaS Application


celloSaaS Stack
You Build This
Product Specific Functionality
SaaS Operational Components
Tenant Provisioning
Authentication and Authorization Services
Licensing Services
Metering Services
Configuration Services
Security Services
System Settings
Engineering Stack
Tenant, Sub Tenant Hierarchy and Tenant Configuration Templates
Tenant Level Customizability & Configurability
Custom Business Rule Builder
Custom Workflow Builder
Custom Query & Report Builder
Configurable UI
User Management
Roles & Rights Management
Data Scope Management
Data Model Extension
Event Auditing
Admin & Master Data Management
Schedulers
Notifications
Code Generator & Fluent API
Application Multi Tenancy & Tenant Data Isolation
Plumbing Stack
Distributed Caching, Exception handling, Logging, Auditing, Data Access etc.,
celloSaaS
.Net 4.0, SQL Server / SQL Azure / My SQL

Abstract - Summary

An application that addresses a diverse set of stakeholders / organizations and multiple levels of user groups is more complicated and technically challenging than an application that is focused on a niche homogenous user base. The Non functional requirements (NFRs) of such applications are hence very critical and important.

Defining the NFRs independent of the functional requirements has many benefits. It leads to a focused effort on NFRs, a robust architectural framework, engineering excellence and consistency. The NFR stack when implemented as a loosely coupled framework, could also become a common stack for multiple applications / products, resulting in significant reduction in cost of development, support and maintenance (TCO).

A NFR Stack should contain all the plumbing, engineering and operational components in a ready to use form (APIs and Services), so that developers using them can focus only on building the application specific functionalities. While building or buying such a framework, care should be taken that the application owner / provider retains ownership and control over the entire application stack both technically and strategically. Vendor lock in or platform lock in should be avoided.

About the Author

Ramkumar is the Director of Product Management at Asteor Software. He was instrumental in incubating and bringing to market two new products in the Cloud / SaaS space: Techcello and Synergita.

Ramkumar is also the founder Director of Mango DVM an angel funded company in the Digital Music space, that has just completed a third round of funding.

Before becoming an entrepreneur Ramkumar, had spent over 2 decades in various corporate functions such as Automation Engineering, Project Management, Marketing, Sales , General Management, HR and Leadership Development.

He can be reached at rsr@rsrinnovations.com or ram.k@techcello.com