Cork Institute of Technology

Department of Computing, Cork, Ireland

Integrating Service Orientated Architecture design principles into Software as a Service applications

By Jonathan Roche

Research project, M.Sc. in Software Development

Supervisor: Ms. Gemma O’Callaghan

Date: February 2014

1 Abstract

Author: Jonathan Roche

Title: Integrating Service Orientated Architecture design principles into Software as a Service applications.

Independently, both SOA and SaaS based applications deal with delivering services to business with a focus on improving agility, increasing speed and providing cost reductions.

This research proposes to investigate an architecture that uses SOA principles to support the development of SaaS applications.

It is my belief that through the use of SOA in conjunction with Cloud technologies, a software vendor can provide complete service based solutions for many small and medium sized enterprises, with a shorter delivery timeframe and at a reduced cost than traditional implementations.

2 Table of contents

3 Glossary

3.1 Vocabulary

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASP	Application Service Provider
BPM	Business Process Management
CIO	Chief Information Officer
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
CRUD	Create, Read, Update and Delete
DCOM	Distributed Component Object Model
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JPA	Java Persistence API
JPQL	Java Persistence Query Language
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
POM	Project Object Model
REST	Representational State Transfer
ROI	Return on Investment
RPC	Remote Procedure Call
SaaS	Software as a Service
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOSaaSOA	Service Oriented Software as a SOA
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Services Description Language
XML	Extensible Markup Language

3.2 General Terms

Cloud Computing, Software as a Service, Service Orientated Architecture.

3.3 Keywords

Software as a Service, Service Orientated Architecture.

4 Introduction

In an article penned for InfoWorld, David Linthicum outlines the scenario when discussing the possibility of developing an enterprise solution using a Service Orientated Architecture with a potential customer:

> *The phone rings. "We're looking for a path to cloud computing for our enterprise that will reduce risks and cost, but also raise our ability to be flexible and agile," the voice on the phone pleads.*
>
> *"How about using SOA as an architectural pattern to drive the movement to cloud computing?" I respond.*
>
> *"Isn't SOA outdated?" the person immediately asks. "I thought somebody declared it dead."*
>
> *There's a long pause. Then I respond: "OK, how about the ability to break up your existing enterprise assets as sets of logical services that can be formed and/or reformed into business solutions? That will provide us with a foundation to evaluate each service as something that may benefit from new cloud-based platforms and determine the best path for migration. This approach will also provide better access to your core information and critical business services, no matter where they reside."*
>
> *"Wow, that sounds like exactly what we should be doing."*
>
> *I make a note to myself: Remove "SOA" from any statement of work, but make sure we use a service-oriented architecture.*[15]

Despite the negative connotations associated with the acronym “SOA”, it is my hypothesis that many of the design principles that are essential to implementing a SOA solution are equally applicable to the development of cloud based applications.

It is the objective of this investigation to provide an in depth analysis of SOA and of its suitability for integration into a standard architecture for providing SaaS applications. It will also involve practical application development to investigate the concepts outlined, which will provide an evaluation of integrating SOA design principles into SaaS applications.

This investigation will address one primary question:

How can service orientated design principles be deployed into a software as a service architecture, and what would be the advantage of doing this?

As part of the investigation to find the answer to this question, a number of smaller questions need to be answered in advance:

- Do SOA and SaaS solutions share any common architectural patterns or implementation patterns?
- Why would a software vendor consider combining both SOA and SaaS in a commercial software product?
- What would the high level architecture of a potential combination of SOA and SaaS look like?
- What potential advantages could a SOA provide for a SaaS solution?

After addressing these questions, this report will consist of:

- An in-depth analysis of the evolution of SOA design principles.
- An in-depth analysis of the SaaS deployment model.
- A detailed theoretical study on the architectural impact of integrating SOA principles into SaaS applications.
- An outline of how practical implementations of SOA design fit into applications that use the SaaS deployment model.
- The development and evaluation of a practical implementation using the envisaged architecture and an analysis of the benefits that such an architecture could provide.

5 Linking SOA and SaaS

From its very inception, the world wide web has been providing services, which through the demands of consumers have been ever evolving. These services range from web browsing and email, to social networking applications and enterprise business solutions. In software, a service can be defined as a loosely coupled, unassociated unit of functionality that is self contained. Today, virtually every major software player is focused on services, with Amazon and Google leading the charge. Names that were synonymous with hardware for many years such as Dell, IBM and Hewlett-Packard are now all repositioning themselves as service providers.

Over the past few years, a lot of software vendors have moved from a conventional application development approach to a service orientated development mentality. One of the more common service driven approaches is to deliver an entire software application as a service which is known as Software as a Service (SaaS).

5.1 SaaS Overview

SaaS can be defined as:

Software that is owned, delivered and managed remotely by one or more providers. The provider delivers software based on one set of common code and data definitions that is consumed in a one-to-many model by all contracted customers at anytime on a pay-for-use basis or as a subscription based on use metrics. [7]

SaaS is a software deployment method where the application is available for consumption from the cloud. One of the main selling points for SaaS applications is that they are charged based on usage. Unlike traditional on premise solutions where a consumer purchased a typically expensive software license for applications such as CRM or ERP, consumers can subscribe to a SaaS application and are only charged based on their usage of the application.

This greatly reduces the cost of the application to consumer as they no longer have to be concerned with hosting the application on premise or dealing with software upgrades. SaaS solutions also have the added benefit that they are available out of the box. Should a company choose to deploy an on-premise CRM solution, then, even with the software available, it will take some time to install and configure locally. This is not an issue for SaaS solutions as one a consumer has their subscription in place, the application is ready to use.

Also, some SaaS vendors are offering different editions on the same application based on the subscription. This is similar to choosing between Standard, Enterprise and Custom editions of software, and can further reduce the cost to the consumer by providing slimmed down versions of applications where enterprise features are not required [2].

While this can make large SaaS solutions more available to small and medium sized enterprises, many still shy away due to the high cost per utilized feature. A small enterprise customer may only require a subset of the features of the standard package, with the vast majority of features not being required. Providing a "pick and mix" of features would be hugely beneficial to both the vendor and the consumer. The vendor could greatly increase its revenue and its customer base, and the small enterprise customer would have their enterprise features at a fraction of a cost of the alternative on-premise solution. Also, as the small business grows, so too will its feature requirement and application usage, resulting in more revenue for the vendor.

While theoretically it is possible for a vendor to provide this with various different editions, the practicality of such an approach would negate the benefit of providing a SaaS solution, and indeed would in all probability incur a greater overhead to the vendor. Each additional feature would add a factorial cost to the number of versions to maintain which is a non runner [29]. Providing a cost effective implementation for this capability revolves around building software applications as a collection of interworking services known as Service Orientated Architecture (SOA).

5.2 SOA Overview

SOA is a set of architectural design principles that supports service orientation. Service orientation involves thinking of the requirements of a solution in terms of the services it requires and the outcomes of those services. A service is a self contained, repeatable activity that meets a business requirement. A service may consist of a number of smaller services and its implementation is encapsulated from the service consumers.

SOA can be defined as:

Service-oriented architecture (SOA) is a software design methodology based on structured collections of discrete software modules, known as services, that collectively provide the complete functionality of a large or complex software application. [3]

Implementing an application based on SOA's design principles results in software components that provide their functionality as a published service which can be utilized by external components. These services can be thought of as reusable building blocks for a complete enterprise solution.

5.3 Relationship between SOA and SaaS

With the exception of the over use of these two acronyms as buzz words by company executives, politicians and the media, on initial investigation both concepts would appear to have very little in common. Indeed, attempting to connect both has caused some heated debate in the past.

In 2009, Gartner analyst Anne Thomas Manes published a very controversial article entitled "SOA is Dead; Long Live Services" [17], which at the time sparked a heated debate amongst the software architecture community. Within the article, Ms. Manes suggests that SOA had turned into a "great failed experiment", and that SOA is "survived by its off-springs" like SaaS and Cloud Computing.

In reality, these two concepts can be used in unison, with an increasing number of software vendors investigating innovative ways to combine the strengths of both concepts to deliver solution that enhances the end users experience and increases the customer's overall return on investment and responsiveness.

In terms of the delivery of software as a service, SOA is what differentiates the current generation of SaaS providers such as Salesforce.com [23], from the failed application service providers of the dot-com era. While SOA is not required to deliver a SaaS solution, it does facilitate the construction of composite applications, with the integration of external services with in-house developments being a key advantage.

A Service Orientated Architecture provides a modular, component based architecture which enables flexibility and reuse. SOA is more than a technical approach and methodology for creating software solutions. It is also a business approach and methodology used by companies to deepen the understanding between the business and the software solution, and to help the business adapt to change [22]. One of the key benefits of a SOA approach is that the software is designed to reflect best practices and business processes instead of making the business operate according to the rigid structure of the technical environment.

In the most simple of terms, SOA can be thought of as a software design methodology and SaaS can be thought of as a software deployment methodology

5.4 Can we combine SaaS and SOA?

Providing SaaS solutions benefits the vendor by taking the best practices and business process focus of SOA to the next level. These benefits can be seen by both cloud SaaS providers and end users. SaaS applications that implement services through black box components which are loosely coupled and designed to provide a clearly defined level of service, result in software applications with increased levels of scalability and elasticity.

Cloud service providers that architect their SaaS solutions using reusable service orientated components can easily identify individual components that can be deployed in various different SaaS offerings.

This approach facilitates companies that want to leverage existing components to create new SaaS applications that are consistent, controlled, and more easily managed. Within SaaS applications, SOA can be seen as a business approach to implementing an efficient solution that supports reuse and gives the business flexibility to react quickly to opportunities or threats that may present themselves in the future [14].

In addition to enabling the provider to easily swap some of the SaaS applications' functionality, a service orientated approach allows the SaaS vendor to attract an eco system. Many SaaS vendors are providing their interfaces openly with can be used by independent vendors to build their own SaaS applications more efficiently.

These independent vendors create customized SaaS applications that utilize the services exposed by a larger SaaS vendor. As the independent vendor will not have to create the common services provided by the larger SaaS vendor such as messaging middleware or business process services, they can get to market with their application in a much reduced timeframe and with a lower cost. This has become a standard model used by SaaS vendors to provide cost effective, robust solutions with lower budgets.

5.5 Where SOA meets SaaS

In addition to the business functions needed by an end user, every true SaaS offering will provide:

Multi-tenancy, user authorization and authentication, provisioning and ordering, service monitoring, service catalogue and pricing, usage metering, invoicing, billing and payments functionality.

A typical SOA solution will consist of both service producers and service consumers. In a large organisation, the service producers and service consumers may reside within different systems, or even within different organisations and subsidiaries.

In this scenario, a formal service management system needs to be put in place, along with some security measures around the publicly exposed services. As a result some additional features are needed by the SOA solution:

Service catalog management, authorization, authentication, provisioning, usage metering and inter departmental/organisational charging.

As the SOA solution matures, the core requirements of a SaaS solution become a necessity as part of the solution.

The reverse relationship is a little more visible from the outset. For any SaaS application, a vendor will want to support the addition of new service to their service catalogue, or to modify some of the existing services, without a complete redesign of the existing solution. To enable this, the basic core functionality as outlined earlier such as pricing, metering etc. should be reusable by different SaaS solutions. This inherently implies that the underlying architecture needs to be service orientated. The use of SOA allows much easier adoption of plug and play services, and will lead to a lower cost of providing additional features [8]. Figure 5.1 outlines some of the common components between modern SaaS application and SOA applications of the past.

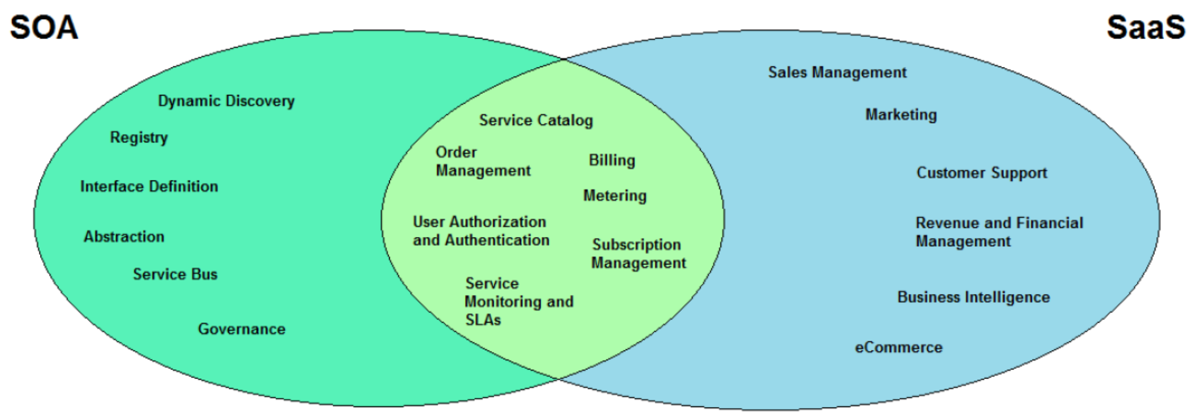


Figure 5.1

In the past, many SOA deployments have fallen short of vendor expectations, with many providing very little or no return on the investment. To fully reap the rewards of a large scale SOA solution, it is almost essential to have the management functionality that is core to SaaS in place. This is where both concepts meet, and in my opinion will be the natural architectural progression for a true low cost service orientated solution.

With SaaS, services are delivered to a greater number of clients, and having it built on top of a SOA architecture makes the application easier to scale than a more monolithic "do it all" application. In a nutshell, SOA involves designing service-oriented services. These services can then be deployed and provided in SaaS model.

6 Evolution of Service Orientated Computing

Over the past ten years, a vast number of companies worldwide have implemented Service Oriented Architecture (SOA). SOA involves the deployment of business processes with the aid of pre-built, pre-tested services, the idea being that visual software tools can be used to wire together Services on a screen to implement particular business functionality.

SOA promised faster deployment cycles, enhanced flexibility and responsiveness, easier change management and alignment of business and IT functions. On the change of a business requirement, an organization with an underlying SOA based solution would be able to implement the change rapidly by simply utilizing pretested business services via simple user interface, thus reducing the greater expense of programming and deploying a solution using traditional methods.

6.1 From Procedural to SOA

Service Orientated Architectures were not developed as an isolated green-field concept, but almost as an evolution of existing software development models. The real building blocks for Service Orientation can be traced back to the 1980's when a shift was made from Procedural programming models to Object Orientated Models. Object Orientated Models paved the way for Component Based Development Models in the 1990's, which in turn provided the starting point for Service Orientated Models.

Also, the underlying architectural concepts are far from new, with many having evolved from older technologies such as CORBA and DCOM, both of which provided discovery and late binding functionalities. Similarly, encapsulation, abstraction and programming to interfaces are some of the fundamentals of Object Orientated Analysis and Design.

While Service Orientated Architectures preserve some of the key benefits provided by a Component Based Architecture such as encapsulation and dynamic loading and discovery, it addresses the shortcomings of the Architecture. One of the most notable evolutions centred around messaging, with a move away from remote method invocation and the introduction of message sending between remote objects. Along with describing the message structures, in a SOA, schemas also include the message handling definition, thus decoupling the sender from the handling of the message.

Unlike its predecessors, SOA provides the ability to construct complex enterprise systems from autonomous services through loose coupling. When a resource is only utilized through a well established stable interface, changes to the underlying service implementation by the service provider will not have any adverse impact on the service consumer. Also, alternative services can be interchanged with minimal impact on the existing application, even if the service interfaces or implementations are not based on the same technologies.

Although traditional architectures also evolved, with a move from mainframe applications to client server models and n-tier models, the application components generally remain tightly coupled, with most bound both at compile time and runtime. Quickly replacing a component based on a new business requirement is prevented due to this tight coupling.

One of the major changes between SOA and Traditional Architectures is the exposure of the applications functionality. When building a traditional application, the business components are constructed to fulfil the needs of the particular application, with the user interfaces of the application being the only consumers of the business layer components. However, with a SOA model, the business components are built as standalone components that can be used by a wide variety of applications. As a result, service interfaces cannot be changed, as the service has no knowledge of the applications that are using it. The stability and availability of a SOA applications service is critically important [6].

6.2 Benefits

From the comparison with Traditional Architectures, a number of key benefits of implementing a SOA solution can be identified:

- SOA can facilitate a closer alignment between an enterprise solution and its business requirements. With ever changing business requirements, a more flexible enterprise solution can be provided through SOA to quickly adapt as needed.
- SOA can help to provide reuse of already existing solutions.
- Through the use of stable, well defined interfaces, the integration of new services becomes more straightforward with SOA.
- SOA provides a robust architectural model for integrating various stakeholders' interests into an enterprise solution, improving customer satisfaction and reducing costs.

6.3 SOA Design Principles

There are 8 design principles to keep in mind when designing a SOA service [5]:

6.3.1 Standardized Service Contract

When creating a SOA implementation, the service contract is probably the most important artifact of the design process. This service contract will define the capabilities of the service, and how the service will function. Standardizing the service contract leads to enhanced interoperability with other internal applications and services and external platform vendors. The service contract has to be created before the service itself is implemented, keeping the contract independent from any specifics of the implementation. The service contracts will contain:

- Service Definition, containing the technical details of the service and the functionality it provides.
- Data Definition, detailing the format of the data exchange.

In the context of web services these would correspond to the WSDL and XSD. Other contracts such as security and enterprise policies, and SLAs may also be outlined. Each of these contracts are created independently, allowing services to utilize the same data while avoiding an extra overhead of data transformation and service integration. Contracts also need a specific "owner" to dictate the changes to the contract over its life time, but without having a vested interest in a particular implementation of the service contract.

6.3.2 Loose Coupling

Within SOA, the concept of loose coupling is derived from object orientated design principles, but with particular emphasis on the service contract. This service contract can be seen as the gateway to the services' underlying functionality which will be used by service consumers. In a SOA offering, the service contracts should be independent from the service logic. By keeping the service contracts of services loosely coupled, a SOA solution can support vendor diversity and provide increased interoperability.

6.3.3 Service Abstraction

The service abstraction design principle states that a service contract should only contain information that is relevant for its invocation, and that no other document that contains additional information with regards to the service should be made available to the service consumers. Should a service consumer be provided with a service contract that exposes the internal functionality of the service, the service may be used in a manner that was not intended. This could lead to reusability issues if changes are made to the service implementation in a later version of the service.

6.3.4 Service Reusability

This design principle focuses on building services with the correct type and quantity of application logic. By maintaining a concentrated effort on the quality of the service, the potential for reuse is greatly increased. A reusable service will not contain logic that is linked to any particular business process, and can instead be used in multiple instances across the solution.

6.3.5 Service Autonomy

The purpose of this design principle is to create services that are independent from their execution environments. As a service may have little or no control over shared external resources, it can have no guarantees that the resource is available when requested. With SOA solutions, services used may exist outside the solution itself, leading to failover support needing to be provided to maintain a service's autonomy.

6.3.6 Service Statelessness

This design principle emphasises the separation of a services state from the service itself, with the state management being delegated to a dedicated state management service. By removing the state from a service, it generally becomes more reusable and flexible. However, due consideration needs to be given when implementing, as this will incur an increased overhead.

6.3.7 Service Discoverability

A software service can only be reused if it can be found and understood. To enable service discovery, the service needs to be documented in a consistent fashion, stored in a searchable repository and facilitate the retrieval of documented information in an efficient manner. Not only does this increase the reuse of the service, it also reduces the amount of development as existing services may be discovered in advance of creating a functional duplicate.

6.3.8 Service Composability

This design principle is key for SOA development as it should be possible to build a software solution by combining independently existing components. The overall effectiveness of this principle depends on the correct implementation of the others. The standardized contract principle enables interoperability amongst services, service loose coupling allows using a service without adverse dependencies and service autonomy and service statelessness increase the availability and reliability of the service.

6.4 Why such bad press?

Despite potentially providing huge benefits, the term SOA has been associated repeatedly with software project failures, and is regarded by some as a taboo subject. It is widely acknowledged that a large percentage of companies that had embarked on adopting a SOA strategy were not satisfied by the end result [13]. However, quoting individual findings without any context has led to an unfair tag for SOA. In many cases though, the reasons for failure many be tracked down to human error rather than technology failures, though invariably the technology is blamed.

Successful SOA implementations have been expensive for many reasons, with some of the critical ones being:

6.4.1 Inherently Complex Technology

The vast majority of "SOA Stacks" used in SOA implementations today are based on older BPM style approaches, with a central hub maintaining the state of business processes while end point services implement process steps. Implementing integration processes with such an approach requires inordinate amounts of programming, greatly increasing professional services outlays even for simple projects. On average, the consulting outlay for a successful SOA project has been between four to seven times that of license cost, significantly eroding target ROI on such projects.

6.4.2 Change Management Issues

Constraints of the underlying technology also make it difficult to implement and manage business process change. Since the tools are inflexible (because of inherent flaws in the underlying implementation technology), changes are difficult to implement and maintain. In the typical case, significant programming or consulting time is required for each implementation change, with multiple develop, test and deployment cycles. In addition, projects are disrupted because the core software does not allow dynamic changes to processes.

6.4.3 Scaling Problems

With current approaches, scaling implementations (as more users are added or more processes deployed), is very expensive. For the most part, all existing SOA technology is hub/spoke based and does not scale linearly. When the limits of a given hardware machine are reached, an additional 'hub' must be added, incurring massive software costs. Scaling also requires disruptions since in the typical case the entire system must at some point be stopped, changed or modified, and then redeployed across and additional number of hubs. Existing software stacks based on BPM style approaches make scaling particularly difficult since there is no easy or seamless way to deploy a portion of a business process on a new hub, should the need arise.

6.4.4 Architecting for the Cloud

In addition, with the proliferation of cloud based technologies, enterprises attempting to implement a traditional stack will find further elements of rigidity in the system. Traditional stacks were not built with the distributed, ubiquitous enterprise in mind. In order to achieve the elasticity benefit of the cloud for a distributed application, the application tier has to be stateless. It is an important best practice that all application logic in the cloud should be stateless. Object instances, session beans and server cookies have to be avoided. If the load on an application becomes too great, the cloud should respond by provisioning adequate resources to meet the need. The less state an application contains, the more equipped the cloud will be to facilitate this seamless elasticity. The cloud may need to create additional instances to handle the load, and some instances may crash. However, because of the clouds requirement of high availability and partition tolerance, such a crash must not interfere with the process that the instance of the cloud is supporting [30].

6.4.5 SOAP Complexities

When at is peak, most SOA communication involved SOAP, which created a dependency hotspot through WSDL. Any changes to the document schemas for service operations contained in the WSDL caused a change to the WSDL itself. As a result, this limitation with SOAP created, by association, a limitation with SOA. This is should not be the case as SOA does not need SOAP, and with the rise of REST this can be avoided for SOA applications.

6.4.6 Incorrect use of the Enterprise Service Bus

In many cases, the ESB was used to host business logic, or to orchestrate some processes. Turning the ESB into a service coordinator or a process coordinator creates dependencies that impact performance and incur an increased cost of change.

Also, many ESBs were deployed in a hub and spoke architecture, turning it into a failure point and a bottle neck. By creating these dependencies, an implementation is breaking all the conventions of a SOA and is doomed to failure.

6.5 Why Consider SOA Now?

Despite the perceived failures of SOA in the past, resulting in the acronym being approached with much trepidation, the truth is that many organisations are have experienced success, and others are continuing to ramp up their investment in SOA. Many IT consultants face a difficult sell when trying to convince CIOs that adopting a SOA strategy is correct for cloud solutions, due to its perceived failures and even its death.

The train of thought for the death of SOA, is generally attributed to Ms. Mannes article title "SOA is dead! Long live Services" [17]. However, the article itself proposes the death of the term due to its negative connotations, rather than that of the underlying service orientated technology.

SOA is an architectural style, and can be thought of as a collection of best practices, encompassing many commonly recognized highly regarded concepts such as loose coupling and reuse [8]. While the reasons for failing when attempting to implement a SOA solution vary, virtually all failed because the organisation failed to implement best practices, and in doing so were not implementing a SOA. It is the misapplication of the acronym SOA to practices that are far removed from its principles that needs to be eradicated.

Despite the talk of SOA being dead, the reality is that SOA has matured as an architectural style. Many companies are now focused on the consumption of services, which is the cornerstone of the SOA design mentality. The fact that the hype has long vacated the SOA world has not lead to a reduction in its uptake, however it can lead to an "out of sight, out of mind" mentality from CIOs looking for the "next big thing". As experienced from the misapplication of SOA in its infancy, the priority should be on addressing the needs of the business, regardless of any amount of hype over the latest buzz word.

Over the years, SOA has been tightly linked to the Simple Object Access Protocol (SOAP) as a result of its wide scale industrial support and implementation. SOAP was developed to decouple a service's logic from its messaging, allowing services to communicate seamlessly. While this is good in theory, it incurs a cost were services use a common messaging protocol.

In the case of services that only use HTTP as their messaging protocol, only a very small subset of the protocols functionality is utilized. Due to the unnecessary overhead of SOAP, lightweight alternatives facilitated by REST are adding an increased agility to the development of SOA based solutions.

6.6 REST-based SOA – Basic Principles

REST (Representational state transfer) based SOA is an emerging technique of implementing SOA projects that resolves the three critical issues discussed previously. Rest based SOA provides a model for creating loosely coupled, event driven, document centric, asynchronous, message based enterprise business services.

Through the combination of a simplified service model and event driven messaging, Rest based SOA allows the deployment of linearly scalable, easy to change, manageable SOA projects at a fraction of the cost associated with traditional SOA stack implementations [6].

While document style interfaces are the norm for generic, web services centric SOA implementations, REST based SOA takes the notion to a whole new level. In the traditional web services centric approach, document style interfaces are described in a WSDL file, with the constraints on input and output messages being described by strongly typed XML schemas. The consequent tight coupling mandates an RPC style of programming with centralized state management, impeding scalability and flexibility.

The REST based SOA approach relies on three fundamental principles:

- Messages.
- Documents.
- Content Based Routing.

All communication between Service end points occurs via messages. Distributed message processing is a critical component of a REST based SOA system. Inside messages are documents; the REST based approach does not use Remote Procedure Call (RPC) or anything remotely like it. RPC is popular because it is easy for programmers to understand and view a problem space using this method. When RPC goes away and documents and messages remain, it’s a different way to think about and view the problem – one has to solve the problem with a combination of asynchronous communication and content based routing.

An important characteristic of documents is that they must be human readable: If you stop all the computers and hand out the documents to humans they should be meaningful. Content based routing is the third fundamental principle of REST based SOA architecture: You don't know the end point of your messages by network addresses, or server names. You don't address individual resources by network addresses; rather, as described in subsequent sections of this paper, you insert URI's in the document and that gets it to its destination via late binding at runtime [27].

6.6.1 Document centric Interfaces and State Management

In a REST based SOA architecture, service interfaces are not based on any formal contract, relying instead of the GET, PUT, POST and DELETE operations familiar from HTTP. Messages containing documents flow between Services, typically distributed via an asynchronous, message based ESB infrastructure.

6.6.1.1 Documents as Interfaces

In REST based SOA, the document is the interface. There's nothing else to rely on for interfaces in this architecture – no protocols (such as WSDL) are required. The developer of a service must express the interfaces in the document, which are understood by downstream services via embedded message formats. This mechanism removes the need for defining a specific, formal contract between Services, enabling a late binding, loosely typed, easier to modify and efficient implementation resulting in lower maintenance and development costs.

6.6.1.2 A Service Oriented Approach to State Management

Further, in the REST based approach, all state information is carried within the document. As they flow across Services in a process, documents are dynamically augmented with state information and persisted locally, obviating the need for a centralized state repository. Information is never lost due to a system shutdown for any reason.

On restart, processing picks up where it left off, making a REST based SOA implementation significantly more scalable than traditional approaches.

6.6.2 Document characteristics and practices

Documents in a REST based SOA system are self describing. Documents are not dependent upon any particular stack of software to be used or to be useful. They are completely neutral to the services that consume and produce them. XML is not an explicit requirement although in most large projects several documents tend to be XML based.

Documents specify the contract between services. In REST based SOA, this is not done explicitly. Since documents are self describing, services that process a document implicitly understand the message structures embedded within the document, obviating the requirement for explicit interfaces such as WSDL.

Since documents describe the interfaces between services, they are created first – before any services - in any REST based SOA system. A good rule of thumb in the REST based approach is to be able to express all document characteristics in a non XML syntax. This ensures that documents are simple structures, typically hand crafted and easily understood, ideally avoiding references, "includes", namespaces and other XML centric features. The focus in a REST based SOA implementation is to trade off build time purity for on runtime simplicity, mandating an asynchronous, late binding, message centric approach to implementation.

6.6.3 Resources and Logical Routing

All sources of information in a REST based SOA implementation – whether they are business processes, documents, service entry or exit points or anything else – are resources. All resources are addressed by URI's which are very much like a Web URL, conformant with W3C naming specifications. URI's are used across domains (such as, for example 'development', 'test', 'production', etc.). All URI's in the system are registered with the underlying ESB infrastructure, allowing documents to be routed and filtered and policies to be enforced via URI's, as described in more detail in the following section.

6.6.4 Late Binding – dynamic, runtime extensibility

There is no build time mapping to particular resources in a REST based SOA system. Services sending and listening for documents can talk to any message types and senders can send documents anywhere routable.

At runtime, the ESB infrastructure routes documents based on content and URI's to the appropriate destination. In contrast, in the traditional approach most SOAP endpoints point to a WSDL which contains an IP address or server name with a specifically programmed interface, creating a fairly tight coupling that is fragile.

In a REST based SOA system:

- Messages can be routed to any resource – there is no 1 to 1 mapping or tight coupling of any kind.

- Resources and message contents are loosely coupled – each resource can handle many types of messages.
- Reliance is placed on dynamic typing – service endpoints need to be able to handle lots of different message types. A particular service endpoint may not be able to process each particular incoming message type but it needs to be able to handle them in conjunction with a rich, underlying message driven ESB infrastructure.

6.6.4.1 Strong typing and early binding vs. loose coupling and late binding

The REST based approach described above does not work well with SOAP for obvious reasons. A SOAP call is statically bound to a particular version of WSDL, making it inherently inflexible.

The difference between traditional SOAP/WSDL centric SOA and REST based SOA boils down to a classic argument between strong typing and early binding versus loose coupling and late binding from the programming languages world (C++ vs. Smalltalk). In the late binding approach, you don't get errors at compile time and then have to do a lot of debugging when your program fails at runtime. This problem does not occur in an SOA implementation that is supported with a rich run time environment with support for asynchronous messaging and powerful, dynamic exception handling. Service end points can throw exceptions for unhandled message types, hand the message back to the ESB and nothing rolls over and dies. The resulting system is inherently more flexible and dynamically extensible.

6.6.5 Asynchronous Messaging: the foundation of REST based SOA Platforms

Late binding and logical routing provide an underlying infrastructure that supports asynchronous messaging at its core, preferably with a distributed architecture for scalability.

With an asynchronous system, there are no blocking calls and no waiting for responses, making the system inherently more scalable than a blocking, synchronous system. Message senders are "fire and forget", in that they only publish the messages to the ESB and then move to the next step. Message listeners are like handlers: they wait for an event and when that event occurs, they know what to do with the embedded document in the event.

From a programming perspective, this approach is very different from the synchronous, RPC style approach. However, most asynchronous systems do support pseudo synchronous styles of programming, allowing some flexibility in this regard.

With an asynchronous approach, if a message needs to go through a series of steps in a process and the power dies or there's a processing problem, you can pick up where you left off since messages may be optionally persisted to disk at every step, providing assured delivery; there is no central state repository, making the system inherently more scalable.

6.6.6 Business Benefits of using REST based SOA

Technology must ultimately serve business interests to be viable in a commercial environment. REST based SOA, implemented correctly with a good underlying architecture, provides the following key benefits to the business:

- Easier to implement & maintain.
- Reduces complexities.
    - Significantly less reliance on and reduced cost on professional services.
    - Faster time to market.
- Linearly scalable system.
    - Extend the system with build as you go functionality instead of a huge upfront investment in a monolithic system.
- No centralized bottleneck.
    - Reduced cost and risk of failure.
    - Efficient use of resources through load balancing across multiple distributed peer servers.
    - Reduced hardware costs.
- Flexible and dynamically extensible.
    - As decision making data points change, change the system rapidly to incorporate new business intelligence – no huge IT intervention and no extended professional services required.
- Reuse services.
    - Significantly reduce costs through reuse of services.
- Dynamic exception handling.
    - Less downtime via easy extensibility.
- Stateless application architecture suitable for cloud applications.
    - Provides a perfect platform for elastic, distributed cloud applications.

7 Software as a Service Architectures

Moving from a self managed on premise solution to a cloud based offering imposes a number of technical obstacles which need to be addressed. Legacy applications and their underlying frameworks were not designed with a transition to a SaaS environment, and generally require a lot of redesign and re-architecture.

One of the major concerns for most companies moving to the cloud is security. While SaaS applications offer customers a remotely managed centralised solution, with a significant reduction of overheads, an enterprise will need to relinquish some of the control over its own data. The SaaS provider needs to be trusted completely with, in many cases, highly sensitive data. Any successful SaaS architecture will need to be secure and robust, along with being cost effective and efficient to maintain [21]. Two key factors for providing this are multi tenancy and metadata driven architectures.

7.1 Multi-Tenant Data Management

Depending on the requirements of each customer, the level of isolation for a SaaS application can vary, and generally fits into one of the following three categories [18]:

- Separate Database, Separate Schema: This is highly isolated but provides very little benefit to the service provider, and invariably a high cost to the consumer. It should be avoided if possible.
- Shared Database, Separate Schema: An improvement on the first option, but still carries a sizeable maintenance overhead.
- Shared Database, Shared Schema: This is by far the most preferred Multi-tenant architecture approach and can be a much more cost effective solution for both providers and customers.

These three approaches are outlined in Figure 7.1.

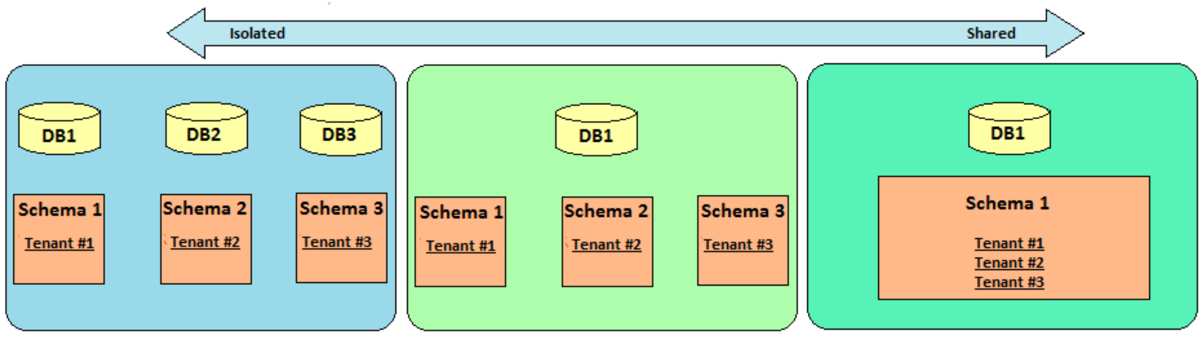


Figure 7.1

7.2 Meta Data Driven Architecture

Traditional applications are static by nature and are not suited to address the needs of a multi tenancy application. A true multi tenancy offering needs to be dynamic by nature in order to fulfill the needs of various different customers and their individual needs. The components of the application need to be created at runtime from metadata - both for the common and client specific components of the application.

This provides the ability to update both the core features and the customer specific components without impacting other tenants of the application. Virtual database structures can be constructed using user data, metadata and pivot tables as illustrated in Fig 7.2. This structure allows functional access to the actual data contained in the virtual tables.

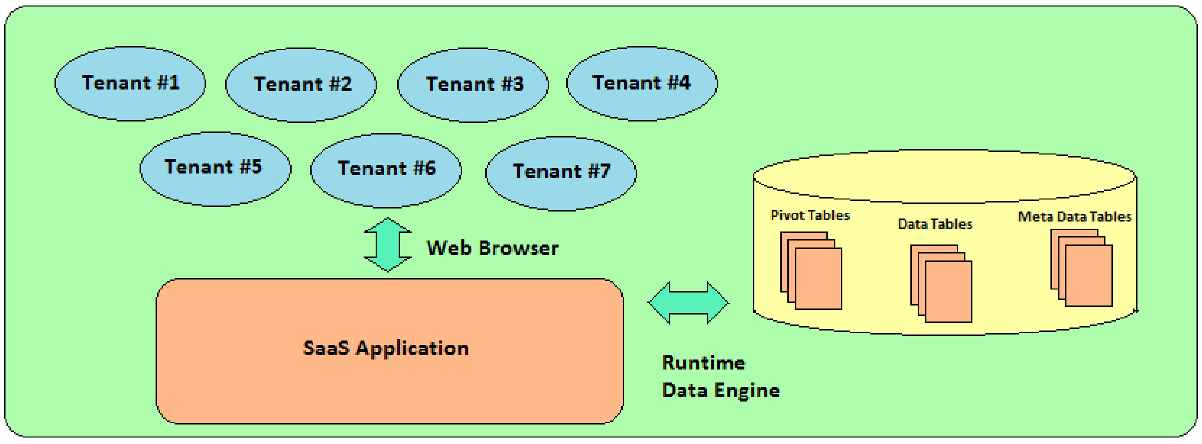


Figure 7.2

While the database structure outlined above supports a multi-tenant architecture, the application being offered by the service provider remains relatively static, similar to the ASP offering. In order to release the true potential of any SaaS application, further architectural changes need to take place.

7.3 Scalability

Though it is a term regularly thrown about with reckless abandon, and indeed is often a neglected requirement and deprioritised, scalability is an essential requirement for any enterprise software application. Due to the nature of SaaS applications, scalability takes on a much higher priority, as any performance restrictions will have an impact on every user of the application.

As outlined in the section above, multi tenancy addresses the issue of scalability from the entity layer of the application. To ensure that the underlying application servers are scalable, applications need to be cluster enabled, facilitating that addition of cluster nodes as the volume of data increases. Application services should also be stateless services, eliminating the need to handle session replication and session persistence.

The tried and tested design principle of service modularization is a cornerstone of many existing scalable systems. Along with ensuring that system has a higher level of fault tolerance, service modularization allows component level scaling depending on individual service usage. Finally a caching mechanism should be considered for data that is common amongst multiple end users. Retaining such metadata within the system will reduce database transactions and increase the overall performance of the system.

7.4 SaaS limitations

Before committing to developing or availing of a SaaS solution, it is important to consider the limitations and drawbacks of such an offering [16].

7.4.1 Security

By far the biggest concern for any enterprise evaluating a move towards SaaS applications is security. Entrusting sensitive enterprise data and business processes to an external service provider requires robust and secure data management, with a particular focus on identity and access management of end users.

7.4.2 Outages

Despite the premise and best attempts of providing one hundred percent availability, virtually no service level agreement will tie a provider to such a condition. Even the most robust service will suffer from the occasional outage, with issues ranging from environmental issues to basic human error.

While minor outages may be an inconvenience to the end user, more prolonged outages of critical applications can present serious consequences. Before outsourcing such applications, both the service level agreements offered by the provider and the historical performance of the offering need detailed attention.

7.4.3 Performance

When compared to a locally hosted application, a web based alternative hosted in a remote data centre can present performance and latency issues. In many cases the performance of the application is almost entirely dependent on the speed of the underlying internet connection. When migrating to a SaaS environment, applications where performance is not seen as a critical requirement should be considered for the initial transition. Also, performance analysis tools can be employed to help enterprises and service providers monitor the performance of applications after the transition to the data centre.

7.4.4 Compliance

Hosting business data in a service provider's data centre raises compliance issues that need to be addressed. As data protection laws vary from country to country, it is entirely plausible that the transition to the data centre changes the regulations under which the data is governed. Such jurisdictional issues need to be investigated, with any compliance failings rectified. Alternatively, Compliance-as-a-Service offerings can be employed to address such issues.

7.4.5 Data mobility

Given the youthful nature of the SaaS market, many service providers will not have a proven track history, and inevitably some will fail. In such a scenario, the enterprise data entrusted to the service provider will need to be maintained and protected. During the initial investigation of moving towards a SaaS offering, the possibility of such an event needs to be addressed, with an exit strategy in place to preserve the enterprise's data.

7.4.6 Integration

Enterprises that employ multiple SaaS applications, or need to associate cloud based software with already existing on-premise applications, often encounter software integration issues. In the event of an inability of the enterprise to utilize the provided application programming interfaces and data structures to perform the integration, an Integration-as-a-Service solution may need to be considered.

7.5 Differentiating between SaaS and ASP

The concept of Software as a Service is essentially an extension of the failed Application Service Provider model, but with a number of significant changes [9]:

- While most ASP vendors offered a service provider arrangement for third party applications, SaaS vendors are predominantly responsible for creating and managing their own applications.
- Most ASPs offered the possibility of hosting traditional client server applications, which required the installation of application specific software to communicate with the ASP. By contrast, the vast majority of SaaS applications communicate via the internet, and the only user software needed is an internet browser.
- Finally, most ASP providers deployed one instance of a software application for each of its customers. This is not the case for SaaS applications, where a SaaS provider will typically provider a multi-tenant architecture, meaning a single instance of their application will be used to serve all of the customers that utilize that service.

The failure to differentiate between SaaS and ASP has already lead to some negative perceptions of SaaS applications, with the acronym SOSaaS being used, where the SO stands for Same Old.

8 User Application Cardinality

Before delving into integrating the design principles of SOA into SaaS applications, it's worth taking a look at the relationships between the user and the application instances.

8.1 User to Application Relationships

In traditional software applications, the relationship between the customer and the application instance is one to one. Taking a simple business application as an example, traditionally a customer purchased a copy of the application, installed and ran the application on premise, creating a one to one relationship between the customer and the application instance.

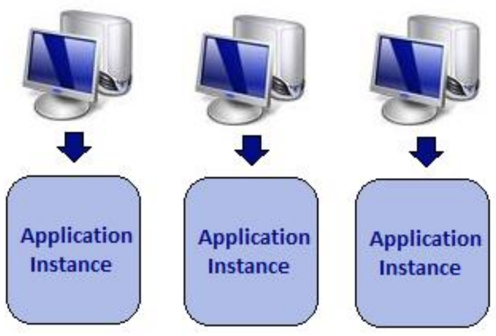


Figure 8.1

Depicted in Figure 8.1, this created a number of overheads for the customer:

- The application had to be physically installed and configured locally.
- For many applications, a certain number of specifications were required to run the application.
- In many cases, software applications were extremely expensive, with no weight being given towards light and heavy users of the application.
- Most importantly, customer data was extremely vulnerable, with data lost in many cases due to hardware failures or even theft.

In the late 1980's, the first attempts were made at reducing some of these issues. Outsourcing of software solutions became a reality, with the first generation of Application Service Providers. ASPs were responsible for hosting vendor specific client server applications in data centres, with the management of the application being handed over entirely to the provider. However, hosting vendor specific applications incurred large costs which negated the perceived benefits to the vendor for outsourcing their solution. Additionally, as the ASP did not have any expertise with regards to specific applications, vendors had to provide some internal expertise to ensure the applications functioned correctly in the ASP's data centre. As all applications hosted by the ASP were vendor specific, they were also by default, single tenant, thus capping the revenue stream to one client, and in many cases leading to the demise of the ASP.

Through multi-tenancy, the SaaS deployment model provided a fresh perspective and eradicated these issues by and large. Applications that were out of reach for many small and medium sized enterprises are now seen as viable cost effective solutions. Customers no longer had to concern themselves with application installation and hardware requirements, and were safe in the knowledge that their data was secure with their chosen SaaS provider. Along with this, many SaaS providers employ a metering service charge rather than a flat service charge, with customers paying on a per usage basis.

This model can be seen as to have a many-to-one relationship, as a single instance of the application is used by all its customers (Figure 8.2).

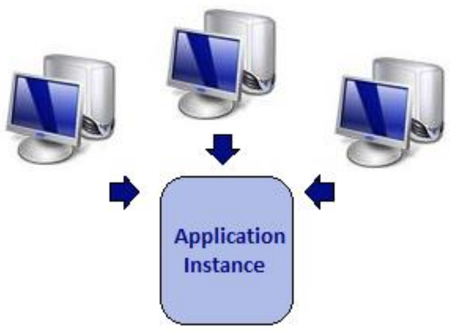


Figure 8.2

While this model has the possiblilty of bringing large scale enterprise solutions to the masses, it lacks the ability to cater for specific needs of individual customers, and indeed has many resemblances to the failed ASP solutions of the past.

8.2 Its time for ... SOA

While SOA is not required to provide a SaaS solution, it does provide the ability to build composite applications and customize solutions much more efficiently. Indeed, without the existence of an underlying SOA, the scalability and flexibility of the application can be severely restricted. This lack of flexibility is what differentiates Application Service Provider (ASP) solutions from SaaS solutions.

By contrast, a SOA based SaaS solution is multi tenant, providing the potential to serve a large number of customers with their software requirements from the same instance of a single application. Furthermore, the underlying architecture and design principles allow applications to scale and incorporate changes at a minimal cost. SOA becomes invaluable for interchanging different software components as it is essential that a SaaS provider has the ability to offer a solution that is flexible enough to meet the customers' needs.

As shown in Figure 8.3, SOA has the ability to create a many-to-many relationship between the customer and the application instance

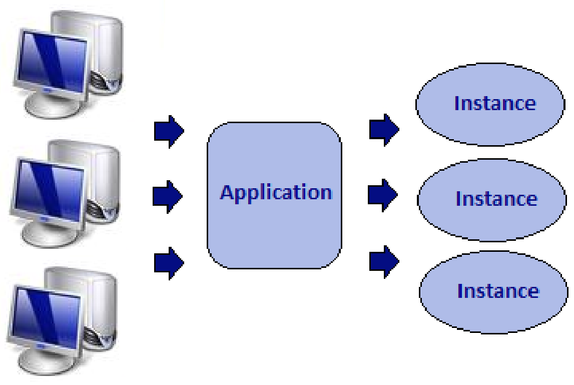


Figure 8.3

This is facilitated through the service orientated nature of the application based on SOA design principles. Rather than create a single application that satisfies a number of use cases, the application is developed as a core applications and a number of services in accordance with the previously outlined SOA design principles.

The underlying architectural progression to move from one to one, towards a many to many relationship is discussed in the next chapter.

9 Combining SOA and SaaS - Architectural Progression

To evaluate the progression towards a combined SOA-SaaS architecture, attention needs to be paid to the architectural progression that leads towards each.

9.1 Traditional Architectures

In the early days of application development, no consideration was given to the possibility of reusing any of the software that was constructed. Logical layers were not considered, with user interface, business logic and data access, all contained and spread throughout a single application. One of the few architectural similarities that early applications had with modern enterprise application was of data storage across multiple formats (Figure 9.1).

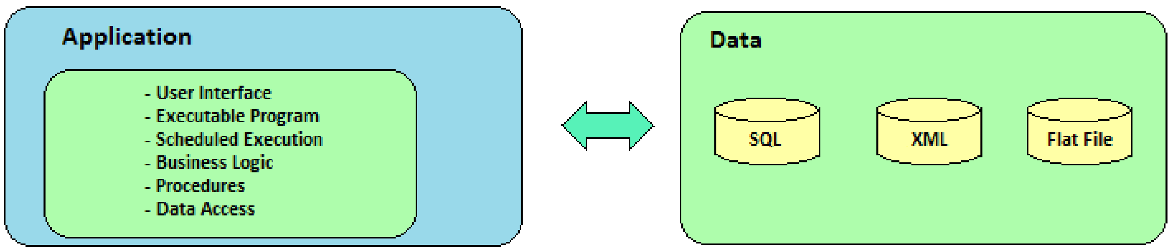


Figure 9.1

In many cases, the implementation of the data access was achieved through the use of in house languages, and tightly coupled to the business logic and user interface. This architecture was suitable for small hand written applications, as they could be implemented by a single person and in a timely fashion. However, issues such as reusability, maintainability, security and scalability needed a fresh approach towards application architectures and software engineering in general.

9.2 Composite Applications

With large scale application development, traditional architectures were replaced by component based applications. Rather than having code interwoven throughout an application, individual layers for presentation, business logic and data access were created. This re-architecting the application into layers can be seen as the first step towards creating reusable components as depicted in Figure 9.2.

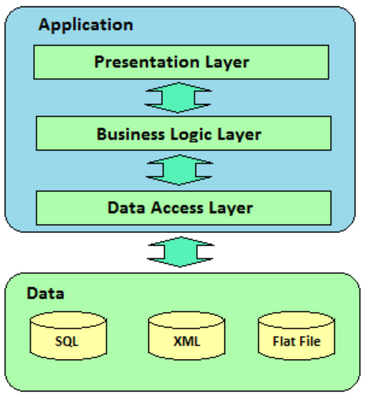


Figure 9.2

This architecture enabled the distribution of implementation tasks across a number of programmers, leading to systems that were more maintainable, robust and scalable. Indeed, at a high level, this architecture was the standard application blue print for over a decade.

The rise of distributed application revealed a number of limitations with the component based architecture. As components may be written in different languages, communication between applications became an issue. Even on overcoming the language barrier through the use of CORBA, data sharing is still an issue. In many cases, business logic and data access logic was only accessible through the user interface, limiting the communication between applications. Overcoming this through implementing application specific integration code within the communicating application created a tight coupling between both applications which was highly undesirable.

To facilitate communication between different applications, a common communication mechanism needed to be facilitated. Also, instead of considering the development of components, the architectural mentality shifted towards creating services.

9.3 SOA Applications

Following on from the layered application architecture, the definition of services can be seen as a granular progression of the architecture that preceded it. While the service will still contain the business logic and data access code, the presentation layer does not exist within a service. This decoupling of the presentation logic from the business and data access layers allows the service to only concern itself with how the service should behave, with no concern for how the information returned was handled (Figure 9.3).

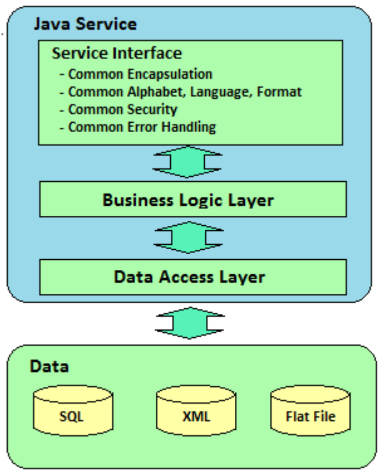


Figure 9.3

To provide this functionality, a "Service Interface" is created as an access point into the business logic. The service interface is as such a wrapper for the business logic, governing how the service may be accessed.

As a service is designed using open standards, services can be accessed by each other even though they might be running on different platforms (Figure 9.4). The service is now available for use by any user interface that requires the service. The only requirement enforced upon the user interface is the knowledge of how to use the communication method implemented by the service such as REST or SOAP.

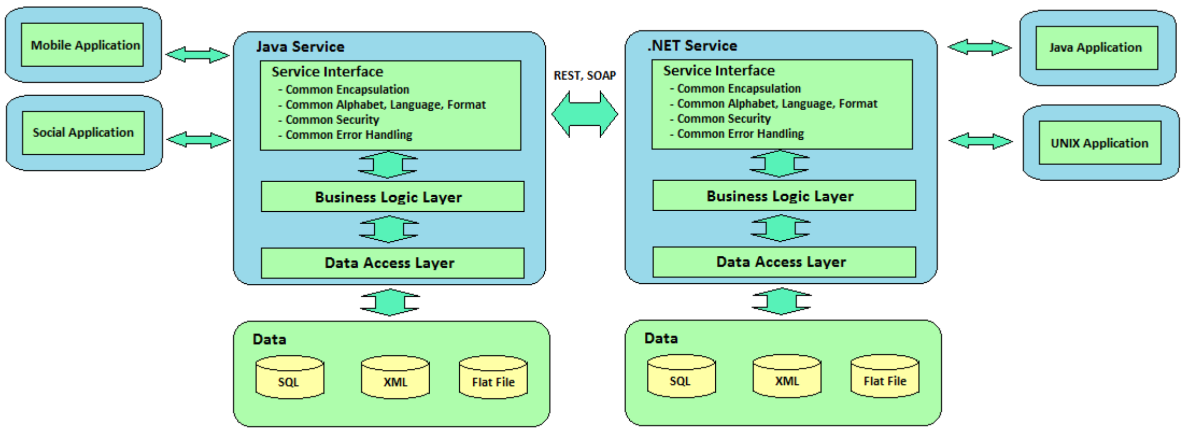


Figure 9.4

The presentation layer, whether for consumption by an end user or another application, is designed to interact with and utilize the service. With this architecture, the presentation layer is completely decoupled from the service, and is platform independent. In the diagram above, consumers build on different platforms can avail of the service, along with inter service communication, despite the fact that they run on different platforms. Switching between different technologies is not an issue as long as the communication method remains unchanged.

9.4 SOA in Enterprise Applications

The SOA architecture revolves around building services that interact with many different business components, with each service defining a specific business function. Business processes are realized by the interaction of a number of services and their workflow. This service orientated design is essential for the flexibility of an application. By designing, implementing and exposing functionality as services, business functionality can be reused and shared regardless of physical location or platform deployment.

By correctly adopting a SOA design strategy, a number of advantages are realized, that is beneficial in software development in general and could be applied to most software development projects [28]:

- Encapsulated Complexity - The inner workings of the complex services are hidden from the outside world. The service consumer simply thinks of the service as a black box and is only concerned with its service interface.
- Mobile Code - The location and platform of the service is not a concern for a service user once a common communication mechanism is used.
- Focused and Parallel Development - As the layers are defined and in some cases decoupled, developers can work on specific tasks without impacting on other developers on the same project.
- Multiple Client Types - As the presentation layer is not tightly coupled to any platform, multiple client types can avail of the service offering.
- Additional Security - Services that are more vulnerable than others can be placed behind a firewall, while those that are not susceptible to malicious attacks can be treated as less of a security risk.
- Reusability - As the services are exposed and made available, the reuse of services has become a reality rather than just a best practice when developing applications.

Indeed, the last point on service reusability is one of the major features of Software-as-a-Service applications.

9.5 SaaS Applications

Unlike SOA which is a development method, SaaS is a delivery method for software applications, more often than not across the internet. Rather than having to install, host and maintain an application, the end user simply accesses the application through the web, and are not concerned with the complexities of hardware and software management.

As previously discussed, a multitenant architecture is the cornerstone of any SaaS application. Unfortunately and all too often, once an scalable infrastructure and a multi-tenant database has been implemented to support a component based application, the architecture of the application is finished, as, in theory, it meets all the use cases as specified for the required SaaS application. In doing so, the design principles and best practices that were refined through years of SOA development, are discarded without further thought (Figure 9.5).

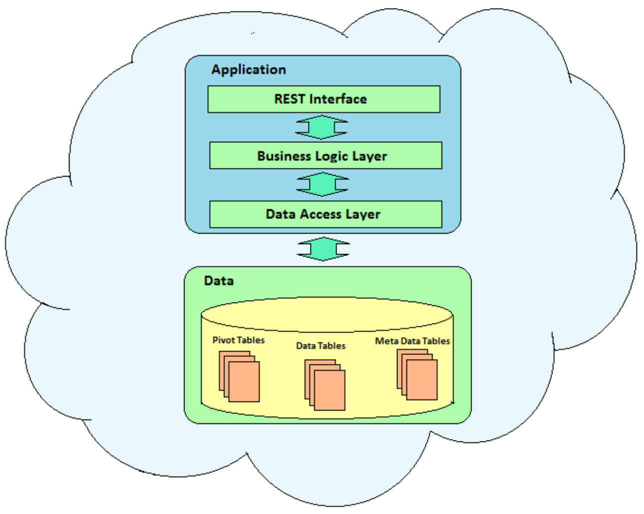


Figure 9.5

However, the SOA design principles are as valid now as they have been in the past for developing the most basic SaaS solution. In a traditional application deployment scenario, a fixed charge was in place for an on-site customer application deployment, or via licensing. Customer management and accounting took place in a different department for the software provider, and the application usage by the customer was not a concern for the software provider once it stayed within the specified limits of the application. With the move to a SaaS deployment model, all of these are issues for the service provider and need to be addressed through the provision of Business Support Services.

9.6 Combining SOA and SaaS

In providing the support for these business use cases, a service provider will need to utilize a service orientated approach, keeping the application code completely decoupled from the business service code. This facilitates reuse should the service provider have a number of applications each requiring similar business services. This can be seen (Figure 9.6) as a service platform that is needed to facilitate an application to be deployed as a SaaS offering.

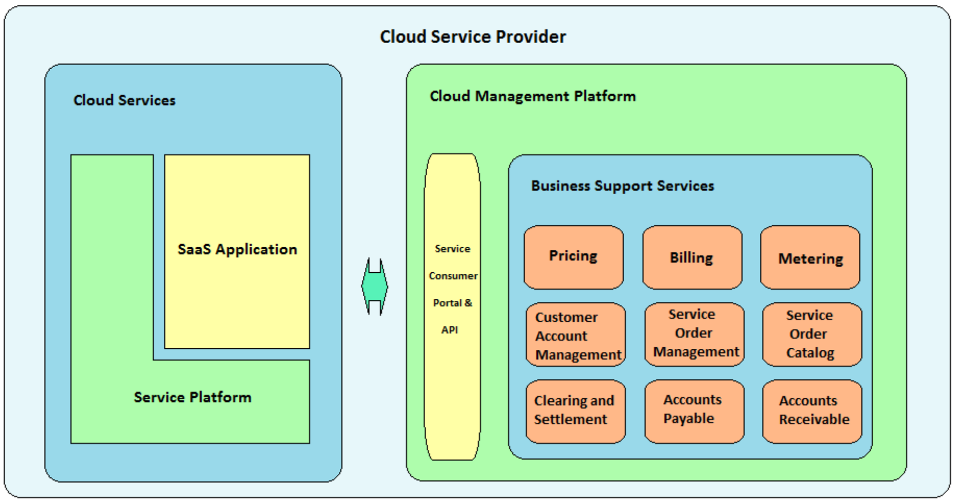


Figure 9.6

Because of SOA design principles, constructing a business service platform is not as daunting as it may first seem. Many companies such as Google and Oracle offer platforms that make deploying a SaaS application relatively straightforward. A service provider may also wish to build their own platform from an IaaS starting point, with the ability to choose individual business services from other service providers or to build their own business services.

Even though the diagram gives the impression of a fully-fledged service orientated offering, the actual application is still little more than a composite application. To move towards a true service orientated solution, the business layer of the application needs to be investigated.

9.7 Service Orientated Business Layer

Applying SOA design principles to the business layer, and decoupling the business functionality from the core application, will lead to a much more scalable, flexible and maintainable SaaS application. If one business service needs to be replaced at some point in the future, the overall impact on the application as a whole will be negated, as only the particular business service being replaced will be impacted.

Taking a typical sales application, the following architecture (Figure 9.7) is often the case:

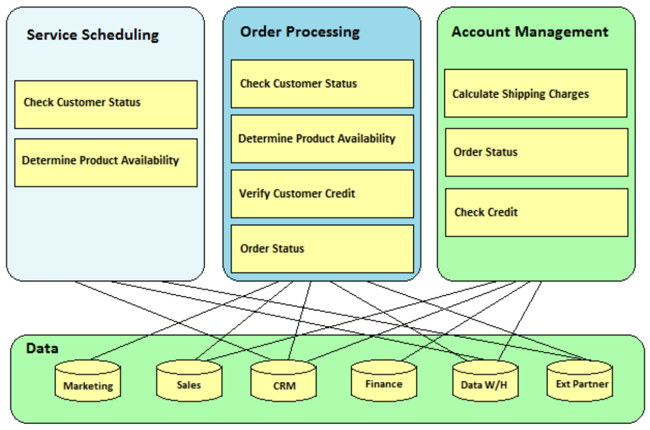


Figure 9.7

In this example three different business processes are supported. However, as each of the processes is only concerned with their own functionality, some of the common functionality is repeated amongst the three. This presents overheads for development and maintenance, as functionality updates may need to be replicated in multiple locations across the code base. Also, making a change in one area of the business process may have adverse effects on other parts of the same process as the changes are not confined to services.

An alternative architecture (Figure 9.8) is to structure the SaaS application using a SOA:

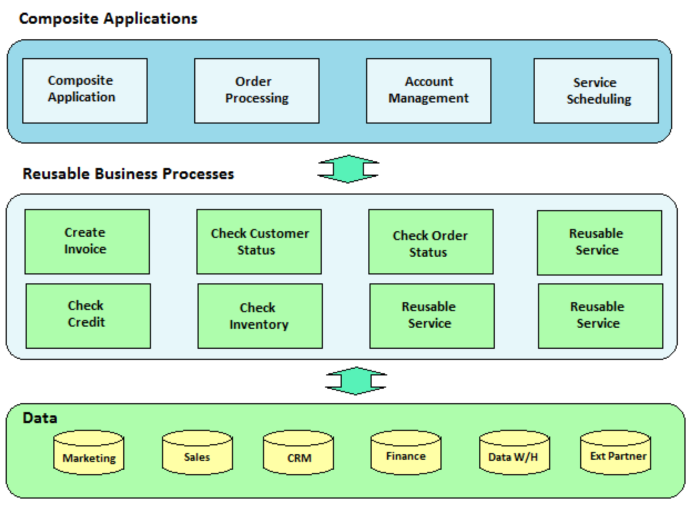


Figure 9.8

When developed using a SOA, the improvements in the overall architecture are extremely visible. Reusable business services lead to a much more maintainable solution, with any changes only impacting the individual business services.

Code duplication is removed as each business service delivers a specified piece of functionality as outlined by its service interface.

9.8 SOSaaSOA Applications

With a SOA based SaaS application, the service offering starts to provide extra flexibility for the service provider. The SOSaaSOA architecture is outlined in Figure 9.9.

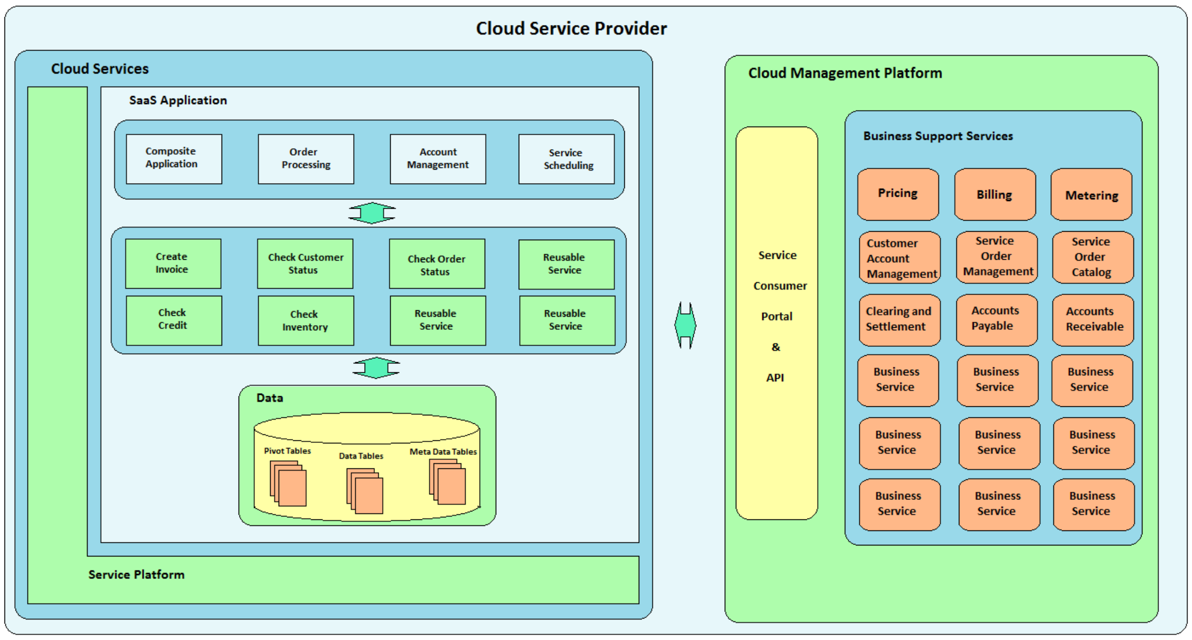


Figure 9.9

Aside from benefiting the development team with a clean concise architecture, there are also a number of significant business benefits to the service provider.

9.8.1 Many to Many

One of the key advantages of employing a SOA solution is the ability to provide composite applications. This is particularly important when it comes to realizing the true potential of a SaaS application. While SaaS solutions differentiate themselves from ASP solutions by enabling many customers to utilize the same application, the SOSaaSOA architecture differentiates itself from SaaS by enabling many customers to utilize many applications through the one service offering.

This opens up the functionality of a large scale enterprise application, that was previously unaffordable to small and medium sized enterprises that only require a fraction of the functionalities provided by the SaaS offering. As the composite applications are composed based on the customer's business service needs, the architecture brings the functionality of the large scale application to a much larger customer base.

9.8.2 Business Service as a Service

In the same way that a service provider can avail of on-demand Business Services such as billing and metering, with the implementation of a SOA architecture for its SaaS offering, the application's business services can now be offered as a service, with a potential for additional revenues. In the example outlined, services such as invoice creation or credit checks may be viable to expose as services. Though exposing the service will involve a service level support from the platform, it will allow for an extra return on the SOA investment for a relatively small overhead.

10 Architecture in Action

10.1 The Requirements

To truly evaluate the benefits envisaged by the aforementioned architecture, a proof of concept application needed to be developed that met a number of key requirements:

- Service Orientated - The application architecture needed to be service based, giving an end user the possibility of availing of just the services that were required on a subscription basis.
- Published Services - The services that are the building blocks of the application need to be accessible to external applications as a possible additional source of revenue.
- Stateless Messaging – To enhance the scalability of the application, no client context can be stored on the server between requests. Any request from any client must contain all of the data necessary to service the request, with session state held solely on the client.
- Cloud Deployment - As the application is an enhanced version of SaaS, the application will need to be deployed on a cloud platform.
- Multi-tenancy – As the application will need to support multiple clients, a multi tenant database will need to be implemented, preferably using a common schema for all end users.

10.2 The Framework Components

When designing an architectural proof application, an effort was made to use some of the latest industry standard open source technologies.

10.2.1 Spring

The Spring Framework provides an extensive programming and configuration model, that is completely independent from any deployment platform, for java based applications. One of the key elements of spring is its application level infrastructural support. Through the use of inversion of control, spring handles the dependency management and the underlying configuration of the application, allowing development teams to focus on the business logic of the application, and eliminates the overhead of tailoring a solution for specific deployment environments [26].

Some of the main features of spring that are used by the proof of concept application are:

- Dependency Injection.
- Spring's AOP support to handle cross cutting concerns.
- Spring's MVC and RESTful web service framework.
- Spring's integrated support for JDBC and JPA.

10.2.2 Hibernate

Developed specifically for java based applications, Hibernate ORM is an object-relational mapping library, that provides a framework to facilitate the mapping of an object-oriented domain model to a traditional relational database. Hibernate addresses object-relational impedance mismatch issues by providing a layer of high-level object handling functions, replacing direct persistence-related database accesses within the application code.

The primary feature of Hibernate is to map entity java classes to database tables, convert data types from java to SQL and to facilitate data query and retrieval. By generating SQL calls, Hibernate relieves the application developer of manually handling result sets and object conversion. Applications that implement Hibernate for relational mapping are portable to supported SQL databases without incurring a significant performance overhead [11].

10.2.3 Querydsl

Querydsl is a framework which enables the construction of type-safe SQL-like queries for multiple back ends including JPA, MongoDB and SQL in Java. Instead of writing queries as inline strings or externalizing them into XML files they are constructed via a fluent API [20]. Using Querydsl in conjunction with Hibernate gives a number of advantages:

- As all calls are via java APIs, code completion can be utilized in the IDE.
- Syntactically invalid queries are minimized.
- Both properties and domain types can be referenced safely.
- Provides a simple mechanism for re-factoring changes in domain types.

10.2.4 PostgreSQL

PostgreSQL is an open source object-relational database management system, and is widely regarded as one of the most advanced open source database systems available. Originally designed to run on UNIX platforms, PostgreSQL was developed at the Berkeley Computer Science department in the University of California, based on POSTGRES 4.2. It was only at a later stage that PostgreSQL's portability was introduced, allowing it to run on various platforms such as Windows, Mac OS X and Solaris.

With a liberal open source license, developers are permitted to use, modify and distribute PostgreSQL as they need or see fit. Due to its stability, PostgreSQL requires little maintenance efforts, providing a total cost of ownership that is relatively low in comparison to other database management systems. Unlike some other open source alternatives, PostgreSQL is regarded as being extremely reliable because it is ACID (Atomicity, Consistency, Isolation, and Durability) compliant, guaranteeing the integrity of the data queries [19].

10.2.5 ExtJS

ExtJS is a web application framework written in javascript and provides support for DHTML, Ajax and DOM scripting. ExtJS includes interoperability with Prototype and jQuery, and includes advanced charting and graphing capabilities without relying on external plugins. ExtJs also provides cross platform browser compatibility, with browser specific implementations handled by the framework [24].

10.2.6 RESTful Communication

REST is a stateless, client-server, cacheable, communication architectural style. Rather than using complex mechanisms such as CORBA, RPC or SOAP to handle the communication needs of an application, simpler protocols such as HTTP can be utilized. Typically, RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST can use HTTP for all four CRUD (Create/Read/Update/Delete) operations [6].

10.2.7 Apache Maven

Primarily used for Java projects, Apache Maven is a build automation tool that addresses two aspects of building software - how the software project is built, and the dependencies contained within the project. Contrary to preceding tools like Apache Ant which require detailed build instructions, Maven uses conventions for the build procedure, and only exceptions need to be specified. A project object model (POM) XML file dictates how the software project is built, specifying the project's dependencies on external modules and components, the project's required plug-ins, build order and directory structure [1].

10.2.8 Cloud Foundry

Cloud Foundry is an open source platform as a service, publicly available on github, providing a cloud deployment platform that supports a number of developer frameworks and application services. Versions of the platform are available through a variety of private cloud distributions and public cloud instances, with Pivotal providing one such instance [4].

Pivotal CF Hosted, available at run.pivotal.io, supports JVM based languages such as Java, Groovy, and Scala, as well as Node.js and Ruby. Supported frameworks include Spring and Play for Java, Lift for Scala, Grails for Groovy, and Rails and Sinatra for Ruby. It also supports a number of database and messaging implementations such as Postgres, RabbitMQ, MongoDB, MySQL and Redis through the Pivotal CF Marketplace.

10.3 The Application

With the key requirement being a service orientated cloud offering with a many to many application cardinality, the domain of Business Management was chosen for an architectural implementation proof of concept. Depending on the end user requirements, the application can be a customer management application, a product management application, a payment tracking application, a sales management application, purchase management application, or indeed various different combinations of each. Individual services provided by the application can be availed of on a subscription basis, providing the end user with a custom application to meet their individual requirements. The end product is the "Your Business Solutions" (YBS) application.

Taking a service oriented approach, with an emphasis on delivering each service as an individual business process, gave a true insight into the proposed SOSaaSOA architecture. Adhering to the core principle of SOA, each service needed to be designed and implemented as a highly cohesive loosely coupled component, the functionality of which could be availed of, not only by the YBS application, but also by external applications. By using RESTful communication between the web front end and the underlying web server application, all communication is stateless, with application functionality made available through simple HTTP requests. Also, clearly defined interfaces to the web server provide end users with a contract of service provided by the YBS application.

Due to its promotion of loose coupling though dependency, spring was chosen as the application framework. Also, though the use of aspects, many common code components can be removed from business components as cross cutting concerns, giving a more maintainable POJO based application. All database transactions are handled using Hibernate, with an additional layer provided by Querydsl to further simplify the database communication by combining the dynamic capabilities of a criteria query with JPQL in a type safe manner.

A PostgreSQL multi tenant database persists application data for every end user in a centralized repository. The application is deployed on Pivotal's Cloud Foundry platform, resulting in a high level architecture as outlined in figure 10.1.

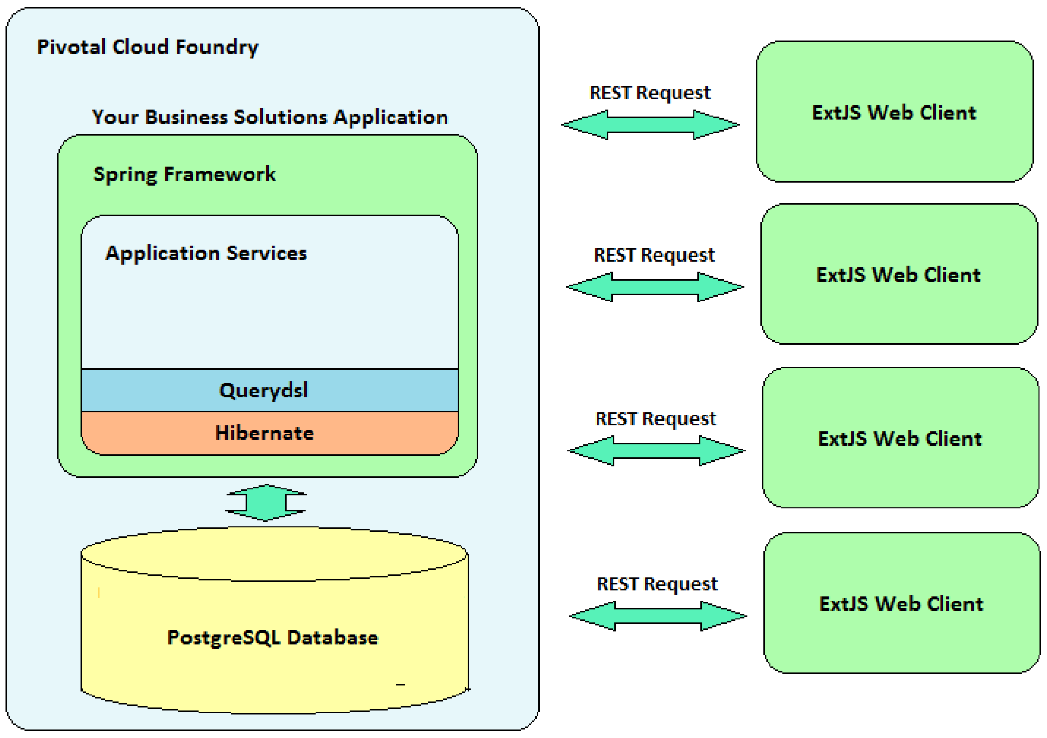


Figure 10.1

Rather than existing as a single monolithic application, the YBS application itself is composed of a number of services that run within the spring container. The services that are available to the end user are chosen during the subscription to the application, with different service combinations resulting in an application that is customized to meet the end users requirements. The application also avails of the spring security service for end user authorization and authentication, with MD5 used for password encryption.

10.3.1 Application Services Architecture

Each business process within this application is written as an individual service, though with a common architecture as depicted in Figure 10.2. The architecture can be broken down into three distinct layers

- Client Interface Layer – This handles all incoming client requests, and delegates the processing of the request to the appropriate business layer component.
- Business Layer – This is responsible for processing requests from the Client Interface Layer, and querying the Entity Layer for persistent data.

- Entity Layer – This handles all communication with the persistent items in the database, with all requests coming from the Business Layer.

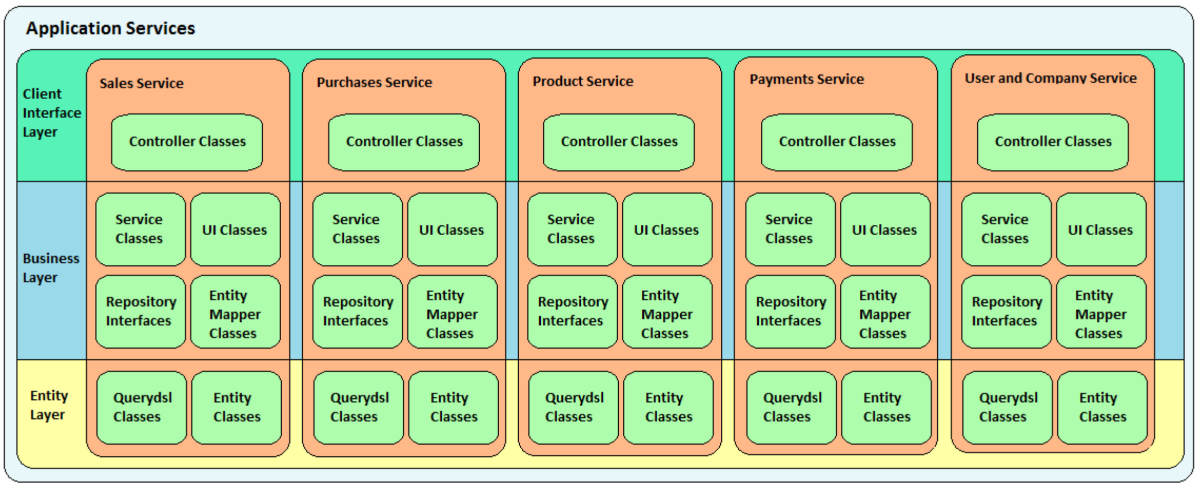


Figure 10.2

By separating the functionality across layers, internal changes to individual components within each service will have less of an impact on other components within the service, thus reducing the cost of change at a later date.

10.3.2 Service Composition

Along with being designed in a layered architectural model, each service is constructed from a number of components, as depicted in Figure 10.3.

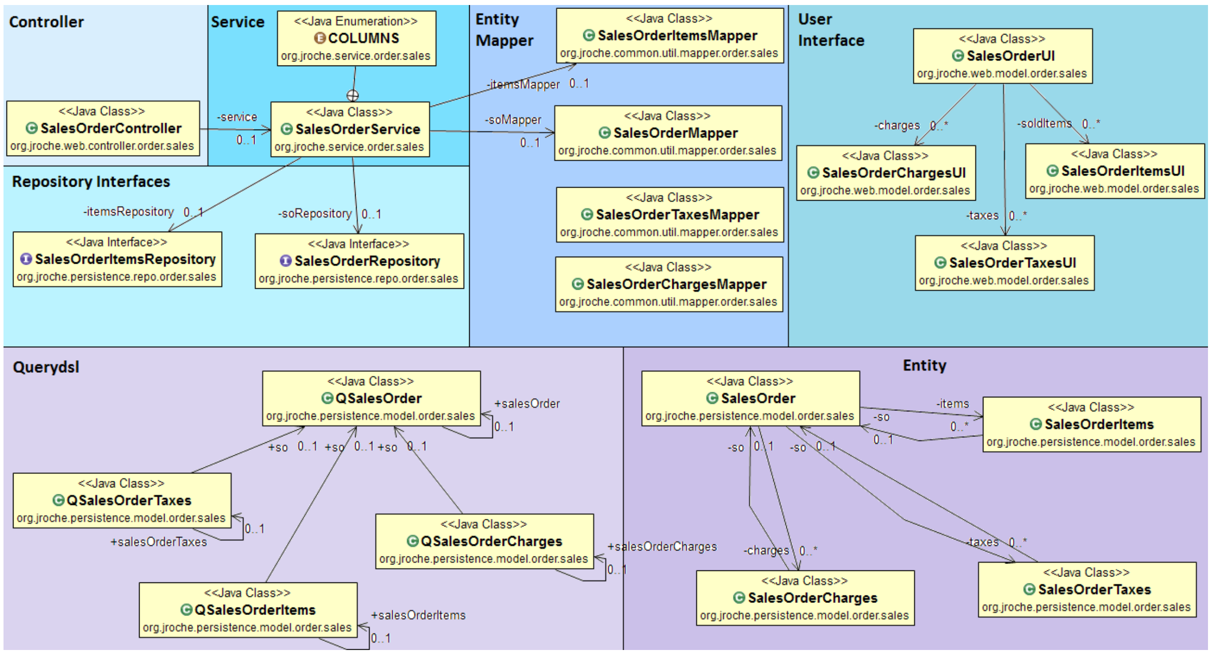


Figure 10.3

Though in this example the sales service is depicted, class diagrams for each service are available in the Appendix.

10.3.2.1 Controller

The Controller is responsible for handling all client requests for the service and passing the requests to the business layer. Each controller class is annotated with two annotations, @Controller and @RequestMapping. The @Controller is a child of the @Component annotation, allowing auto discovery by spring via classpath scanning, thus passing the instantiation of the component to the spring framework. The @RequestMapping specifies what HTTP request is handled by the Controller class.

10.3.2.2 Service

The service component carries out the business functionality of the service. Each service is annotated with the @Service annotation, which is also a child of the @Component annotation, allowing auto discovery and inversion of control. The service component is responsible for receiving the request from the controller, querying the entity layer and creating the user interface objects needs for the response to the client.

10.3.2.3 Entity Mapper

The Entity mapper component is responsible for converting an entity object to a user interface object and vice versa.

10.3.2.4 User Interface

The User Interface component consists of a collection of client side representations of entity objects.

10.3.2.5 Repository Interface

Custom interfaces extending from both JPA and Querydsl to provide a more maintainable mechanism for performing database queries.

10.3.2.6 Querydsl

A set of Querydsl query type classes, with a one to one mapping between each Querydsl and entity class.

10.3.2.7 Entity

Comprises of a set of entity classes, each of which represents a table in the underlying database. Each class is annotated with @Entity and contains an @Id, enabling JPA to create this mapping, and also identify the column in the database that will be used as the primary key.

11 Architecture Integration Results

Developing the YBS application using the previously outlined SOSaaSOA architecture combines the benefits of both SOA and SaaS. Rather than being alternative approaches to application development, both can be used in unison to deliver a service orientated cloud deployed solution. The SOSaaSOA architecture is a SaaS application architecture based on SOA design principles.

By adhering to SOAs design principles, individual business processes are delivered as isolated services providing a defined functionality set. Instead of being concerned with the intricacies of the application as a whole, it is possible for independent development teams to focus on specific areas of the application, contributing services that adhere to a predefined interface.

11.1 Benefits of SOSaaSOA

As the SOSaaSOA architecture is a specific architecture for SaaS applications, it automatically avails of the benefits of such a deployment model. Only one instance of the application needs to be maintained, with a regular income from the application in the form of a subscription based revenue stream.

However, the true value of SOSaaSOA is provided by its underlying architecture. In adopting the SOSaaSOA architecture, many of the benefits of a SOA solution become engrained within a SaaS application. Each service adheres to the requirements of its interface, providing a contract of service to end users of the service. In developing each service as an independent business process, tight coupling between components is reduced. In the YBS application, the use of the spring framework further reduces the coupling between components by relinquishing the responsibility of dependency management.

Such an approach also improves the maintainability of the application, with faults being contained within individual services rather than propagating throughout the application. Future additional services are also easily integrated into the YBS application due to its architecture. As all services share a common architecture, all future services should consist of the same architecture.

Also, as springs classpath scanning mechanism is used by the application, specifically declaring the service in a configuration file is not necessary. Once the service is annotated in the same manner as the already exiting services, the spring framework will handle the service instantiation.

All services within the application are sufficiently abstract, in that only the functionality provided by each service is exposed. The internal operations performed by the service are not visible outside of the service. Additionally, each service interface is visible outside of the application, enabling the service to be consumed by external applications.

A number of the services in the YBS application are fully autonomous, with the company/user management and product management services capable of providing a business process without the need of any additional services. Due to the nature of the services, sales purchases and payment services require the availability of user and product services.

Utilizing RESTful communication between the web client and the services controller classes ensures a stateless service. Any client side state information will not be contained within the service offering, removing the need for any state information maintenance within the service.

All services within the YBS application adhere to the service composability design principle as each service may be used as a component of an alternative application. If the end user wishes to use the YBS application as a payment management service, then user and payment services will be used by the application, whereas a more complex application may avail of additional services such as product, sales and purchases.

Overall, by applying the SOSaaSOA architecture, the YBS application delivers a solution that is tailored for the requirements of the individual end user. Additional features can be integrated with minimal impact on the already existing services. The application is also much more future proof, with potential issues isolated to individual services, and a common service architecture enabling a far reduced learning curve when developing an understanding of the operations of existing and future services.

12 CONCLUSION

Contrary to popular belief, SOA principles are far from dead, and are every bit as applicable today as they were when SOA carried the hype factor now associated with cloud and as-a-service applications. With a less than positive legacy, the term service orientated architecture regularly sends shivers down the spines of company executives, with many proponents of the concept choosing to eliminate the acronym from any discussions where it may be applicable. In the majority of the cases when SOA implementations failed, the causes of failure were more commonly due to a lack of understanding of the concept leading to incorrect application implementation rather than the failings of SOA as a concept. In effect, its own hype turned out to be its worst enemy.

Today, due to hype of a similar nature, the focus of many enterprises has turned towards cloud and as-a-service applications. Without due consideration, many as-a-service applications, despite the best intentions during inception, can turn into expensive ASP like solutions that offer little return on investment. Simply deploying an application in the cloud rather on a locally hosted server provides a very limited benefit, with the cost of the change possibly negating any benefits from the cloud deployment.

Rather than concentrate on just the deployment of the application when moving towards an as-a-service solution, the architecture of the application should be analysed in depth to ensure that the proposed application provides the best return on investment. Simply offering a large scale enterprise application on a usage basis does not release the maximum potential of the application when deployed as-a-service.

By combining SOA design principles with an as-as-service deployment model, the result of which has been tentatively entitled a service orientated software-as-a-service orientated architecture (SOSaaSOA) solution, such an application could be applicable to a far greater customer base. Developing an application as a set of services which can be availed of on an individual basis, brings the application to end users that could not justify availing of the service as a whole. Such an a-la-carte offering fulfils the needs of customers requiring a complete service solution, while also transforming the application into a viable offering for end users requiring a sub set of the applications functionality.

Along with providing functionality for the enterprise service offering, consideration should be given to exposing internal services to external applications. In many instances, individual services may be applicable to third party applications, with such services being made available for use as application components. Such an implementation could provide an additional revenue stream, and increase the return on investment for the as-a-service offering.

Aside from the possible increase in revenue, implementing a SOSaaSOA solution provides a more robust and scalable application. As services are encapsulated, issues that may arise are confined to the individual service. Typically, in the event of software-as-a-service failures, every customer of the application becomes impacted. However, as not every customer may have availed of every service, the number of impacted end users is reduced, along with the potential compensation required on the event of a breach of SLAs.

Even in the event of service failure, the outage is restricted to the individual service rather than to the application as a whole, reducing the impact of the failure to the customer.

Far from being dead, SOA, and how it’s applied in the architecture of as-a-service offerings, can differentiate cloud offerings from the failed ASP service offerings of the past, and may even determine the success or failure of applications offered via the cloud deployment model.

13 Further Analysis

13.1 Migrating Legacy Applications towards SOSaaSOA

The underlying concept of this proposal is to develop an application based on service orientated architecture design principles, and subsequently deliver the solution as a software as a service application. However, this does not just apply to new applications. Existing legacy applications can be migrated in the same manner, the first step being to transition from legacy to SOA.

Converting a legacy application to SOA is generally performed through one of the following four techniques

- Wrapping – This involves the creation of a new interface into existing components to make them accessible as services to other components.
- Reengineering – Here reengineering techniques are used to contribute SOA functionality to existing legacy applications.
- Replacement – Consists of completely rewriting the existing legacy application, or indeed replacing it with an off the shelf solution
- Migration – Involves transporting the existing legacy application to a SOA environment, while keeping the legacy systems functionality and data.

While there are a large volume of studies into the transition from legacy to SOA as outlined by Khadka [12], two of the more referenced academic bodies of work are discussed in the next section.

13.1.1 Legacy Software Wrapped for Reuse

Proposed by Sneed [25], this approach involves wrapping the legacy software into an XML shell, allowing the existing functionality of the legacy application to be provided as web services. This involves three stages:

- Salvaging the underlying legacy code.
- Wrapping the legacy application code.
- Linking the new web services to their business processes.

In stage one, valuable legacy code is identified for reuse, as well as functions and business operations. Stage two involves encapsulating the identified legacy code behind a WSDL interface, with each entry being transformed into a method, and all parameters to an XML data element. A SOAP framework is then used to package the legacy code. Finally, in stage three, web services are linked to the business processes through a proxy.

13.1.2 Architecture Transformation, From Legacy to Three-Tier and Services

Supporting SOA by transforming legacy applications to a three tier architecture was proposed by Heckel et al [10]. The four step process consisted of:

- Annotate Legacy Code by Category.
- Reverse Engineering.
- Redesigning the Source Graph.
- Forward Engineering.

In the first stage of this process, each piece of the legacy application is marked as one of the three tiers – user interface, business layer or persistence layer. Reverse engineering involves constructing a source graph model from the previously marked legacy source code. The third stage remodels the source graph towards the three tier target map via transformation rules. Finally, the target graph is used along with the annotated source code to generate the target code.

14 References

[1] Apache Maven, Apache Maven Project, [online] Available at http://maven.apache.org/

[2] AstoriaSoftware, "Calculating the Value of Software-as-a-Service to Your Organization", A Business Whitepaper, 66 Bovet Road, Suite 280, San Mateo, CA 94402 USA, 2006.

[3] Bell, M., "Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture", Wiley, 1st edition, 2008, ISBN-10: 0470141115

[4] Cloud Foundry, Deploy & scale cloud applications online, [online] Available at http://www.cloudfoundry.com/

[5] Erl, T., "SOA Principles of Service Design", Prentice Hall, 1st edition, 2007, ISBN-10: 0132344823

[6] Erl, T., Carlyle, B., Pautasso, C., & Balasubramanian, R., "SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST", Prentice Hall, 1st edition, 2012, ISBN:0137012519 9780137012510

[7] Gartner IT Glossary, Software As A Service, [online] Available at http://www.gartner.com/it-glossary/software-as-a-service-saas/

[8] Gold, N. and Mohan, A., Knight, C., Volantis Systems UMIST, Munro, M., University of Durham, "Understanding Service-Oriented Software", Published by the IEEE Computer Society, 2004

[9] Hancheng, L., Design of university teaching & research resources sharing platform architecture based on SaaS", International Conference on Computer Science & Education, 2009., ICCSE'09., IEEE.

[10] Heckel, R., Correia, R., Matos, C., El-Ramly, M., Koutsoukos, G., & Andrade, L., "Architectural transformations: From legacy to three-tier and services", In Software Evolution (pp. 139-170), Springer Berlin Heidelberg, Germany., 2008.

[11] Hibernate ORM, Idiomatic persistence for Java and relational databases, [online] Available at http://hibernate.org/orm/

[12] Khadka, R., "Service Identification Strategies in Legacy-to-SOA Migration", Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands., 2012.

[13] Král, J., & Zemlicka, M.,"Popular SOA Antipatterns", In Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009., Computation World '09., Computation World: (pp. 271-276)., IEEE.

[14] Laplante, P.A., Penn State University, Zhang, J., Illinois University, Voas, J., SAIC, "Distinguishing between SaaS and SOA", Published by IEEE Computer Society, 2008.

[15] Linthicum, D., "Perfect fit: The cloud and SOA - but don't call it that", [online] Available at http://www.infoworld.com/d/cloud-computing/perfect-fit-the-cloud-and-soa-dont-call-it-190520

[16] Lu, Y., & Sun, B., "The fitness evaluation model of SaaS for Enterprise Information System", IEEE International Conference on e-Business Engineering, Macau, 2009.

[17] Manes, A.T., "SOA is Dead; Long Live Services", [online] Available at http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html

[18] Mietzner, R., Leymann, F. and Unger, T., "Horizontal and vertical combination of multi-tenancy patterns in service oriented applications", Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Stuttgart, 70569, Germany, 2010

[19] PostgreSQL, PostgreSQL, [online] Available at http://www.postgresql.org/

[20] Querydsl, Querydsl, [online] Available at http://www.querydsl.com/

[21] Reese, G., "Cloud Application Architectures: Building Applications and Infrastructure in the Cloud", O'Reilly Media, 1st edition, 2009, ISBN-10: 0596156367

[22] Rotem-Gal-Oz, A., "SOA Patterns", Manning Publications, 2012, ISBN-10: 1933988266

[23] SalesForce, Sales Cloud, [online] Available at http://www.salesforce.com/eu/

[24] Sencha Ext JS, JavaScript Framework for Rich Desktop Apps, [online] Available at http://www.sencha.com/products/extjs/

[25] Sneed, H. M., "Encapsulation of legacy software: A technique for reusing legacy software components", Prellerweg 5, D-82054 Argent, Bavaria, Germany., Annals of Software Engineering, 9(1-2), 293-313., 2000

[26] Spring, SpringSource Community, [online] Available at http://www.springsource.org/

[27] Szepielak, D., "REST-based Service Oriented Architecture for Dynamically Integrated Information Systems", Deutsches Elektronen-Synchrotron, Notkestrasse 85, 22607 Hamburg, Germany., 2006

[28] Tsai, W., Xin Sun X. and Balasooriya, J., "Service-Oriented Cloud Computing Architecture", Department of Computer Science, Arizona State University, Tempe, Arizona 85281 USA, Information Technology: New Generations (ITNG), Seventh International Conference, 2010

[29] Velte, A.T., Cloud Computing: A Practical Approach, McGraw Hill, 2010, ISBN 978-0-07-162694-1

[30] Wang, L. and Von Laszewski, G., "Cloud Computing: a Perspective Study", Service Oriented Cyberinfrastruture Lab, Rochester Inst. of Tech., 102 Lomb Memorial Drive, Rochester, NY 14623, U.S.A., 2008

15 Appendix

15.1 Application Code Repository

http://github.com/jonroche/sales-purchases.git

15.2 Cloud Foundry Application Deployment

http://ybs.cfapps.io

15.3 Service Class Diagrams

15.3.1 Sales Service

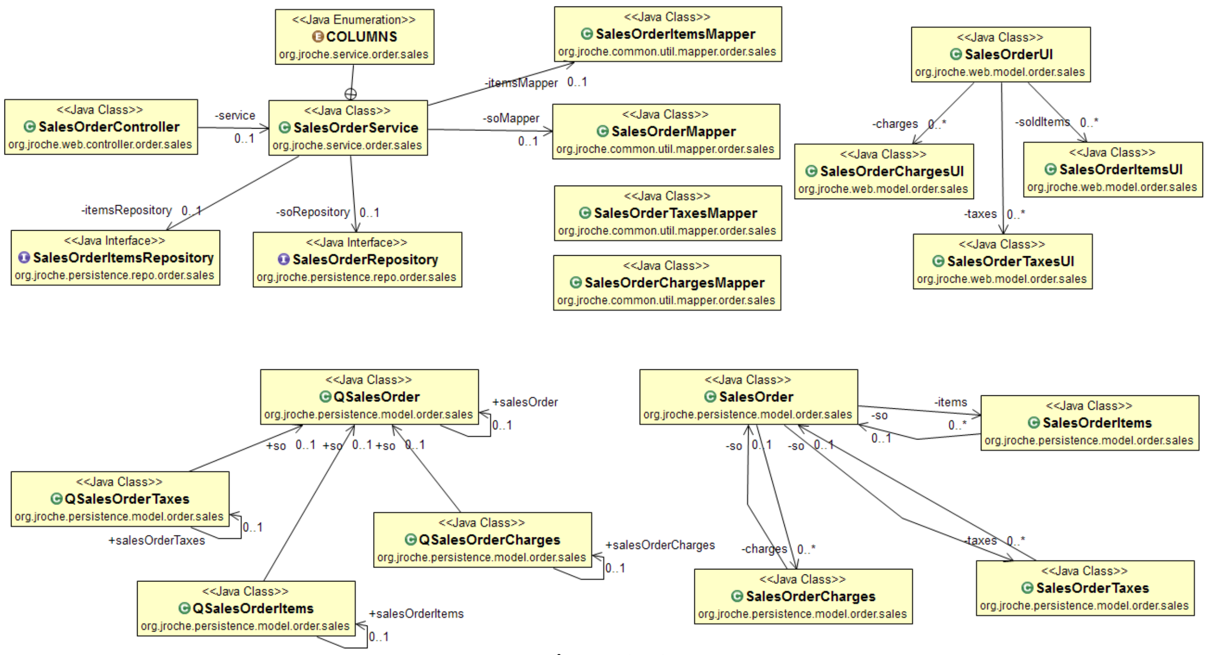
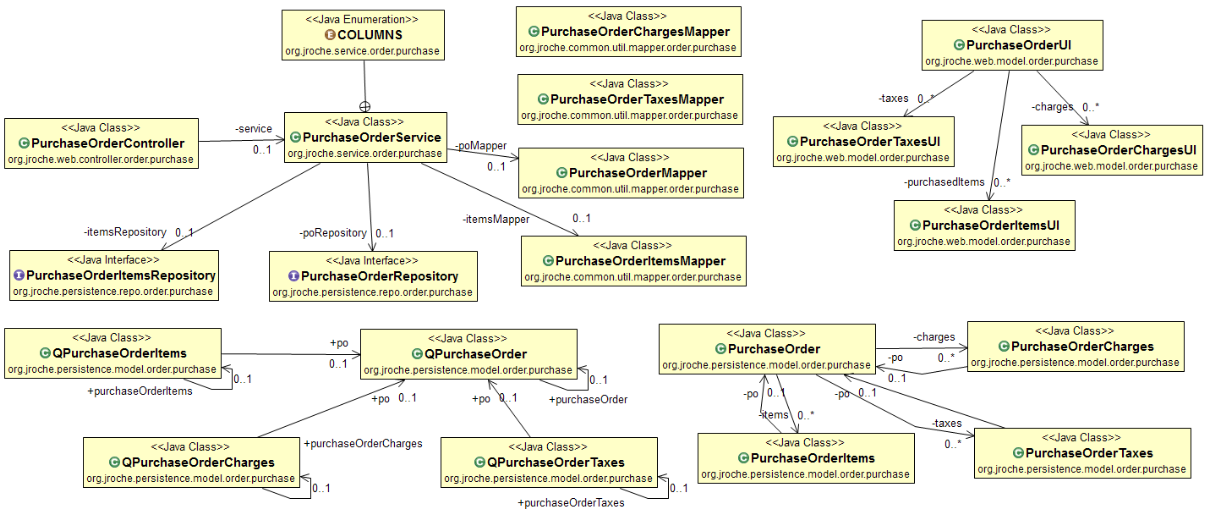


Figure 15.1

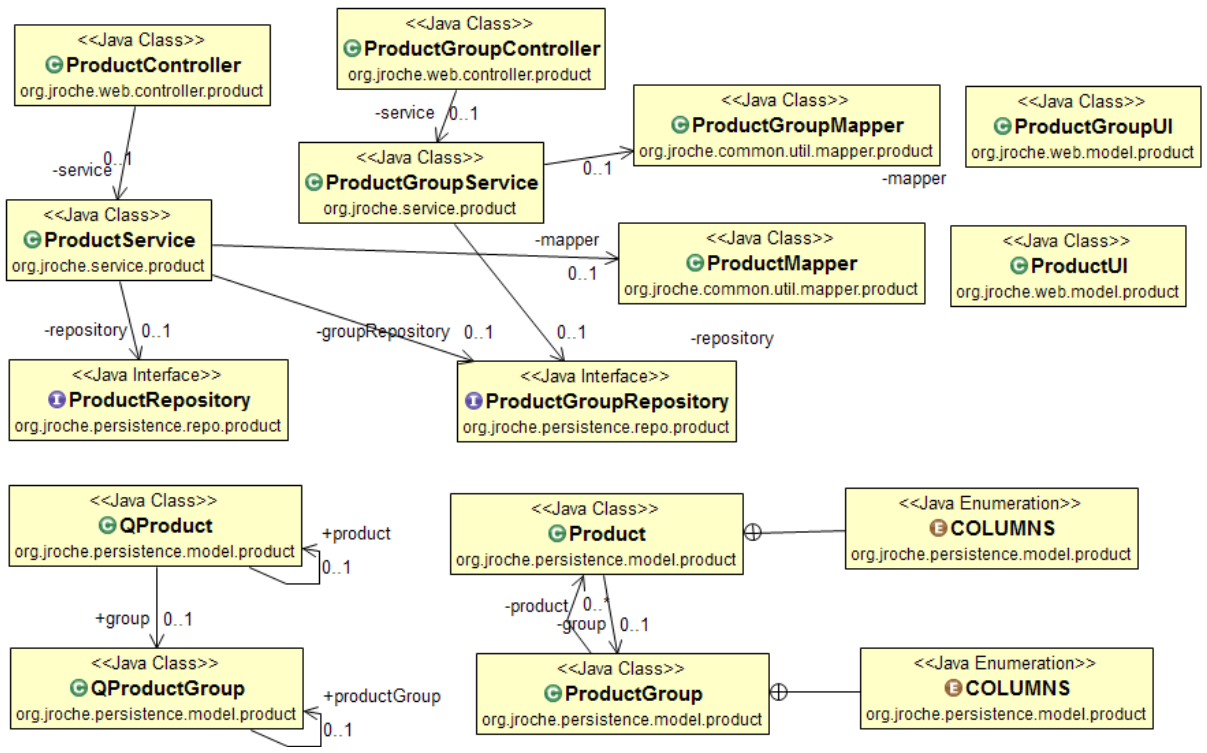
15.3.2 Purchases Service



<<Java Enumeration>> COLUMNS org.jroche.service.order.purchase
<<Java Class>> PurchaseOrderChargesMapper org.jroche.common.util.mapper.order.purchase
<<Java Class>> PurchaseOrderUI org.jroche.web.model.order.purchase
<<Java Class>> PurchaseOrderTaxesMapper org.jroche.common.util.mapper.order.purchase
<<Java Class>> PurchaseOrderController org.jroche.web.controller.order.purchase
-service 0..1
<<Java Class>> PurchaseOrderService org.jroche.service.order.purchase
-poMapper 0..1
<<Java Class>> PurchaseOrderMapper org.jroche.common.util.mapper.order.purchase
-taxes 0..*
<<Java Class>> PurchaseOrderTaxesUI org.jroche.web.model.order.purchase
-charges 0..*
<<Java Class>> PurchaseOrderChargesUI org.jroche.web.model.order.purchase
-purchasedItems 0..*
<<Java Class>> PurchaseOrderItemsUI org.jroche.web.model.order.purchase
-itemsRepository 0..1
-poRepository 0..1
-itemsMapper 0..1
<<Java Interface>> PurchaseOrderItemsRepository org.jroche.persistence.repo.order.purchase
<<Java Interface>> PurchaseOrderRepository org.jroche.persistence.repo.order.purchase
<<Java Class>> PurchaseOrderItemsMapper org.jroche.common.util.mapper.order.purchase
<<Java Class>> QPurchaseOrderItems org.jroche.persistence.model.order.purchase
+po 0..1
<<Java Class>> QPurchaseOrder org.jroche.persistence.model.order.purchase
0..1
+purchaseOrderItems
0..1
+po 0..1
+po 0..1
+purchaseOrder
<<Java Class>> PurchaseOrder org.jroche.persistence.model.order.purchase
-charges
-po 0..*
0..1
<<Java Class>> PurchaseOrderCharges org.jroche.persistence.model.order.purchase
-po 0..1
-items 0..*
-po 0..1
-taxes 0..*
+purchaseOrderCharges
<<Java Class>> QPurchaseOrderCharges org.jroche.persistence.model.order.purchase
0..1
<<Java Class>> QPurchaseOrderTaxes org.jroche.persistence.model.order.purchase
0..1
+purchaseOrderTaxes
<<Java Class>> PurchaseOrderItems org.jroche.persistence.model.order.purchase
<<Java Class>> PurchaseOrderTaxes org.jroche.persistence.model.order.purchase

Figure 15.2

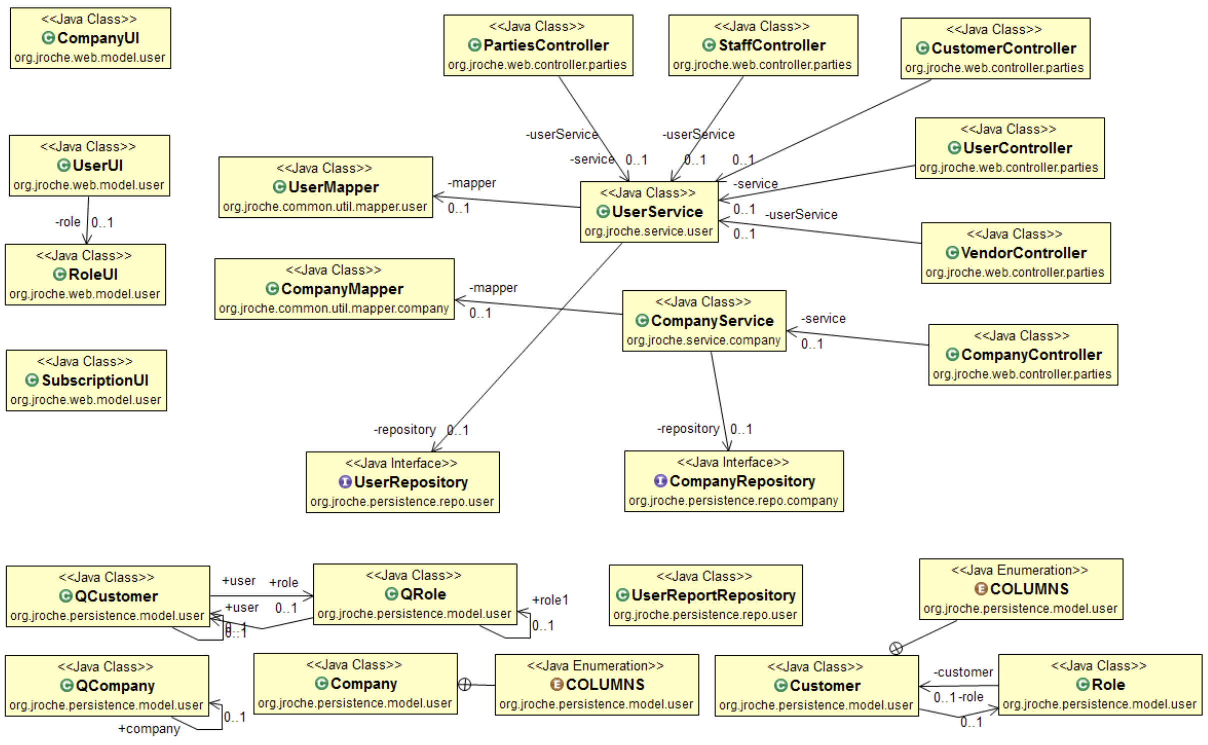
15.3.3 Product Service



<<Java Class>> ProductController org.jroche.web.controller.product
<<Java Class>> ProductGroupController org.jroche.web.controller.product
-service 0..1
-service 0..1
<<Java Class>> ProductGroupService org.jroche.service.product
0..1
<<Java Class>> ProductGroupMapper org.jroche.common.util.mapper.product
-mapper
<<Java Class>> ProductGroupUI org.jroche.web.model.product
<<Java Class>> ProductService org.jroche.service.product
-mapper 0..1
<<Java Class>> ProductMapper org.jroche.common.util.mapper.product
<<Java Class>> ProductUI org.jroche.web.model.product
-repository 0..1
-groupRepository 0..1
0..1
-repository
<<Java Interface>> ProductRepository org.jroche.persistence.repo.product
<<Java Interface>> ProductGroupRepository org.jroche.persistence.repo.product
<<Java Class>> QProduct org.jroche.persistence.model.product
+product 0..1
+group 0..1
<<Java Class>> Product org.jroche.persistence.model.product
<<Java Enumeration>> COLUMNS org.jroche.persistence.model.product
-product 0..*
-group 0..1
<<Java Class>> QProductGroup org.jroche.persistence.model.product
+productGroup 0..1
<<Java Class>> ProductGroup org.jroche.persistence.model.product
<<Java Enumeration>> COLUMNS org.jroche.persistence.model.product

Figure 15.3

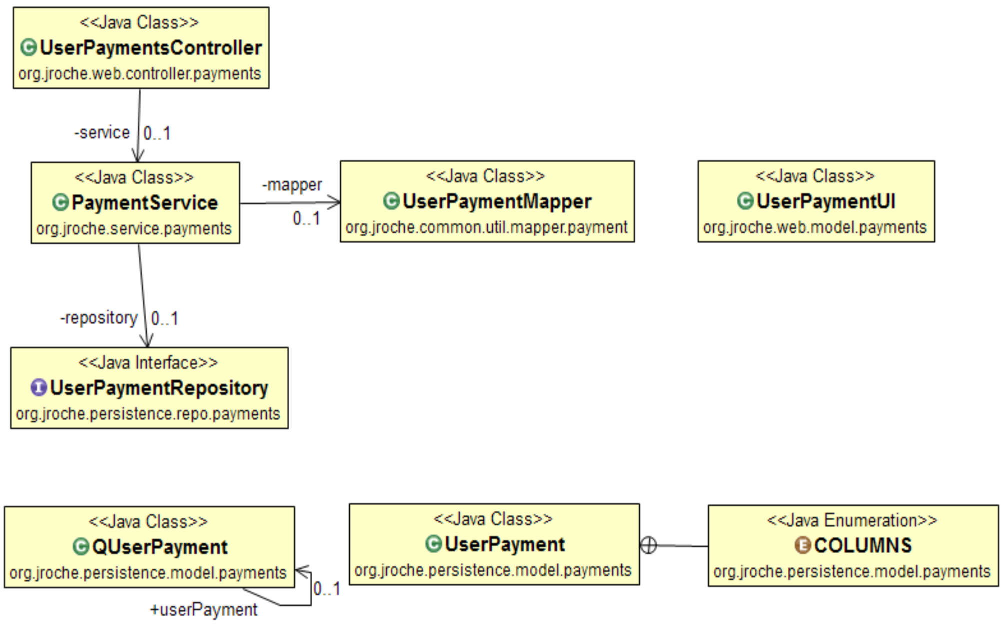
15.3.4 User / Company Service



<<Java Class>> CompanyUI org.jroche.web.model.user
<<Java Class>> PartiesController org.jroche.web.controller.parties
<<Java Class>> StaffController org.jroche.web.controller.parties
<<Java Class>> CustomerController org.jroche.web.controller.parties
<<Java Class>> UserUI org.jroche.web.model.user
<<Java Class>> UserMapper org.jroche.common.util.mapper.user
<<Java Class>> UserService org.jroche.service.user
<<Java Class>> UserController org.jroche.web.controller.parties
<<Java Class>> RoleUI org.jroche.web.model.user
<<Java Class>> VendorController org.jroche.web.controller.parties
<<Java Class>> CompanyMapper org.jroche.common.util.mapper.company
<<Java Class>> CompanyService org.jroche.service.company
<<Java Class>> CompanyController org.jroche.web.controller.parties
<<Java Class>> SubscriptionUI org.jroche.web.model.user
<<Java Interface>> UserRepository org.jroche.persistence.repo.user
<<Java Interface>> CompanyRepository org.jroche.persistence.repo.company
<<Java Class>> QCustomer org.jroche.persistence.model.user
<<Java Class>> QRole org.jroche.persistence.model.user
<<Java Class>> UserReportRepository org.jroche.persistence.repo.user
<<Java Enumeration>> COLUMNS org.jroche.persistence.model.user
<<Java Class>> QCompany org.jroche.persistence.model.user
<<Java Class>> Company org.jroche.persistence.model.user
<<Java Enumeration>> COLUMNS org.jroche.persistence.model.user
<<Java Class>> Customer org.jroche.persistence.model.user
<<Java Class>> Role org.jroche.persistence.model.user
-userService
-service 0..1
-userService
0..1
0..1
-service
0..1 -userService
0..1
-mapper
0..1
-role 0..1
-mapper
0..1
-service
0..1
-repository 0..1
-repository 0..1
+user +role
+user 0..1
+role1
0..1
0..1
0..1
+company
-customer
0..1 -role
0..1

Figure 15.4

15.3.5 Payment Service



<<Java Class>> UserPaymentsController org.jroche.web.controller.payments
-service 0..1
<<Java Class>> PaymentService org.jroche.service.payments
-mapper
0..1
<<Java Class>> UserPaymentMapper org.jroche.common.util.mapper.payment
<<Java Class>> UserPaymentUI org.jroche.web.model.payments
-repository 0..1
<<Java Interface>> UserPaymentRepository org.jroche.persistence.repo.payments
<<Java Class>> QUserPayment org.jroche.persistence.model.payments
0..1
+userPayment
<<Java Class>> UserPayment org.jroche.persistence.model.payments
<<Java Enumeration>> COLUMNS org.jroche.persistence.model.payments

Figure 15.5