OpenLearn

The Open University

An introduction to web applications architecture

About this free course

This free course is an adapted extract from the Open University course TT284 *Web technologies*: http://www.open.ac.uk/courses/modules/tt284.

This version of the content may include video, images and interactive content that may not be optimised for your device.

You can experience this free course as it was originally designed on OpenLearn, the home of free learning from The Open University – *An introduction to web applications architecture*.

There you'll also be able to track your progress via your activity record, which you can use to demonstrate your learning.

Intellectual property

Contents

Introduction

This free course, *An introduction to web applications architecture*, provides an overview of the design and implementation of computer software that runs on web servers, instead of running solely on desktop computers, laptops or mobile devices.

Web applications are software which can be accessed via a network using a web browser. Websites such as Amazon and eBay, forums and online banking systems are examples of web applications. Web services describes a standardised way of integrating web applications using the internet standards and protocols.

This OpenLearn course is an adapted extract from the Open University course TT284 *Web Technologies*.

Learning Outcomes

After studying this course you should be able to:

- understand the development of web application architecture leading to a more modular approach
- sketch out the components that would be used in a range of approaches to web application architecture
- outline a range of benefits and potential problems associated with a specific approach to web application architecture
- contrast developments in architecture (SOA, Cloud) with more traditional tiered approaches.

1 Architecture

'Architecture' is a term normally associated with the design of buildings. In the construction industry, architecture is often taken to relate to the style and structure of buildings. In the context of the architecture of web applications we will focus on structure, both in the terms of their physical components, the 'system architecture', and software (the programs making up an application), the 'software architecture'.

Let's first define what we mean by 'software architecture' by considering the following interpretations.

According to Shaw and Garlan in *Software Architecture: Perspectives on an Emerging Discipline*, (1996):

> The architecture of a software system defines that system in terms of computational components and interactions among those components.

The *Microsoft Application Architecture Guide* (2009) cites Kruchten et al's definition of architecture based on the work by Shaw and Garlan:

> Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behaviour as specified in collaboration among those elements; composition of these structural and behavioural elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns.

In *Patterns of Enterprise Application Architecture*, Fowler (2002) states:

> 'Architecture' is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change.

In *Pattern-Oriented Software Architecture, Volume 4, A Pattern Language for Distributed Computing*, Buschmann et al. (2007) cite the following quote from Booch (2007, p. 214) which emphasises that design and architecture cannot be divorced from one another:

> All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

In *Software Architecture for Developers*, Brown (2012) describes architecture more widely:

> So, we can see that the word 'architecture' means many different things to many different people and there are many different definitions floating around the internet. For me, architecture is comprised of four things:
>
> - Structure – the building blocks (components) and how they relate to and/or interact with one another.

- Foundations – a stable basis on which to build something.
- Infrastructure services – the essential services that are an integral part of whatever is being built. With a building, this might be power, water, cooling, etc. With software, this might be security, configuration, error handling, etc.
- Vision – it is crucial that you understand what it is you are building and how that process will be undertaken. Vision can take the form of blueprints, guidelines, leadership, etc.

There are quite a few other descriptions of architecture which, like those from Shaw and Garlan and Microsoft , focus on components and have similar wording to Achimugu et al. (2010):

> The foundation for any software system is its architecture. Software architecture is a view of the system that includes the system's major components, the behaviour of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the overall system's goal. Every efficient software system arises as a result of sound architectural basement. This requires the use of good architecture engineering practices and methods.

In summary, a 'software architecture' relates to the arrangement of software components and the interactions between these components. It also relates to the decisions that are taken about the design and composition of the overall system, how it is broken down into components and how these components interact. This type of structuring reflects high-level decisions that are difficult to change later because they determine software-level decisions. It is thus difficult to change the architecture of a system without starting the design from the beginning.

Activity 1 Defining 'system architecture'

Allow 15 minutes to complete this activity

Produce your own definition in answer to the question 'What is system architecture?' To do this, consider the definitions of a software architecture given above, and briefly research definitions for 'system architecture' and 'design' yourself using an internet search engine.

Answer

A 'system architecture' can be defined as the configuration of hardware, network and software components of a computer system which satisfies the requirement for the provision a particular application or service.

As with a 'software architecture', it also relates to the decisions that are taken about the design and composition of the overall system, how it is broken down into components and how these components cooperate. A 'system architecture' holds the key to success or failure of a system every bit as much as the 'software architecture' does for the software.

We will look at 'client–server' and 'multi-tier' approaches, which represent mainstream common 'tried and tested' approaches, and then 'network type' approaches, which represent more recent developments that many organisations are using because these

developments have been shown to have additional benefits over the established approaches.

By looking at the development of architecture it is possible to understand the potential benefits of adopting one approach to application design over another, and to understand why a specific architecture itself imparts specific qualities that can lead to a better (or worse) solution.

1.1 Client–server architecture

A **client–server architecture** (Figure 1) divides an application into two parts, 'client' and 'server'. Such an application is implemented on a computer network, which connects the client to the server. The server part of that architecture provides the central functionality: i. e., any number of clients can connect to the server and request that it performs a task. The server accepts these requests, performs the required task and returns any results to the client, as appropriate.

Consider an online bookstore as an example. The application allows a user to search and look at the details of a large range of books, and then to order a book. The application software provides an interface and a means of selecting or finding a book's details, as well as displaying book information and allowing a book order to be generated.

The application could take the form of a single 'chunk' of software downloaded from the web. However, if the software is one monolithic item, then every time anything is changed or updated, the entire application has to be redistributed again. Obviously this would not work well in this example because the catalogue of books will change regularly. An improvement might be to split the application into two parts. One part, the client, can provide the interface for users and be distributed to them. The other part can be kept and run on the company's own server machine (Figure 1). The client application can display information and be used to pass information to the server for searching, such as the title of a book. This client application, or software, is quite commonly called the 'presentation' layer or tier.

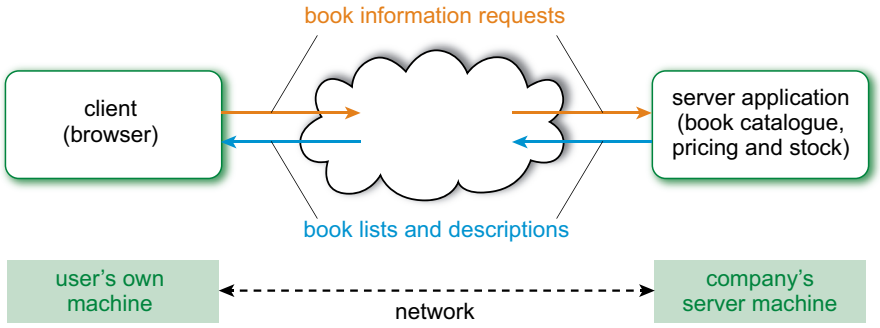

Figure 1 Simple client–server application

In this client–server model, many clients can connect to the server application and request information about books. The server has to process these requests and send the response to the client that originated the request and not to any other client. As long as the network is working well and the server can keep up with responding to all the requests it receives, such a 'split' application will provide much the same level of service as the monolithic version. This simple client–server architecture is also commonly called 'two-tier architecture'.

The catalogue of book information can be held centrally on the server and then be easily updated. This allows other 'centralised' information to be maintained and sent to clients, such as the stock level of each book. Users of the client will find it much simpler and smaller to work with than the complete application. At the same time the company will have better control and be able to, for example, monitor usage of the server application itself. A common client used to access applications is a web browser that accesses server applications (such as applications on websites) using HTTP. The use of a web browser as the client end of an application is interesting because, for most applications, the browser is provided by a third party. This means that application builders must rely on agreed standards for the behaviour of the client component.

There are also different distributions of functionality across a two-tier architecture. For instance, suppose you have a client that accesses your bank account online. If that client is a web browser, for example, it can be used to request and display your accounts' statements from the bank's server. The information you may obtain is restricted to the pre-defined views of the information provided by the server. So, while the server's information might include your account balance, if you want to find out the total payments in and out over the last week or year, then you will still have to calculate it yourself, based on the figures the server provides. Alternatively, you might access your bank server over the internet using another, 'more intelligent' client, such as a mobile app. This client might include a facility to extract figures from your bank statements and to perform whatever calculations you require. Such a client might also create bar or pie charts that display your income and expenditure across different categories that you define.

The web browser client in the example of the online bank simply displays the information that the server provides. The 'more intelligent' mobile app client allows you to take the information the server provides and to manipulate and display it in various ways according to your own personal needs. The web browser with little functionality of its own is often termed a **thin client** while the more intelligent client is usually termed a **thick (or 'thicker') client**.

We have seen that a two-tier approach seems to have some advantages, at least for applications that operate over networks. The client that is distributed to users may change, while the server part can be a centralised component that maintains dynamic, global data in a consistent and secure way for the organisation and for users to access and use. If a third party component, such as a web browser, provides the functionality required to support the application, then it can be adopted as part of the solution, with a significant saving of development effort. There is a potential disadvantage from splitting the application across a network in that data has to be transmitted over what may be a slow or unreliable connection.

There are other, slightly less obvious advantages in breaking the application into components in this way. We shall look at some of these in more detail shortly, but let's look at one advantage now. This is an advantage from which you will have benefited if you have ever decided to change your web browser. You might have changed from Internet Explorer to Mozilla Firefox, for example, just as a personal preference. When you did this you changed the client component of all the online applications you use. This is only possible because web browsers are based largely on common standards and because they are not realised as an intrinsic or built-in part of any of the applications you use; in other words, the client is loosely coupled to the server application.

1.2 Multi-tier architecture

The approach of splitting an application into tiers can be taken further. Both the client and the server parts can be further subdivided if this is appropriate for the application. The client, for example, may be responsible both for some processing of data received and for the presentation of information. In the example of a bookshop, we may need to calculate the total cost of an order and display sample content of books. These two functions might be separated into two tiers at the client end.

The server software might include one or more data stores (for instance, in the form of a database system). In the example of a bookshop, one data store might include the images, cost and reviews of a book; another might be used to store more dynamic information, such as the current stock levels or order time for each book. Splitting data up in this way would, for example, allow occasional backups of the more static data with more frequent backups of the dynamic data.

So, an architecture used over the internet might look in outline like that shown in Figure 2. Here we have added two more tiers on the server side: a data tier and another tier that handles interactions with the data tier, such as retrieving requested information or validating data that is put into the data stores. This tier, between the data tier and the server tier, is sometimes termed the 'middle tier' or **middleware**. An application that uses middleware to handle data requests between a user and a database is said to employ **multi-tier architecture**.

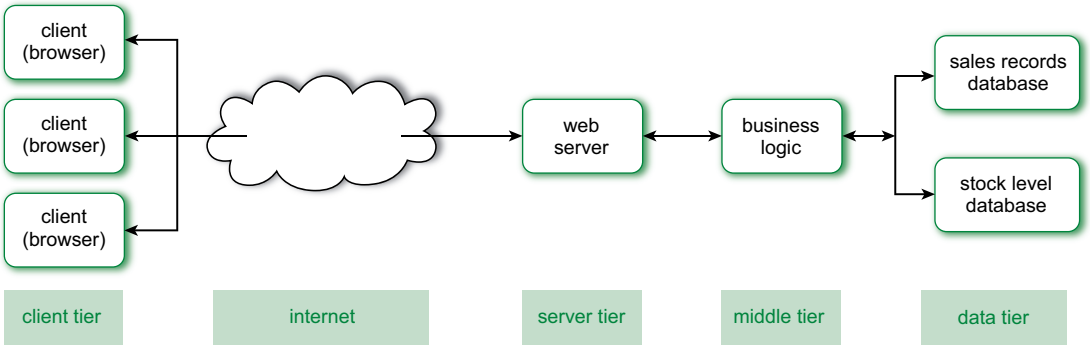

Figure 2 Example of multi-tier architecture

Quite commonly, 'multi-tier architecture' refers to what should more specifically be called three-tier architecture (client, server and data tiers). More tiers than this, however, can be used (as in Figure 2) and so the term 'N-tier architecture' is used generally to mean any architecture that has more than two tiers.

As in the two-tier approach, there are advantages to breaking down the application into multiple tiers. Each tier can be changed more easily as it is less dependent on the precise details of the other tiers with which it interacts. This again depends on how careful we are to adopt an approach that ensures the tiers or components are truly loosely coupled. Simply breaking the application into chunks doesn't guarantee this; we also need to adopt suitable standards, and specify precise and limited interactions between the tiers. N-tier architecture has almost become an all-pervasive approach, and is certainly very mainstream. Many more clients besides web browsers are now available that can realistically be interchanged without prohibitive effort, including databases and web servers.

2 Network and distributed architectures

As our description has moved from monolithic applications to client–server and then to N-tier, the application has been broken down into more and more parts. This trend has been extended in a modern approach called service-oriented architecture (SOA). SOA is based around the idea of breaking down an application into a set of much smaller tasks that can be performed by small independent, 'software components', each performing a discrete task commonly called a **service**. Service-oriented architecture (SOA) will be explained in the next section.

2.1 Service-oriented architecture (SOA)

SOA is the architectural solution for integrating diverse systems by providing an architectural style that promotes loose coupling and reuse. The software components provide services to other components via a communications protocol, typically over a network. The party offering the service is known as a **service provider** (a server), and the party invoking the service a **service consumer** (a client).

A service is some functionality, typically a business process, which is packaged as a reusable software component that is:

- well-defined – a software component with a clearly specified interface and outcome
- self-contained – the implementation of the service is complete and independent of any product, vendor or technology
- a black-box – the implementation of the service is hidden (encapsulated) from the service consumer.

Examples of services might be:

- Currency conversion – a service that converts a sum of money from one currency to another might be used in a wide range of circumstances. It might be used, for example, to convert all the prices on a website store to a customer's local currency.
- Customer credit checks – a service that provides the credit rating for a given customer.
- Provision of weather data – a service that provides selected weather data for a given geographical area and time period.
- Data storage – a service that allows data to be stored and later retrieved, perhaps with a range of capacities, costs and timescales on offer.

SOA principles

SOA is governed by a number of design principles (adapted from Erl, 2007):

> **Interoperability**: services implemented in different systems, locations or even business domains, and using diverse technologies – ranging from programming languages to platforms – need to work together to allow diverse consumers and providers to communicate. For this to happen effectively, services must rely on internationally agreed communication standards. Many of

the current standards were developed for communication over the web – although SOA is in fact technology-agnostic, so that its principles apply regardless of which communication standard is actually used.

Location transparency: consumers should be able to make use of a service without necessarily knowing where the service is located, that is, on which machine or even in which business organisation. Service consumer and provider are only loosely bound together, with the consumer having no knowledge of the service implementation or its platform, and consumer and provider communicating solely through messages.

Discoverability: the consumer must be able to find out about relevant services, which usually involves access to appropriate metadata about services. Discovery is often done through registries: a **service registry** will contain metadata and references to services. The actual services are often contained in inventories: a **service inventory** is collection of complementary services within a boundary of the provider, an enterprise or even a meaningful segment of an enterprise. We distinguish between run-time and design-time discovery. In **run-time** (or **dynamic**) **discovery**, the consumer first queries the registry for a service that matches the consumer's criteria; if such a service exists, the registry provides the consumer with the location of the service provider; the consumer can then bind dynamically with the service provider and invoke the service. This model of service collaboration is known as the '**find, bind and invoke**' cycle. While this model is of historical and theoretical significance, it has yet to become mainstream in practice despite much research and development effort in both academia and industry. This is due to the intrinsic difficulty of specifying semantic information which may allow one to reliably predict the interaction of a potentially unlimited number of diverse services. Instead, current discovery mechanisms support primarily **design-time discovery** – that is, they're aimed at developers who are trying to find out about relevant services in the planning stage of a SOA application development.

Loose coupling and encapsulation: coupling between service providers and consumers should be low, and confined to reliance to service public interfaces, with an understanding that such interfaces should disclose as little as possible of the underlying implementation details, which is known as encapsulation. The interfaces should be based on standard communication protocols, rather than proprietary ones, with open standards recommended to foster the highest degree of interoperability.

Abstraction: both the encapsulated implementation, the implementation technology and the physical location of services should be fully transparent to consumers, which should be reliant solely on public interfaces, service contract descriptions and SOA infrastructures to locate and invoke required services.

Autonomy: the more autonomous a service is, the more control it will have over its own implementation and run-time environment, and so the greater will be its flexibility and potential for evolution. Both implementation and run-time environment can be modified without affecting consumers.

Statelessness: excessive state information can compromise the availability of a service and limit its scalability; hence, services should remain stateless as far as is realistic to allow them to do their work.

Standardised interfaces and contracts: to support the service collaboration model and automate service interaction, services need to be described in a similar way using commonly understood standards. In particular, a service description may include descriptions of the service technical interface (that is, the operations which can be invoked and their parameters), the quality of service provided, and a service-level agreement which details both service characteristics (e.g. response time or availability) and cost to the consumer of invocation. Additional metadata may cover a range of information, for instance including user satisfaction rating or future development plans.

Reusability: services must be designed with reuse in mind, so generality of the service in terms of potential reuse across projects and systems is an important design principle. This impacts on both the granularity of the service (the service should strive to support generic business processes and tasks and close alignment with the business) and on the choice of standards and implementation technology, which should foster the highest possible degree of interoperability.

Dynamic reconfiguration: service-based systems should be configurable and reconfigurable dynamically by discovering and incorporating existing services, but while maintaining coherence and integrity. This should be supported by metadata, which underlie the discovery of existing services, coupled with contract agreement descriptors.

Activity 2 SOA

Allow 15 minutes to complete this activity

Given what you've just learnt about services and SOA design principles, which do you think are the advantages of developing business applications based on SOA?

Answer

Here's one possible list – you may have come up with something different.

- Business and technology integration, with close alignment between software services and business processes.
- Flexibility in responding to changes in customer requirements, to new business opportunities or to competitor threats, because services can easily be re-assembled or new services combined or incorporated into existing applications.
- Reuse, as a service can be packaged and made available for reuse in different parts of a business which require the same function.
- Integration of legacy applications, as legacy software can be wrapped as a service and made to interoperate with other applications.
- Potential development–cost savings, as systems can easily incorporate functions (such as credit card validation or online payment) provided as services by external suppliers.
- Interoperability and technology independence, as services written in different languages can interoperate using standard protocols.

Web services

When a service is made available over the internet, it is then usually termed a web service, which describes the service that a client can access from a server over the internet, utilising web protocols and standards to enable the exchange of data between them.

The main protocols and standards employed are XML (eXtensible Markup Language), SOAP (Simple Object Access Protocol), REST (Representational State Transfer) and JSON (JavaScript Object Notation) based.

Discussion of these protocols and standards is beyond the scope of this course, but you'll find articles on these topics in the Further reading section.

The main point to make here is that web services based on the SOA design principles described above will ensure that web applications written in various programming languages can run on various platforms, and can use web services to exchange their data across computer networks in a manner similar to inter-process communication on a single computer.

2.2 Cloud architecture

SOA is well placed to take advantage of a recent development known as 'cloud technology' based on what is often termed 'the cloud'. Cloud technology has changed how organisations use technology to provide computing services. However, there is a lack of clarity as to what the term will really come to mean. Some definitions restrict the cloud to mean that **virtual servers** are made available and used over the internet, but more generally the cloud is seen as consisting of a wide range of different resources.

The National Institute of Standards and Technology (NIST, 2013) provided a wider definition of cloud computing:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
>
> Cloud computing is the practice of delivering computing services – servers, storage, databases, networking, software, analytics and more – on-demand over the Internet. It is a means of providing computing services as a utility to consumers in the same way as other utilities such as gas and electricity. Companies offering these computing services typically charge for cloud computing services based on usage.

You might take the view that whereas SOA provides services on a network, the cloud extends the principle to other resources such as processing power and storage facilities and is not limited to the provision of services. There are, of course, quite a lot of details that are not covered here. How resources are charged for, made secure, 'cleaned' after use, etc. are just some of the aspects that need to be considered.

Activity 3 Benefits and potential problems of using cloud architecture

Allow 30 minutes to complete this activity

Take a look at the Amazon Web Services (AWS) website What is Cloud Computing?

What do you see as the benefits and potential problems of using cloud architecture?