ECM3408 Enterprise Computing Lecture Notes

Ogaday Willers Moore
Department of Computer Science
School of Engineering and Computer Science
University of Exeter

2015/04/09

Contents

1 Introduction

1.1 What is Enterprise Computing?

Enterprise computing is all about combining separate applications, services and processes into a unified system that is greater than the sum of its parts.

> "Enterprise Computing is seen as an integrated solution/platform for separate business problems, such as database management, analytics, reports, which have previously been treated as single business problems treated with specific software solutions."

- `http://www.techopedia.com/definition/27854/enterprise-computing`

1.2 Drivers and Motivations

Up-scaling An enterprise may need to be able to upscale to suit an increase in customers.

Outsourcing Specialist service providers will be able to fulfill requirements cheaper and easier than in-house solutions. Circumvents the need to invest in hardware, software and human resources, which requires *Capital Investment*.

1.3 Assumptions

This course makes two axiomatic assumotions:

1. **Enterprise Computing is based on web technologies.** This is not a controversial claim because of the many examples are out there. Netflix, Google, Twitter, Facebook are all worth billions of dollars and are based upon web technologies. Businesses will all need to provide their services and reach their customers over the web in one capacity or another.
2. **Enterprise Computing is done on the Ruby on Rails framework.** This is a controversial claim because there are many models available for web enterprises and Ruby on Rails provides just one instance of that. However this course will only be concerned with Enterprise Computing with Ruby on Rails.

2 Three-Tier Architecture

Web programs often follow a **Three-Tier Architecture**.

Data Services Tier This provides database access. Having this tier provides abstraction from the database behind the business. This means that migrating services, expanding onto more suitable hardware and scaling up is easier.

Business Services Tier This provides the "business logic", which is the functionality of the enterprise. For instance, Amazon provides for items to be added to carts, carts to be checked out, products to be searched and displayed and many many more small services which comprise their webshop and are all functions of the business. This might require communicating with the Data Services Tier and for the output to be sent to the Presentation Services Tier.

Presentation Services Tier This provides the user interface and means that the responses can be adjusted for each user. Specific users can be targeted differently with specialised content; presentation suitable for each client can be provided. For example, mobile devices could be served mobile websites and and html generated specifically for internet explorer (instead of, say, firefox) can be sent to the client. Alternatively, specific demographics can be targeted with specific presentational services, such as region specific language pages. Facebook, Google and Amazon all often provide experimental new interfaces to a subset of their users. This is made possible by the abstraction of the Presentation tier, which can be altered without affecting or interfering with the underlying business and data tiers. This sort of abstraction is leveraged in the **Model-View-Controller** which is covered in 4.

2.1 HTTP: The Hypertext Transfer Protocol

Communication is provided through the **Hypertext Transfer Protocol** (**HTTP**). **HTTP** is a call and response protocol. The client makes a call (request) and the server sends a response. The response is usually a web page or service. All **HTTP** communication is in form text (Strings), rather than ints, bools or other types.

2.1.1 HTTP Requests

HTTP Requests consist of a request line, request headers and a request body.

The Request Line The request line consists of a method (or verb), a resource and a version. eg:

METHOD	RESOURCE	VERSION
GET	/images/logo.png	HTTP/1.1

The request methods are

GET	For receiving data.
POST	For sending data / amending data.
PUT	For amending data.
DELETE	For deleting data.

Note **GET** requests print the data in the URL, which is really bad for anything except debugging. Therefore bank details which are sent with a **GET** request, for instance, will leave those details in the browser in plain text, so either the request can be intercepted and read, or the browser can later be scraped because there will be a record of the url, along with the details, will be recorded in server log files.

Request Headers Headers consist of `Name: Value` pairs on separate lines. eg.

```
Content-Length: 255
```

Some of the most important headers are

Header	Meaning
Host	domain name of server
Cache-Control	caching allowed
Connection	connection preferred
Content-Length	length of body
Cookie	cookie data
Date	date sent
User-Agent	name of client

The Request Body The request body is empty except for the case of **POST** requests, which send the data in the body.

Note that the server can also deny your request! It is under no obligation to follow the request, so for instance `DELETE google.com HTTP/1.1` would likely not be followed by the google servers.

Furthermore, most browsers can only send **GET** and **POST** requests. They simulate functionality of **PUT** and **DELETE** using the other verbs.

2.1.2 HTTP Responses

HTTP responses consist of status line, response headers and a response body.

Status Line The status line consists of version, code and reason eg.

VERSION	CODE	REASON
`HTTP/1.1`	`200`	`OK`

The string "reason" is determined by the code. The code indicates different responses from the server, and the first digit classifies the response.

INITIAL DIGIT	REASON
2	Good
3	Relocation advice, resource has moved.
4	Bad ie. 404 resource not found.
5	Transient bad, server overloaded etc.

For instance

CODE	REASON
`200`	OK
`301`	Moved Permanently
`404`	Not Found
`500`	Internal Server Error

Response Header Response headers consist of `Name: Value` pairs again.

NAME	VALUE
Cache-control :	Caching allowed
Connection:	Connection preferred
Content Length:	(length of body)
Server	name of server
Set-Cookie	cookie data
Transfer-Encoding	content encoding used

A case where caching might not be preferred is a live score ticker, so it will continue to update and wont use the cached value of the score.

Note that is possible to get a snapshot of past browser activity by looking at the browser cache.

Response Body This is often in HTML, but can be anything else (javascript, xml, etc).

3 Introduction to Ruby

Ruby is duck typed, interpreted, high level, Object Oriented language. It was invented/designed by Yukihiro "Matz" Matsumoto in Japan around two decades ago and has risen to popularity because of its expressivity and Ruby on Rails, a very successful web framework for building web appications. Ruby is introduced by countless high quality tutorials which I will not try to match.

4 Model-View-Controller

4.1 Design Patterns

Design patterns come about when people find a solution to a challenge and encode them in a pattern. What makes you stand out is the ability to not only learn from your own mistakes, but from others' mistakes as well by using their design patterns.

In the **Three-Tier Architecture**, a common design pattern is the **Model-View-Controller** which is a pattern upon which the Business Tier can be built. In this course, the **Model-View-Controller** design pattern principles are the "rails" upon which we and the ruby code are sat.

4.2 The Model

The model is responsible for **database access**. Why use a model to access the db? Because then the pattern is independent of database architecture, or at least insulated from the databse and then the enterprise is more portable to different architectures and DB services.

4.3 The View

Responsible for **data presentation**. By separating the presentation from the data handling, it is easier to focus on creating the best presentation. It is possible to have visualisation experts to specifically work on the view. The view is very important because gives customers strong impressions of the enterprise, and can influence customer decision. For instance, Amazon and Netflix provide different views in some cases in order gauge the effect on customers. Facebook and Google can upgrade and modify their interfaces without risking the other aspects of their service.

4.4 The Controller

Responsible for **data processing** and business logic. This section sends emails, adds things to cart, does programmatic things.

4.5 The MVC Pattern

This pattern is also reproduced in other languages and frameworks, such as PHP, and most e-commerce platforms follow this design pattern.

See the lecture slides for an example of an implementation of the MVC design pattern in Ruby on Rails.

5 Service-Oriented Architecture

The element that separates enterprise web systems from simple web applications and fundamental wed development courses is scalability. Enterprises (hopefully) grow very rapidly and the **Three-Tier Architecture** can become bottlenecked or defficient for a rapdily scaling application.

Refactoring is the process of changing the structure or layout of software without changing the underlying functionality. The aim is that the functionality is delivered in a simpler, more easily maintainable or more effective way.

Service-Oriented Architecture is a refactoring of the **Three-Tier Architecture** so that it is better suited to enterprise web systems. In order to achieve this the architecture incorporates multiple servers and databases, decomposition of service, a load balancer and a data warehouse into the **Three-Tier Architecture**.

5.1 Websites and Web services

Websites are what you expect: human users can navigate to them in their browsers and interact with the services. Web services, on the other hand, provide services for other software via a formal, well defined and constrained interface. Often enterprises will have services which require communication with other services. This is known as Enterprise-to-Enterprise communication. The software that requests the data is known as a service requester while the software that provides it is known as the service provider.

Websites deliver their content to the client in the form of html or other mark up formats, which is then rendered in the browser. Web services traditionally are queried using a specific SOAP (Simple Object Access Protocol) which is usually transmitted by **HTTP**. **XML** can be used to serialize communicated objects, and **XML** will be described in 11. **APIs** are now moving towards a more RESTful architecture which will be described in 8, which do not rely on SOAP.

5.2 Features of SOA

A significant feature of **Service-Oriented Architecture** is the decomposition of services in conjuction with multiple servers/databases and a load balancer. This provides:

Understandability The decomposition of services allows for division along organisational lines. The codebase for each service can be developed by teams which can take ownership and specialise in providing that service. Other services can be introduced or outsourced easily as well.

Isolation Service code and data is again kept independent of other services making development and maintenance for that service pertain only to that service. One service being compromised will not compromise other services.

Uniform Access Services can only be accessed via published, supported Application Program Interfaces. This uniformity means that services work how the customer expects because it follows a well known standard. As a side note, this provides a contract

between the user and the service provider. If the API is changed, this contract is broken because the user has come to expect specific behaviour, so APIs need to be well thought out and prepared for legacy support.

Scalability As the number of customers increase, the enterprise must be prepared to increase capacity. Services can be distributed across many (potentially virtual) servers.

Redundency and Failover Business need to protect against fault or attacks which bring down their services. If a server fails they will be unable to serve their customers, which is a cardianl sin in the world of business. As a business scales and gets more custom, redudancy needs to be provided. **SOA** provides many servers for the same service and a load balancer. That way, if a server fails, the service requester or customer can be routed to a different server by the load balancer.

In the **Three-Tier Architecture**, the Presentation and the Business tiers wont do much hard work - All the heavy lifting is done by the Data services tier. Thats why a load balancer is introduced, to make sure the work is evenly distributed across the multiple databases.

5.3 Example: Google

Google offers many, many services which can be used independently of each other. Google offer add space, search engine, social network, maps, email, calendar, video service, cloud storage, cloud computing, version control cloud based office (Google Drive), currency converter, calculator, translator and many more services. They will have servers on each continent to serve different clients and they have to deal with many different clients who may lose connection during their interactions. Google will be able to deal with servers failing and continue to offer all of their services. They will have different teams working on the search engine and the gmail service. Presumably, also, if the servers supporting the search engine functionality all fail, it will not also affect their ability to serve clients who want to use the gmail service.

5.4 Exceptions

Sometimes services cannot be abstracted and decomposed fully from each other, and therefore if there is a failure in one service, the other services will also be affected. The best example of this is the log in service, which mnay enterprises offer. If the log in and authentification service fails then while the search engine, videos and other services may be available, many other of Google's services might not be, such as the email service, the calendar service and others. This is because it it is too integrated into the usage. The log in service will probably still have its own team, however, who can take ownership of it, but other services will need to be able to interact with it, which is why uniformity of access must be maintained.

6 Ruby on Rails

Ruby by itself is nothing special, does everything that python, php and other high level, dynamically typed, OO languages do. This course is going to look at a very useful framework for Ruby: **Ruby on Rails**. It attempts to do a lot of the low level stuff when you when you set up a web application. Two advantages:

1. It does a lot of work for you. It creates a lot essential files and directories that you would make every time, prepares and positions resources in a standard way.
2. You don't forget things and make mistakes because you are notified of everything of you need to do.

Rails provides an outline implementation of the MVC in Ruby. You are compelled to use the MVC architecture when using Rails. Raisl Ddesnt create the whole platform for you, for instance you cant get a replica of facebook. However, this leads to far fewer mistakes being made and furthermore the application will be much more maintainable, because the common frameworks and convention over configuration means new developers can approach the project and see how it works because things are done in a standard way.

6.1 Using Rails

The lecture slides cover a brief tutorial and there are many tutorials online as well. The scope is much larger than that of the lectures and is much more extensive than can be afforded in these notes, so Ruby on Rails itself will not be covered in these notes.

6.2 Object-Relational Mapping

Ruby on Rails employs a design pattern known as the Object-Relational Mapping. This pattern (which also exists in other laguages) dictates that there is direct map between lines in a database (records) and objects which live in ruby code. Therefore objects can be manipulated in Ruby code and the modified record can be saved in the database. This is a very useful pattern and allows for greater abstraction in the model, which as prior mentioned, is very useful.

6.3 Routing

Ruby on Rails uses something known as "Routing", which allows urls to maipulate the control of flow in order to run ruby code. The Ruby code then controls access to the Model and generates the Views.

The Views can contain forms and hyperlinks so that CRUD operations can be sent via HTTP. Might not need forms because the code might be accessed via other code / machines using urls like an API. update and delete/destroy are not described in the view because they cannot be issued by the browser.

7 Remote Procedure Calls

7.1 Parallelism

As an enterprise scale, more CPUs will be required, because the limits of what a serial processor can do will be reached. Furthermore, the limits of serial processors are being reached in that they are no longer becoming faster and more powerful as dictated by Moore's Law, so parallelism is required to match the increased demand for computing power. The problems with this are

1. How to distribute work between multiple CPUs? Do you divide different tasks between different CPUs, or do you divide data between CPUs and have them each do the same task to the different data?
2. How to load balance the CPUs. If one processor is idling, that is costing money for gain. It is wasted power, so the problem of dynamic task management and load balance is necessary.
3. Deadlock and Livelock are common problems as parallel processors block while sending or receiving messages or try to access the same resource simultaneously.

Distributed or parallel computing is the field of study of this problem. One paradigm is that of **Remote Procedure Calls**. A regular procedure call is simply a local method invocation with different args. ie. $f(a, b, c)$. The remote aspect comes from one computer invoking a method call on another processor across a network. There are more problems specific to this form or parallelism.

1. How to address the function? Need to name the machine on which the method is being invoked.
2. How to serialise the arguments that are being passed to the remote method invocation? Ints might be easy enough to serialise but arrays with pointers might be more complicated.
3. Also, return of the results suffers the same problems.
4. It might not even make sense to use RPC! The latency and overhead involved in message passing is many orders of magnitude higher than access to local or cache memory, so it is much more efficient to do a simple computation locally, even if the computation is duplicated across many processors.

7.2 Traditional RPC

For RPC to occur, we must have a client program and a server program sitting on a conected network. The client must have a stub, which lists the procedures and functionality that the server provides and which therefore can be called remotely by the client. The server itself must of course have an implementation of all the procedures listed in the stub! Both the client and the server require a network layer for packing and unpacking procedure arguments and objects. The client sees the stub as a list of regular methods. However, on invocation, the methods in the stub **marshals** the parameters for the procedure and packs them into a method. The server unpacks the methods, unmarshals the parameters, runs the procedure, then marshals the response, sends it over the network, which is then unpacked, unmarshalled by the client.

7.3 Drawbacks of RPC

1. Might be inconcsistencies with convetions and formats of arguments (how big is a long integer?)
2. Every client must have an implementation of the stub in order to speak to the server. This can be a large problem because it requires the client to be up to date with the required level of software which might be more than the average user can manage. Also, different servers might require you to have different stubs, so there is a lack of universality.

3. Also slow because there is a lot of packing and unpacking involved.

One solution to the second problem is to use a standard interface for RPC. There are two well known ones: the **Common Object Request Broker Architecture** (**CORBA**) and in Java there is **Remote Method Invocation** (**RMI**).

7.4 HTTP RPC

However, were not going to use either. A much more widely available one is **HTTP**, consisting of the request line headers, bodies etc. as seen earlier in 2.1. Even the above two standards are not ubiquitous and require the client, who cant be considered universally capable enough to download and install/update software such as CORBA. HTTP however, can be guaranteed to work with pretty much any interface. Furthermore, it is much better at being accepted by most firewalls, so there is an added advantage there.

We can specify the procedure to be called on the server in the resource section of the **GET** request. Can specify arguments as query string values when using **GET**. Follows ? character, as in google, youtube etc. The problem with this is it is possible to leave out arguments or provide invalid argument types because **HTTP** is weakly typed. There is no strong checking of the imminent procedure call so errors might only be found at run time and fail on the server. The main advantage is that the only dependency for HTTP RPC is a modern browser which is **HTTP** capable, which is all of them.

Might want to look at the lecture slides and copy and comment on some diagrams

Anatomy of the request strings. operations and argument information can be specified using query strings in the request line. Connection information can be described in the headers. Semantic difference between GET and POST requests: Arguments are contained in the url string for GET requests which mean a) the args are visible, which is a bad idea for secure information, but also b) serialised objects might be large and there might be loads of arguments which can create long, intricate and ugly urls (which used to be a problem: your browser could be crashed by long links because they couldn't deal with too many characters.)

result code. 404 errors occur because of the weak typing of **HTTP** requests / RPCs. The APIs are not published in a machine readable way. Recall 400s are permanent and 500s are transient.

Of note: HTTP does not require the installation of custom RPC client software such as CORBA. Client users might not have either the permission or the technical knowledge to be able to install such software, so the ease of HTTP RPC for the client is definitely advantageous.

7.5 Close-Coupling

If you invent your own API you get a close coupling of server and client. Therefore, if the API changes, the contract is broken and the client programs will no longer work because their **HTTP** requests will fail.

Client sides will have to update their programs to reflect the changed server protocols. This is expensive, and can cause downtime, so customers and traffic might be lost. As stated before, changing APIs is best avoided.

8 Representational State Transfer

In the previous section we looked at **RPC**. Arguments and return values are marshalled (packed up and serialised) and unmarshalled (unpacked unserialised) in the stub and in the skeleton. Problem is there isnt a universal Web Service Definition Language (WSDL), which is a way of describing which services are provided in a large xml file which can be downloaded and used by a developer to build client software to remotely call procedures to be executed on the server. This is not ideal. Instead, a better alternative is for the server to organise its services via the RESTful convetion.

We would like our web services to be used by other programs as well as by humans, such as in Enterprise to Enterprise communciation. Therefore, our services should be automatically accessible, and RESTfulness allows for this.

8.1 RESTful Features

Ron Fielding tried to study and extract from the web the features that made it so successful. He decided that the following elements made for successful web services.

Client server constraint Server deals with services and data, and client interact with the UI, which means software dosen't need to be downloaded to provide the service. (Apps revisit this model though, which are successful because there is a direct link/contract with the app provider and downloader). Clients browse services provided on the server.

Statelessness The server does not store information about the state of its users. Each time a client accesses a server it is as if for the first time. Statelessness requires the use of cookies to provide state on the client side instead. The key advantage is that if the server becomes overwhelmed or fails, an alternative server, for instance in a load balanced system, can provide the same service to the customer. Without cookies, this element would be fatal to the internet.

Caching Headers dictate whether data can be cached or not. Means some repeated requests are not necessary. The cache obviates long routes to servers for data which has already been requested. Examples of things that should not be cached are things like live results. A side note is that caching trades space for time! If something is cached it takes up space, but requires less time to access.

Uniform Interface Similar to databases, resources are managed using only CRUD. Interesting because delete cannot be uttered by any web browser.

Layered System The internal layers such as databases cannot be seen by the client. One url is used to access a service, regardless of the architecture beneath it. Gets rid of close coupling: the client doesn't need to know the details of the architecture beneath, so the same service can be provided in the same fashion, even if the implementation is different.

Code-on-Demand The original web proposal envisaged that there would have to have **extensibility**. The server must be able to provide code to clients for them to execute. In practice it is extremely difficult to pull this off. The current system is javascript. The big difficulty is arbitrary code execution is dangerous, so the code on demand must run in a sandbox which is very hard to get right. Previous efforts include java applets, which had various problems: requiring permissions to runs, taking time to

load up the JVM etc. There was a phase when javascript was turned off by everyone because of arbritary code execution, now websites won't work without it.

The web is all about manipulating data on a server. Normally the only manipulation is to view data, like GET. Othertimes it is to upload, which is done with PUT.

8.2 Further Points

In the lectures an example of a RESTful Airport code lookup in Ruby on Rails was covered. Some key points from that are

Template Deisgn Pattern This is the ability to embed code in HTML. This is useful for dynamic content generating and for mixing the ORM into your views.

CRUD Operations

POST	corresponds to create.
GET	corresponds to show.
PUT	corresponds to update.
DELETE	corresponds to destroy.

Good Ideas Behind The Web REST constraints provide the foundations for the incredibly successful web.

9 Resource-Oriented Architecture

Why do we want scalability in enterprise computing? Because we want our enterprise to be able to grow fluidly in order to match demand. The internet, so far, does not appear to have a near limit on how far it can scale, so our enterprise needs to be able to (potentially) match that. We dont want to reach a point of growth and find the architecture needs to be redesigned because that is costly and stops the service from reaching the customer. Not only that, modern enterprises require a high speed of scaling. Web enterprises go from being inexistent to being worth billions of dollars, which you won't find in traditional business (the converse also happens of course).

Resource Oriented Architecture develops software as in terms of textbfresources and allows the client to interact with **representations** of those **resources** via a RESTful interface.

9.1 Separation of Resource From Representation

As noted above, the server software is built in terms of resources, which can be records from a database, code functionality or data strcutures. The client shouldn't be given direct access to this resource, of course. Instead, they can interact with a representation of it. In this way, it can also be customised according to the client via headers, if needs be. A spanish client might want the webpage in spanish, for instance. Google maps provide satelite images which are resources, but depending upon the required resolution, the client will be served different scale representations. Different devices or browsers may want different representations of the same resource. There may be many different representations of the same resource.

As noted, this specification is done via headers. Headers can contain geographical data, screen size data, browser information, cookie information, etc.

9.2 Self-Descriptive Representations

By self-descriptive representations, we mean a representation that contains data as well as the metadata that describes its syntax and semantics. Essentially, when you receive a representation of a resource you re also told how to interpret it. Sometimes there is not a standard. Then you often get an XML file with a document type definition (DTD), which will be covered in 11

Why dont we ship pictures instead of xml? Its the same information. Because the picture is much more difficult to work with than the xml / raw code / information. So much structure to wade through in pdfs / images. So representations are much easier to deal with by the client side machine. It is better to send fundamental representations than convoluted ones.

9.3 Manipulation Of Resources By Representation

Representations can be changed. Google maps encouraged people to add pins to it to mark out locations and shops. People like direct manipulation. The above would not have worked if the pins had to be entered via latitude and longitude.

IT's also much easier to change the resource via the representation: rather than having to manipulate machine code to change a variable, it is possible to change the representation and it will change the resource!

9.4 Hypermedia As The Engine Of Application State

HATEOAS is the big idea! Content served to the client is embedded with hypertext. There are two ways of telling a server what to do next: Either click a hypertext url (link) or submit a form with an html submit button. The advantage of this is that the client keeps track of their own state! Imagine if google had to keep track of the state of each of their users: the memory cost alone would be massive, never mind the book keeping. Furthermore, the client can reconnect to another server as determined by a load balancer. The server could crash, and another server on the other side of the world could take over and it wouldnt make a difference, because the state is stored by the client, in the browser history and url etc.

By providing the options or hyperlinks and submit buttons, the server therefore provides the clients with options. The server then does not need to track client state or where the client is at any time, and it can serve out pages on demand to the client with no memory or book keeping costs.

For example, the server provides a choice of coke or fanta to a client. If the client asks for fanta, the server doesnt remember if you looked at the menu or not. it doesn't know if you have previously ordered fanta or coke before. All it knows is that it has recieved a request for fanta.

The client can store urls in the history, but that is responsibility of the client, not the server, as are cookies.

In a world with the state stored server side, even if the server could manage the memory constraints and book keeping of storing state, its not particularly failure / crash safe at all. If the server goes down, so does the state.

This is what was covered by RESTful archicture and is key to scalability, in a huge way.

9.5 Example: Representation of a Coke Vendor

The classic example is a coke machine, which is a real life, physical thing. Some students at MIT "put the machine on the internet". The idea being that they could see the state of the machine on the internet without having to physically visit it. At the time, long before the internet of things, it became momentarily famous, because the status of the machine could be seen worldwide. The machine could have shown the temperature, the price, the stock of the machine. The physical thing is the resource, and the status on the internet is the abstraction or representation of it, which could be interacted with (ie. viewed) on the internet.

10 Storage as a service

To recap, if we build with the contents of the above lectures in mind, then we can be ensured to dodge a few pitfalls ahead in the road of a expanding enterprise.

Microsofts SkyDrive, cloud driven storage service was taken to court by Sky television, so its a relic nowadays.

Coffee bar is an analogy for the MVC. Barrista takes your order and has various hardware to accomplish the order. They may need to access their storage for some ingredients. Just before it is served they ask for special needs such as sprinkles.

Why use MVC? Because its been proven to work, but also because its is compulsory with Ruby on Rails.

Use cases correspond to **http** verbs, ie. GET, POST, PUT, DELETE, used for RESTful web services.

Object Relational Map hides db statements behind objects. So you call ruby code which automatically calls the sql commands.

Template design pattern allows the mixing of html and ruby code, only differentiable by the tags.

Have the feedback such as Created! so the user knows if something has gone wrong.

Curl interacts with **http** requests? look into.

new is for human beings, not for robots.

11 Extensible Mark Up Language

What is mark up?

Machines can read XML because it is a formal language it can easily process with tags containing different aspects which are easy to navigate. Included is a handy description of how to make sense of it.

12 JSON

Why even consider JSON? XML is Verbose. JSON, comparably is sparse. Even bytes saved can make a big difference if you have many customers/clients. The limit is on the number of bytes you want to send, not on the customers. Want to be able to deal with potentially infinite customers. Originally invented/invented to be used for the asynchronous web.

Synchronous web: Two ways of stirring the server: URLs/links/hypertext and form submit buttons. Problem with synchronous web: latency or delay. Lots of idle time during which computation time could be done.

Asynchronous web client makes requests when events occur, ie. telling you if a username is taken. This is all done in javascript. Or when google suggests search terms or lists results before you have finished typing.

Javascript runs in the browser, server program runs too. The requests and responses can be sent in any format, but we want to send it in an easily parsed format. Eg the JSON format.

AJAX: Asynchronous Javascript and XML machinery.

Why is JSON easier to parse than XML? Because of the tags: many more characters for the same functionality/meaning. eg. open array is [vs ¡catalogue¿, etc. Therefore JSON is quicker to read and write.

AJAX controls both ends of the server/client communication, so you will not receive any unexpected requests.

JSON schemas describe the structure of a document with declarative rules. A message which receives a DTD is self describing. Not used in AJAX, because unnecessary. Schemas are an afterthought that attempt to replicate one of the successful features of XML, the DTD.

In industries there can be decided upon formats. Very powerful because it is a common language, which means business to business transactions are easier.

13 Database Integrity

Integrity of data is knowing that the database contains what you think it contains, ie. correct and uncorrupted data.

What is the greatest strength of an enterprise? Not its people, its the data., and one must ensure its accuracy or integrity. Dunhumby is the company which does loyalty/memberships for Tesco. If their database integrity is compromised, then they will not be able to provide the service they promise to their customers.

If the enterprise is divided into many services, each of which requires DB access, there is a higher chance that the data integrity can be compromised.

Integrity rules Entity Integrity: No attribute of a Primary key must be null. When anything is added to a databse, you need to check the primary key. A Primary Key can be comprised of several attributes. If a null PK is added the db will be corrupted and need to be thrown out/recovered: expensive. Referential Integrity: Foreign keys must refer to things which actually exist. Delete is the problem here. Need to delete record from all tables which call that foreign key. Especially confusing for large and enterprises with lots of services.

If a foreign key exists in a relation, then the foreign key must matchsee slides General constraints: Many here. About what data values are allowed. ie. Cant apply for skydiving if too young, cant apply for insurance if too many restrictions, no deliveries outside of the UK, etc

Where should validation be done? All three places! In the model in the view and the controller. Advantage of validating in the view: extremely good error messages, faster feedback. compare to errors thrown by the model ie. errorKeyVal etc. Each component should check. Also, then there is redundancy of data integrity checking. model programmers might not trust ux designers etc.

rails auto generates a bunch of validation methods. When something is wrong, it tells you everything wrong with it.

Rails doesnt: maintain data in the form which you got right (which is good convention).

14 Database Scalability

By now we are familiar with the **Three-Tier Architecture**: Presentation - Business Services - Data services.

As the enterprise scales up, we will need more more servers to provide the business tier. The internet is suited to this because the RESTful (Representational state transfer) architecture means that the server doesnt store any state, its all held by the client in the form of cookies etc. This is good because therefore load balancing is easy, redundancy of server is easy because a user isnt tied to using any server, they can go to any other server and be served in the same way.

Unfortunately, the bottleneck is then at the DB. Many servers querying the DB will lead to it being overloaded.

Databases are represented as stacks of cylinders because they are traditionally stored on discs.

Disks: spinning slow! Typically 100 000 times longer to access something from a disk than from solid state.

How to solve the DB bottleneck? Replication? We add more more servers to the business tier, why not add more DBs? Plus is that databases are now running in parallel, so any one can be read from, so reading from them is much quicker. Minus is that each database needs to be written to when new data is added to the DBs, stopping the whole system until it is dealt with, slowing it all down. Solution might be that we might not need absolute accuracy. This is something called eventual consistency: do the stack of writes at a convenient later time.

Do we allow some slackness before a change is made to the system? On something like fb and twitter, people might not like it. OTher times, there is a noticable delay (a few minutes) but you get a confirmation message. Problem might be that a request is resubmitted by a user who doesnt receive confirmation that their action has gone through. Partition by table. Faster reads because the data + use cases are partitioned. Fast writes for the same reason. Problem is when there are dependencies, like closing an account, which all databases look at. Eg. Facebook could partition its services onto different servers. This is slow when there are dependencies. Ideally, services which dont require the same data. This is unlikely though, Companies want to be seen as one entity, ie their services should be linked. Partition by service. BT can offer phone lines, internet, TV, separately. Faster

reads, faster writes again - similar to partition by table. Can be frustrating if you have to keep giving your details to the same company over and over again. If you change your address for one service, it needs to be changed for the others.

Another advantage can be that you can divide your company into specialists: One team can focus solely and be experts on one service, while another team can be experts on another service.

Example: Facebook could partition its databases by service across servers (walls, profiles, authorisation, abuse, logging, ...)

DB management systems:

As far as we are concerned are likely to be row orientated management systems. Could also be column oriented management systems.

Row oriented is good for one record operations: because the data for one record is contiguous, so much quicker to access on DB disks!

Column oriented is better for doing column operations: eg. getting all columns requires retrieving all records, throwing most of the data away and keeping the one feature from each record. Much quicker if you want to sum across all records, for instance. Questions like how much did we sell last month, What was our lowest stock last month etc.

Sharding: A database management system may shard a table by storing different rows on different servers. Eg. might shard records by EU/NA. Know the query should be one or the other by ISP, GPS, location services etc if you have geographical knowledge. Eg. going to amazon.com rather than amazon uk! Your basket doesnt carry across because the services is separate. Eg. sharding by van colour would be really bad because 90% of vans are white, so one db would be overloaded.

Not only SQL = NoSQL DB management systems may replace a table with a simple key/value store. Advantage is that the data doesnt need to be structured. Can be stored Google does this, but they can search unstructured data very well. More for a DB course.

15 Data Warehousing

Online transactional transaction processing

Beyond storing your data, you need analysis of the data you have in order to find something useful from it. Eg. its not a good idea to open a new hotel in Barcelona because we arent getting enough bookings, or conversely it is a good idea to because bookings are up etc. OLAP: Online Analytical Processing.

Where do we get the data from? Can buy it from other companies. Can use footfall information.

Makes a data warehouse.

What to do with that data? Change your business for the better. Record failed searches so you can find out what your customers are looking for but you dont stock/provide etc. Find most successful products, offer products to customers who are likely to like them.

Data Extraction: Identification and collation of the data from different sources.

Data transformation: integration, cleaning, rearrangement and consolidation. Can be difficult to do data transformation: Dr David Wakeling, Mr Wakeling, his sister is also a dr,

etc.

Data Loading

Key point is that data ages over time. Very old data might be necessary. Not often stored on disks, doesnt need to be accessed frequently.

Data storage pyramid: top point is recent figures, might need to be accessed quickly, frequently. Towards the bottom of the pyramid, the wide base, the data is older, how has the shope been doing in the last decade etc. Data archives at the bottom might be stored on cd or tape.

ie. top is interactive data. Almost live data. Close to the concept of a data dashboard. Interactive sector stores unsettled transactional data, almost immediately. Information can still be changed: (Almost transactional db) a flight might be cancelled, credit may be refused, which would be noted.

Integrated sector. Might be end of the day or week. Integrated with other data settled there. Can be scrutinised.

Near Line data is older, has been dealt with, no longer needs to be changed because its been in the previous tiers already so the chances of accessing it are less, so it goes onto slower technology.

Archival sector. Data can nowadays be stored forever. The probability of being accessed though is very low. The orgs forget nothing. Banks need to store mortgages for years as well.

Retrieving data from the transactional database isnt a zero cost operation. In fact, if a website is performing badly, asking questions can make the website perform worse, driving customers worse and causing a vicious cycle. The Transactional DB is meant for customers, not for the data warehouse. Shouldnt mine your data at the expense of the user.

Need to answer the question: How are we doing right now? Last week? This year? Past 5 years? Historically. Other businesses will do this, so you need to. They will know what their customers want and they will provide it. Thats why you need a data warehouse.

16 Cloud A rchitecture

Analogy between providing electricity and the cloud.

Advantage of this.

Rapid Elasticity. If you buy another 400 pcs, where will you put them? Enterprises need to scale very fast.

Resource Pooling: more for cloud providers. Can get customers to share machines. Can use vitual machines rather than physical. This is known as multitenancy. If a provider buys some machines, you want them to be useful work at all times. Therefore its useful to be able to load balance the virtual machines on the physical machines so they are always working. The VMs are checkpointed so if a physical machine is broken, the VM can be picked up by another physical machine without dropping service. Also, costs are low.

Measured Service.

Cloud Adoption:

Last time: Different levels of architecture: Bare machine, operating system, software as a service.

Above that, the infrastructure is either: Public or private cloud.

Public Cloud vs. Private Cloud

Triangle of CIA: Confidentiality(Only those with access rights can see the data), Integrity (IF you store a 6, you get a 6 back), Availability(When you want your data it is there)

Public clouds might offer better data assurance because:

They are hardened by continual hacking. Of all users, there are a small %age of scumbags who will attack large organisations for the notoriety. Also rival organisations and governments fund attacks. If they would have fallen prey, they would have already fallen victim to it by now and be out of business. And if they havent been fixed, they will be soon. Staffed by top people. Only large data centres can attract the best people, because they will want to work for the best/biggest/most prestigious companies. So industry leaders work for the largest companies. The best data centres can be built for these largest of companies. The largest companies do not buy standard hardware and they design their own computers and servers and buy them in bulk from China. Therefore they offer more assurance because they are built with the best security kit. Their stuff is often recent because they are also expanding and have the capital to create new data centres as they need it.

Private clouds offer less assurance because of they cannot offer the above.

Sporadic penetration testing: Companies cannot test themselves against penetration easily: you tend not to see your flaws, and you also know the intricacies of your architecture so you are inherently biased. Also, security companies arent often invited back to the same place because the company supposedly fixed the faults that the tester found. These tests are expensive, and easy to get lax on. Liable to perimeter complacency. May have been state of the art, but then becomes outmoded. Also, the people who put the system in place wont be able to foresee problems which they havent already thought of. The risk will never have occurred to them. staffed by good, but not the best people. This may leave them open to more sophisticated attacks. So the private cloud may suffer from fewer attacks, but the staff wont be the very best. The customers and board members of a company might not want to hear that they are being served by decent but not the best staff. They dont have the buying power of the public cloud to buy the newest kit as soon as it is on the market.

Service level agreement Legal contract between you and the cloud provider.

Five nines up time is 99.999% uptime. Never promised by cloud providers, thats safety level insurance.

RESTful cloud

Data movement is still slow and expensive. But the cloud offers the ability to quickly

Edward Snowden:

At one point people thought that the big data providers wouldnt give the data away to the government. Now the cloud providers have had a rethink, Google encrypts data everywhere it sends it.