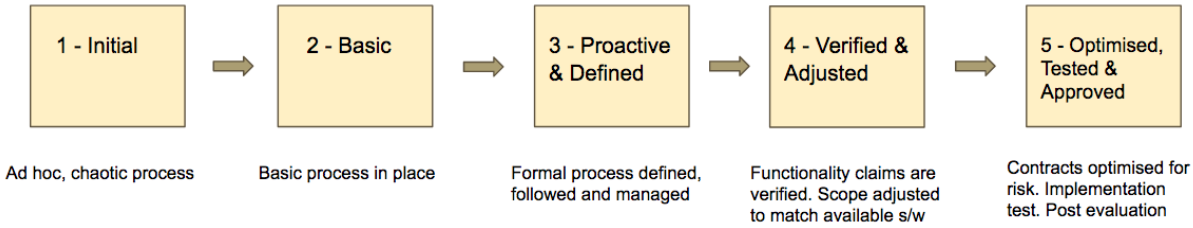
Lab 4

Q: Describe the Software Selection Maturity Scale? What is its relationship to technology adoption in the Enterprise?

A: The Selection Maturity Scale is a five-level scale which measures the maturity of a given enterprise for its technology evaluation and acquisition process. This comes into use as Not all enterprises are created equally in terms of their ability to make data-driven decisions regarding technology strategy, investment and adoption.



Q: Compare and contrast the monolithic and SOA models of enterprise application software. Describe are the benefits of a SOA composition approach to application construction.

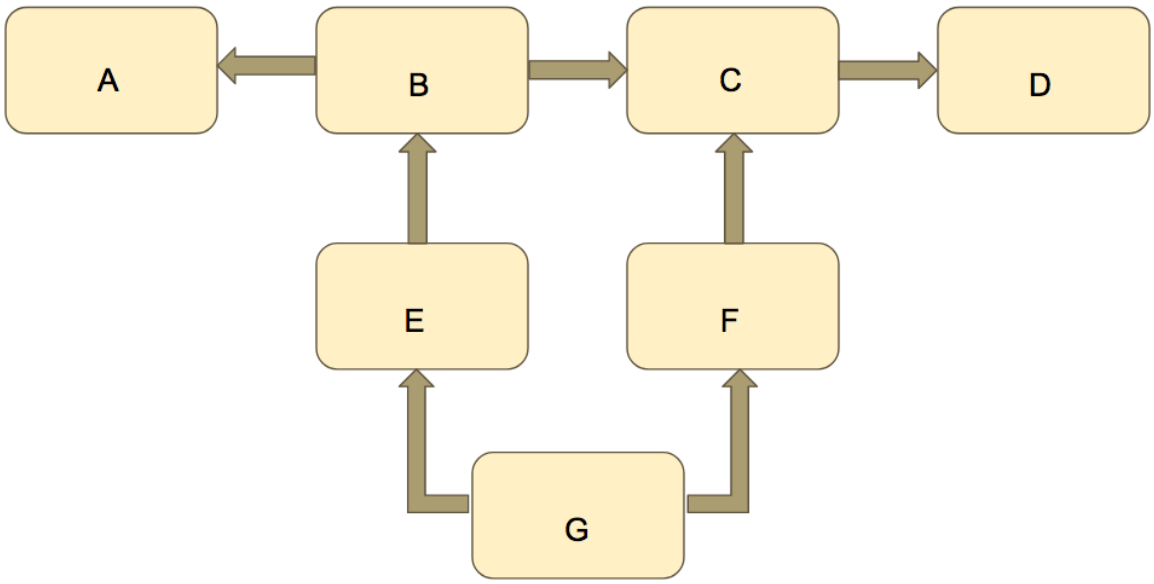
A: In relation to monolithic software data processing applications tended to be monolithic in nature, i.e. a small number of large applications providing many diverse functions on a wide variety of data sources. In relation to SOA a *service-oriented architecture* is a style of software design which sees the construction of software solutions from a set of technology-independent components which can be composed together over a network using some well-defined network protocol. This contrasts with a monolithic application in the sense that solutions to problems not originally address by the monolith can be created by end-users more flexibly from the components.

The components that make up SOAs are known are services, services are the basic unit of development in a SOA environment. Services have four basic properties in the SOA design approach.

1. Logically represents some business activity with a defined outcome
2. It completely self-contained
3. Is considered to be a black-box from a consumer's perspective
4. May itself be composed from one or more other services

A service abstracts the details of the business logic it implements by presenting some canonical view to the consumer. The consumer expects that service to be able to fulfil its specialised task

without the need for external dependencies. Service composition is a natural graph in nature with loose coupling. (shown below)



Q: Why have enterprises moved towards web technologies for service software construction? What are the principal benefits?

A: In recent years, enterprises have adopted much of the web technology stack for services implementation. In particular, HTTP is the favoured application-layer protocol and XML and JSON are the favoured serialisation formats. As a set of industry standards with wide community support, the web stack offers cost-saving solutions to common service construction concerns However, services using web technologies need not necessarily be consumed on the web or by web clients in the traditional sense. Here, the web is just a reusable toolkit of common building blocks. Because of their increasing reliance on open standards and open source software components within their own businesses, enterprises are tending to contribute to the development and support of these in the public fora. It is not uncommon for the larger technology companies to provide resources or financial assistance to standards bodies and software projects in the hope of both maintaining their ongoing viability and influencing their strategic and technical direction.

Q: What are the advantages and disadvantages of HTTP statelessness as the basis of a service application protocol tier?

A: The application layer protocol of choice is the **H**yper **T**ext **T**ransport **P**rotocol (HTTP), part of the Web suite of technologies. These can be combined with VPNs (layer 2 or 3) or TLS (layer 4) for encryption and authentication. One of the main advantages would probably be, that a request can be easily identified and each one is independent. Also HTTP protocol creates a new connection on each request. In contrast stateless apps do not keep information about a user. Cookies are used and these are less efficient than in-memory.

Q: Describe the architectural constraints of the REST architectural pattern

A: REST is architectural model and design for serve network applications. Constraints can be broken down to.

- **Uniform interface** – the interface between clients and servers. Decouples the architecture and enables each part to evolve. This constraint states that all services and service consumers must share a single, overall technical interface. This is the primary constraint that really distinguishes REST from other architectures. In REST, this is applied using the verbs (commands) and media types of HTTP. The contract is usually free from specific business logic context so that it can be consumed by as wide a variety of client types as possible.
- **Stateless** – A client can send multiple requests to the server (request must contain all info necessary so the server can understand it. The principle that no server-side state exists between any two REST requests from a consumer, i.e. requests are self-contained and standalone. Contrast this with a transactional style.
- **Cacheable** – Multiple clients accessing same resource and cause responses to be cached avoiding unnecessary processing. Service responses can be explicitly labelled as cacheable or non-cacheable. This allows intermediating caching nodes to reuse previous responses to requests without having to go all the way back to the service. This is a key idea in making RESTful services scalable.
- **Client Server** – Separates clients and servers so they are not concerned with data storage. The provider offers one or more capabilities and listens for requests for those capabilities. The consumer is responsible for presenting the responses to the user and taking any corrective actions on foot of errors.
- **Layered System** – Client cannot tell if it is connected to the end of the server or along the way. An arbitrary number of nodes can be placed between the ultimate service and service consumer. Their existence must be fully transparent so that they can be added and removed at will. This allows for the distribution and scalability of RESTful service solutions in practice.

Q: Explain the relationship between resources, models and views? What is meant by view aggregation?

A: A first-order approximation, in API design, would try to map resources to server-side models on a near one-to-one basis. A ***model*** is the name given to some abstraction of internal (usually) persistent service state (e.g. a database table). For example, a products database table would be represented by a products model which would map to a RESTful products resource. Following along those lines, you could imagine other tables and models which would represent orders, invoices, customer accounts and so on. These in turn could be exposed as the corresponding resources in REST.

Whereas a model is the abstraction of the service state (e.g. a database table) a ***view*** is the representation of the model . Views are the representation of a model using some serialisation format like XML or JSON. Effectively when a resource is based on a model, the view also

becomes the representation of the resource. So-called model-view-controller (MVC) API builder frameworks take advantage of the model-and-view-as-resource convention to allow the automated generation of server-side API boilerplate code.

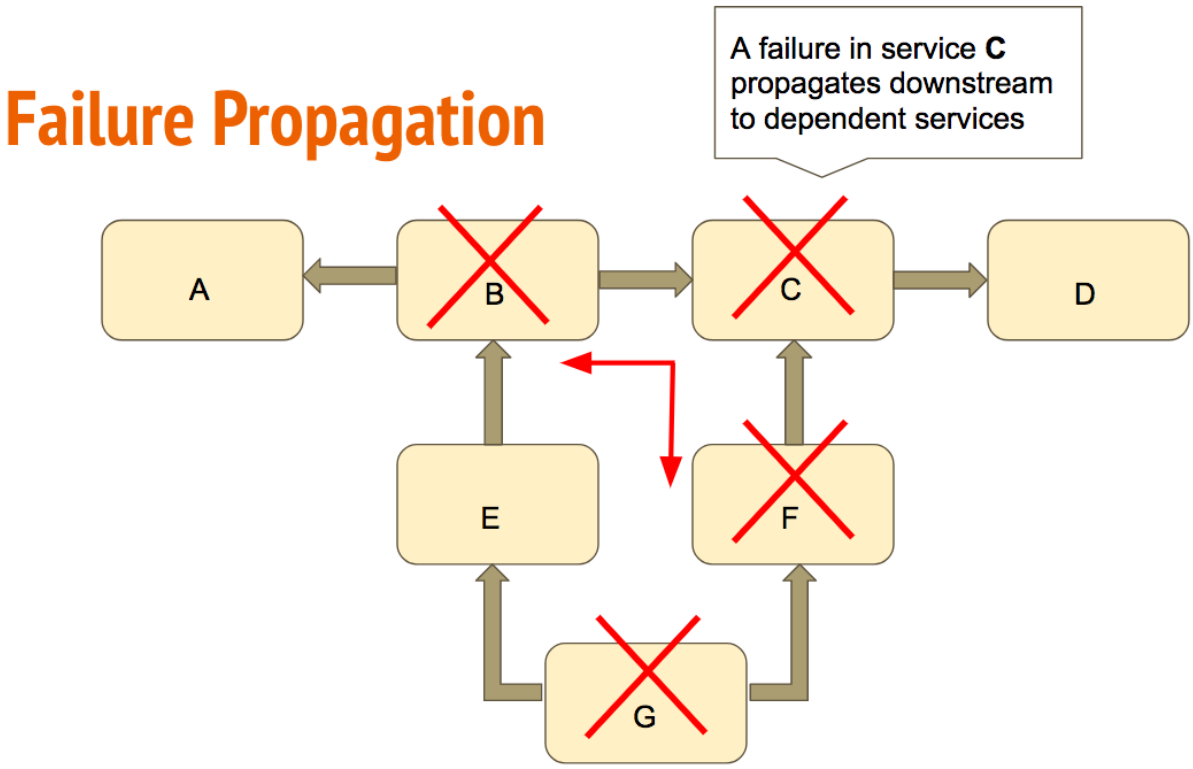
In the simple model-to-resource-mapping approach, the API designer is taking a minimalist approach in the hope that most service consumers can always make use an API like this However, consumers will often want some kind of aggregated views of two or more related resources. In such cases are there two options.

1. Create a new resource on the server side which provides the aggregation required (e.g. some nested structure comprised of multiple resources or resource fragments
2. Craft the aggregation on the client side by making whatever API calls are necessary to do so

Q: Describe the five RESTful operations, giving examples using HTTP. What is meant by idempotence? Mention which of the REST operations are idempotent and why.

A: Recall that, in REST, resources are the primary abstraction of service state and behaviour
A RESTful API is implemented as a message passing interface which carries out some specific. operation on some specific service-exposed resource. Resources are exposed as RESTful endpoints (i.e. URIs). That implementation is closely tied to the features of HTTP. Unlike an RPC-style API where many different kinds of parameterised commands can be defined, REST APIs implement relatively few commands, focusing on the so-called ***CRUD*** operations on resources. Idempotence is the property of an operation such that the operation can be applied multiple times to some value without changing the outcome beyond its first application
In the context of REST, we can say that some operations are idempotent if, after the first application of an operation on a resource (which may alter service state), subsequent applications of that operation don't alter the service state, no matter how many times that operation is later applied. Consideration of idempotence is important in the context of a message passing API over a latent, unreliable network as clients need guaranteed API semantics in the face of potential network partitions.

Q: Explain the problem of failure propagation in SOA systems. What are the desirable characteristics of an API versioning system? What are the two kinds of API compatibility?



The API publisher has the responsibility to ensure that the consumer has the best possible experience using the API. The consumers of service APIs in the enterprise may be implementing business-critical functions, the failure of which could impact adversely on business continuity In an SOA environment, failures in upstream services may (and usually do) cause failures to propagate to downstream dependent services. While by no means the only reason, one of the most common sources of errors occurs when changes to an API doesn't fully appreciate some of its behaviours being depended on by consumers - including bugs!

The idea behind versioning is to provide a shorthand to API consumers regarding which features of the API are supported and how they work. This recognises that APIs do change and that consumers need to be aware of those changes and be able to make an assessment of impact. Versioning schemes take many forms and which one is best depends on the circumstances and particulars of a service and its consumers. A good scheme should convey the following information:

1. **API stability** - meaning how likely it is to change and be relied upon
2. **Major changes** - that new features have been added or existing ones changed
3. **Minor changes** - that existing features have been updated (e.g. bug fixes)
4. **Build identifier** - pinpoints the precise origins (contributing codebase) of the API version (often related to the underlying source code control system)

The idea of API compatibility is to allow for the interoperability between one version of an API and a past version of itself or a future version of itself. There are two kinds to consider:

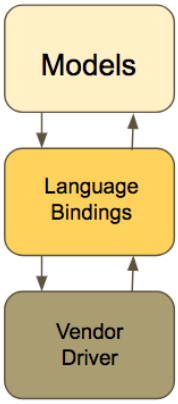
1. **Backward compatibility** - changes to the API still allow legacy API consumers to transparently interoperate with the new version as if it was the old version (i.e. the client cannot distinguish between them). This type is usually critical

2. **Forward compatibility** - the API is designed in such a way that it will transparently interoperate with a future version of itself allowing clients using a new version of the API to work with legacy services at least to the extent of the functionality of the functionality offered by the legacy API

Q: Describe the major elements of the logical data model. Describe how it abstracts the details of database access in the application tier.

A: The logical data layer abstracts the database logic and entities in to some language-specific models (e.g. classes and objects). In theory, the logical data layer should be entirely independent of the database vendor implementation. The enterprise developer's should be hidden from the details of how the database accepts queries, executes them, returns results and the specifics of how data is represented in the database. To achieve this, the logical data layer itself is made up of a number of internal logical layers in turn.

- **Models** - Language-specific data structures which abstract the representation of the database entities (e.g. classes and objects)
- **Language bindings for SQL** - Generic API for interface from the middleware language to SQL services on the database (e.g. execution of queries)
- **Database vendor driver** - Implementation-specific logic for translating SQL bindings into the raw connection API (e.g. binary protocol for parameterised query data)



Models are the language-specific abstraction of the database entities. In RESTful API services, models provide the architectural bridging between database entities and the exported resource state and behaviour. In the simplest view, a model is just an internal memory representation of persistent data structures which can be validated, processed and represented independently of the database itself. In practice, models are implemented as classes and objects in OOP languages like Java.

The middleware tier is responsible for implementing the logical data model which is an abstraction of the database entities, relationships and data representation. The LDM itself comprises a model layer, a SQL binding layer and a database driver. The application programmer view of the database is provided by a simpler model-based abstraction of the underlying complexity.

Q: Describe in detail the pathology of a SQL injection exploit. What should the application developer to avoid this kind avoid this kind of vulnerability.

A: What the attacker wants to do is exploit a query formation vulnerability by repeatedly sending queries to the service with malformed input in the hope that he will find a flaw. Once a flaw is found, the attacker can potentially mount arbitrary attacks on the system to learn more about the schema, the data and extract or modify critical values. Let's consider how this can be done using a number of examples.

Suppose that a user login check query string was built as a string as follows: (Vulnerability)

```
const q = `
    SELECT field-list
    FROM users u
    WHERE u.email = '${email}'
    AND u.passwd_hash = crypt('${passwd}', u.passwd_hash);
`;
```

Update Vulnerability:

```
const q = `
    UPDATE products SET
        products.name = '${name}'
    WHERE products.id = '${id}';
`;
```

The fundamental problem with generating SQL query strings this way is that the attacker has the opportunity to inject the bad code before the query planner gets a chance to parse it. The fix is to generate the queries a different way by:

- Using pre-parser function which checks the validity of the query before it is executed (e.g. `mysqli::escape_string` from PHP) - not as secure in practice
- Using prepared statement or parameterised queries
- Using a stored procedure with typed arguments
- Isolating the execution within a tight security sandbox using database privileges