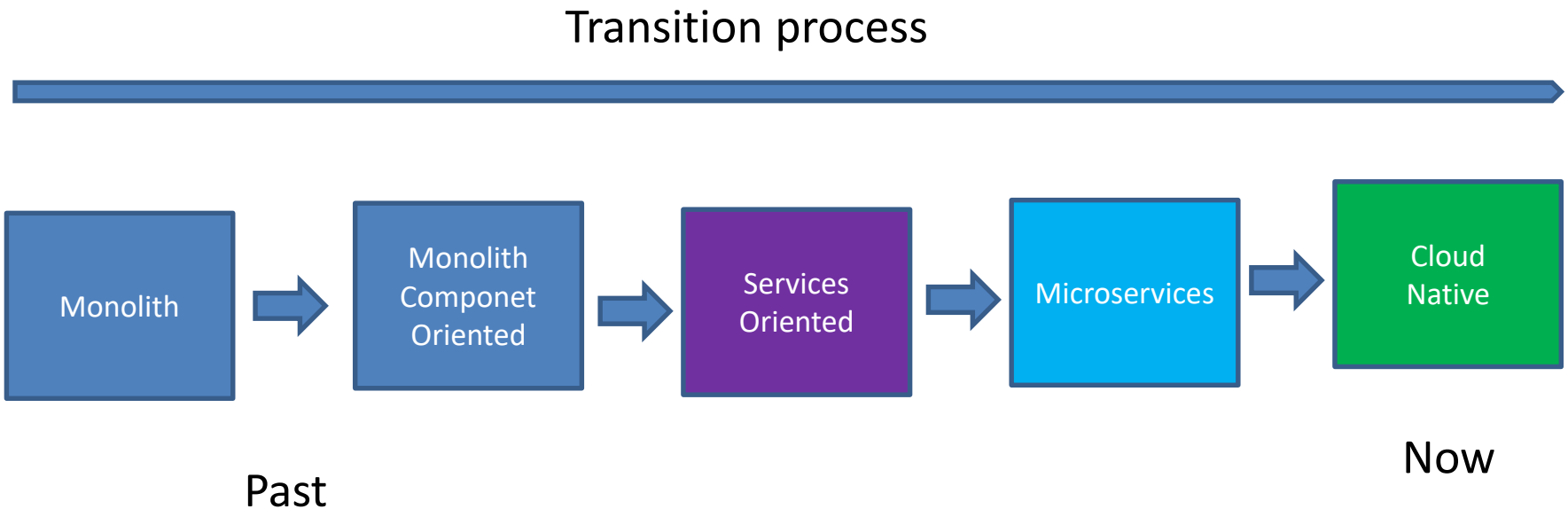
Distributed Application Evolution

Application Architecture



Distributed Component Technology

EJB

Enterprise Java Beans

Enterprise Java Beans

Definition

A combination of server-side components and distributed object technology from Java RMI and CORBA

Enterprise Java Beans

Remainder

- Remember the following Distributed object characteristics
  - Location
  - Latency
  - Concurrency
  - Existence
  - Failure points (’at-most-once’ call semantic)

Please never forget them

Enterprise Java Beans

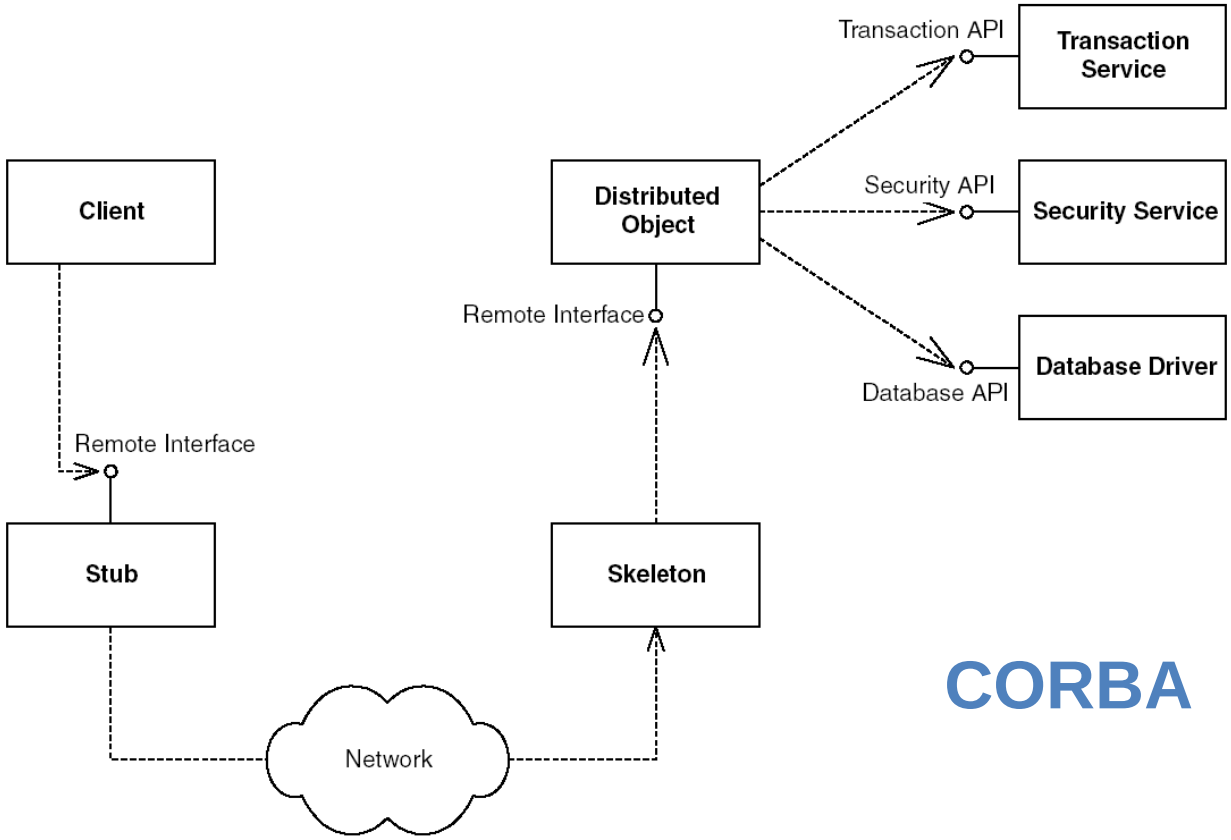
The Vision

- To develop scalable and portable applications
- To simplify application development by automatically maintain/provide low-level system facilities
  - Resource management
  - Persistence
  - Concurrency
  - Security
  - Transaction integrity
  - Naming

This is important and we will have to keep this in our mind for the decision whether or not te use EJBs for a given application

Enterprise Java Beans

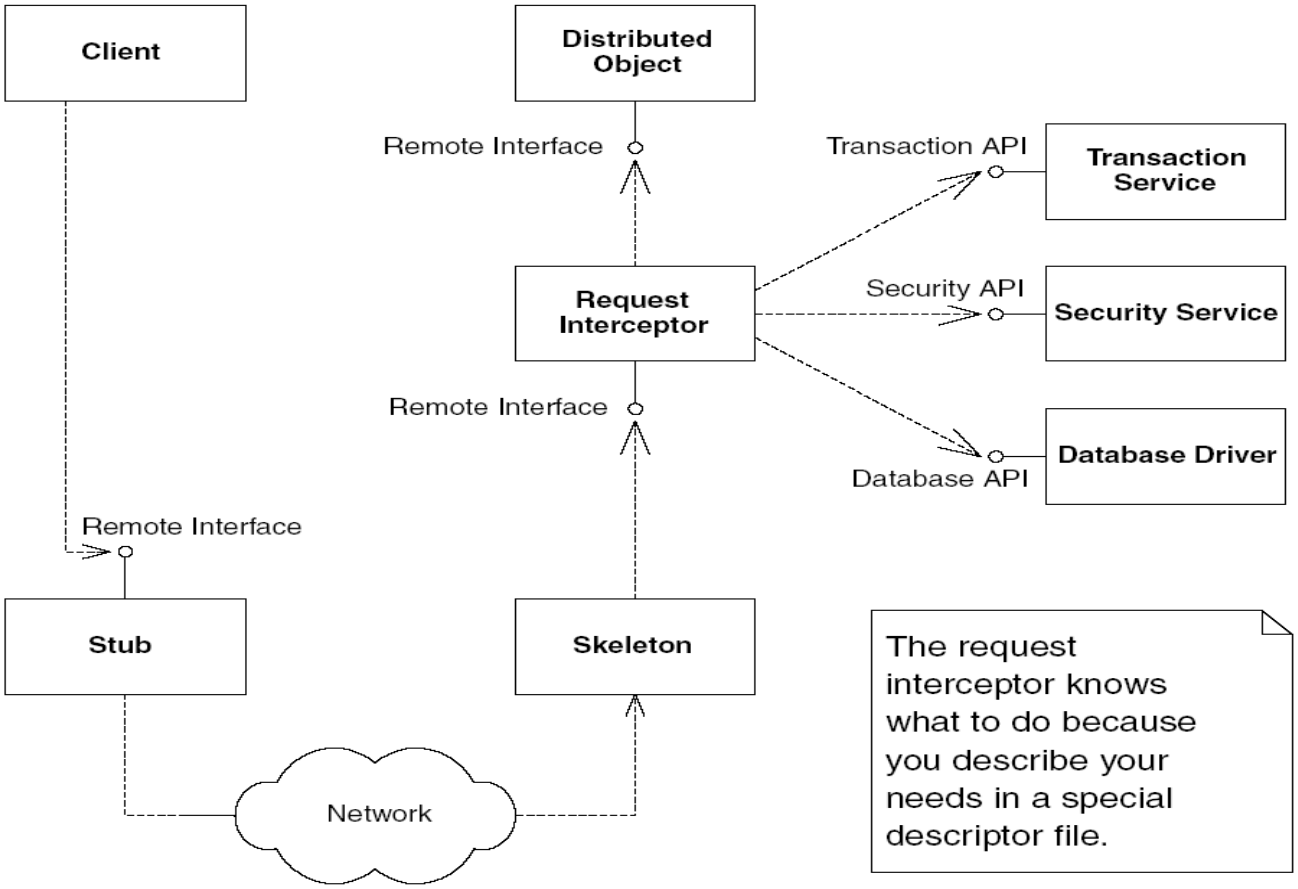
Explicit middleware



CORBA

Enterprise Java Beans

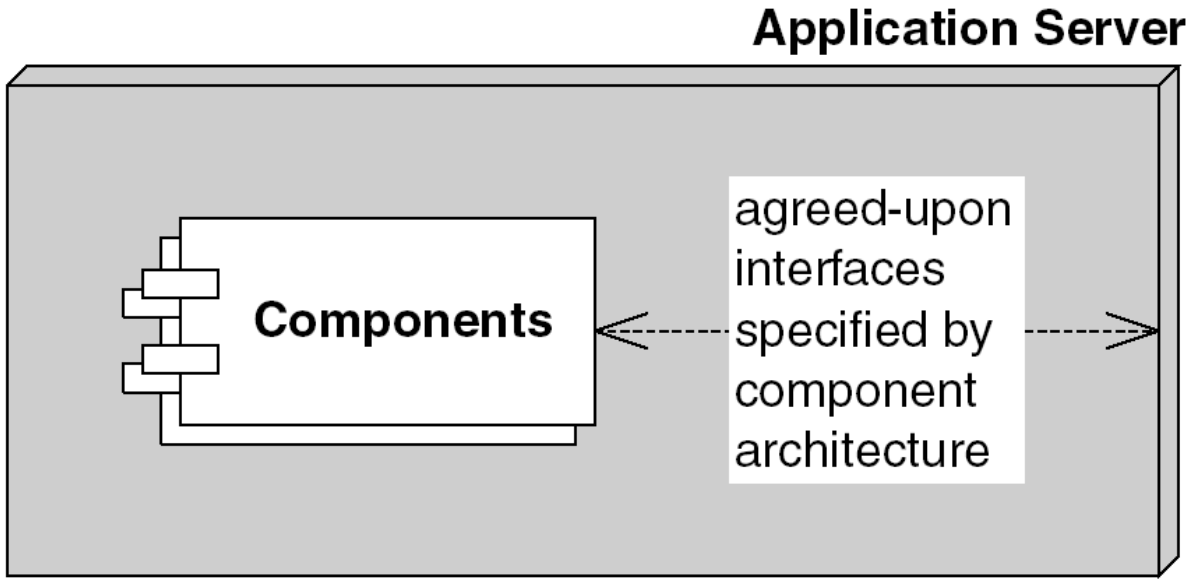
Implicit middleware



Distributed component technology

Enterprise Java Beans

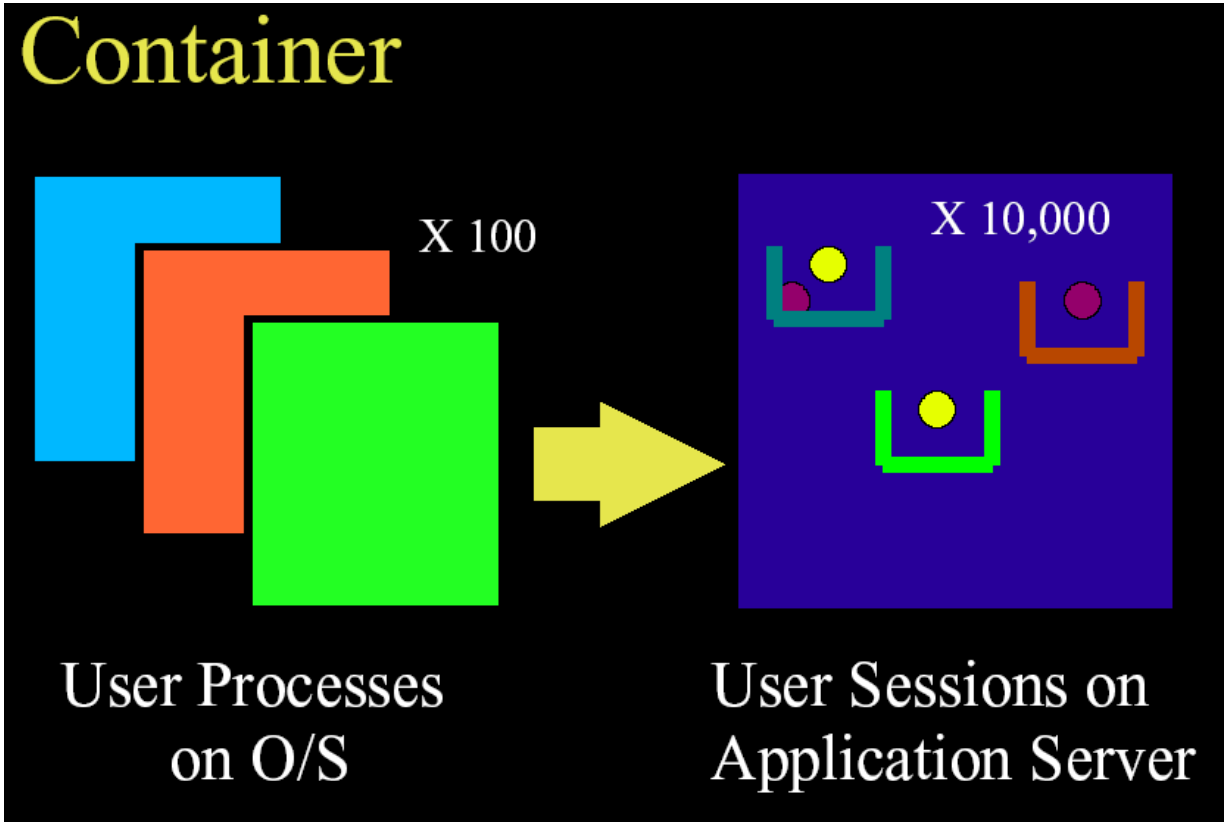
Component architecture



Server provides system services

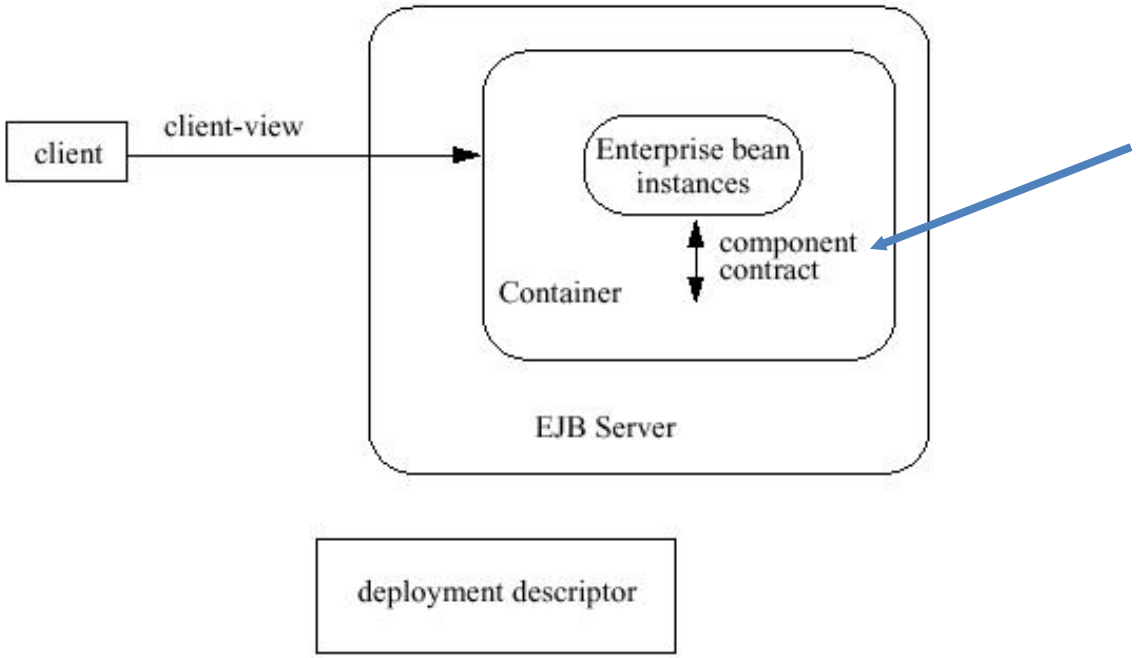
Enterprise Java Beans

Component architecture



Enterprise Java Beans

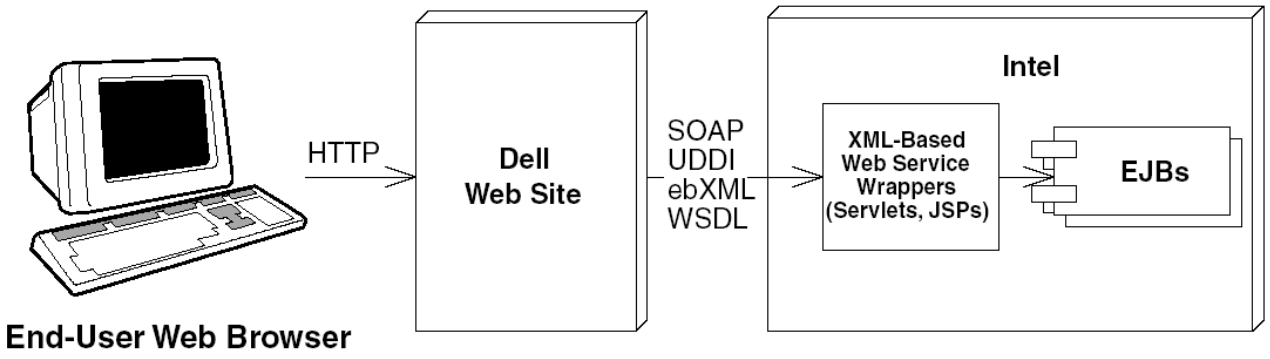
The high-level architecture



Component/Container based on callbacks implemented by the component

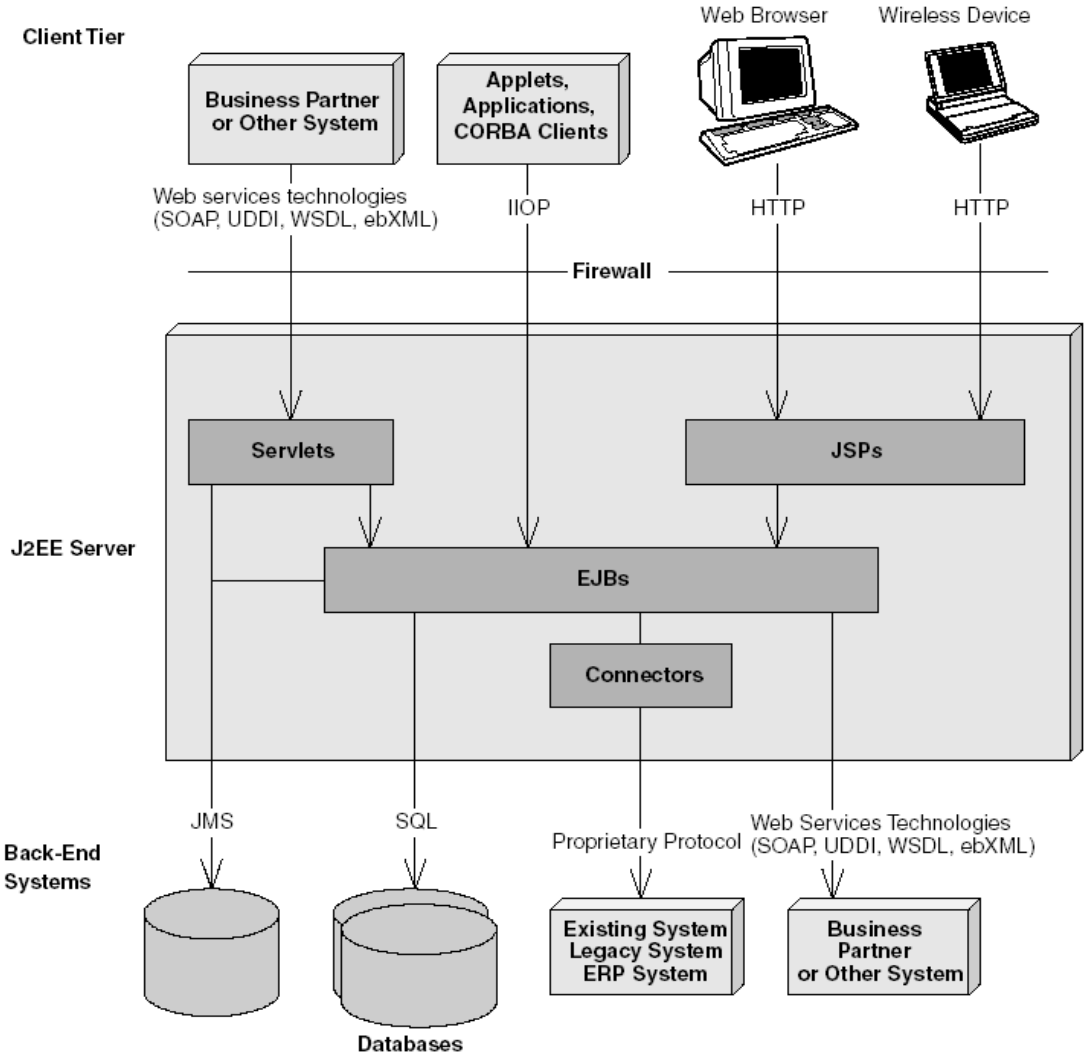
Enterprise Java Beans

EJB back-end to WS



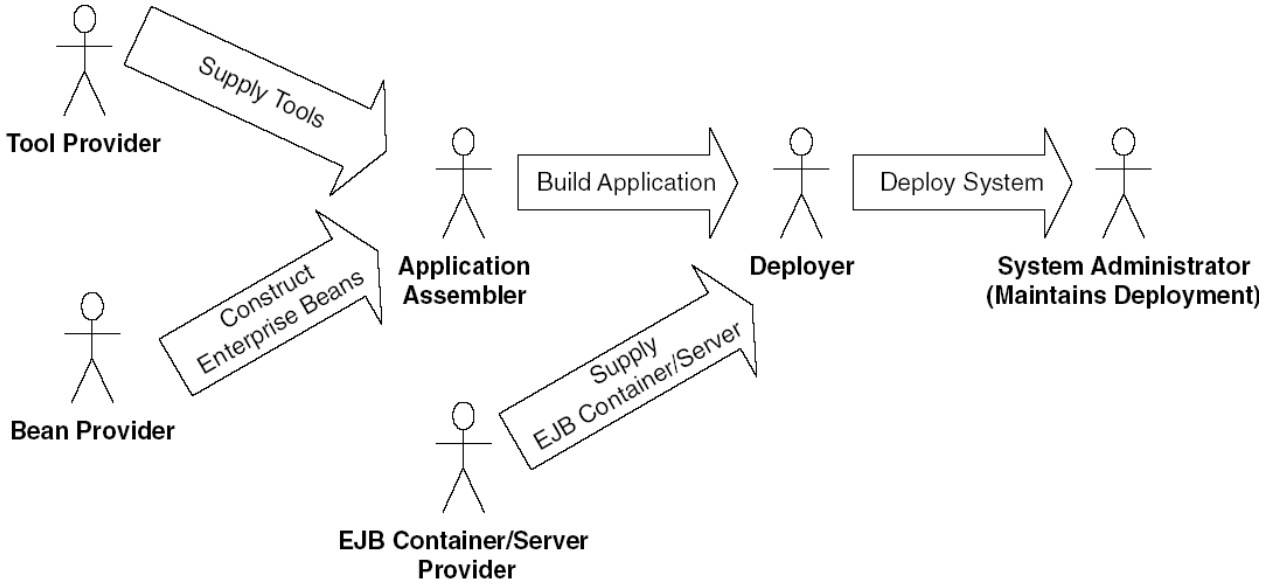
Enterprise Java Beans

vs. J2EE



Enterprise Java Beans

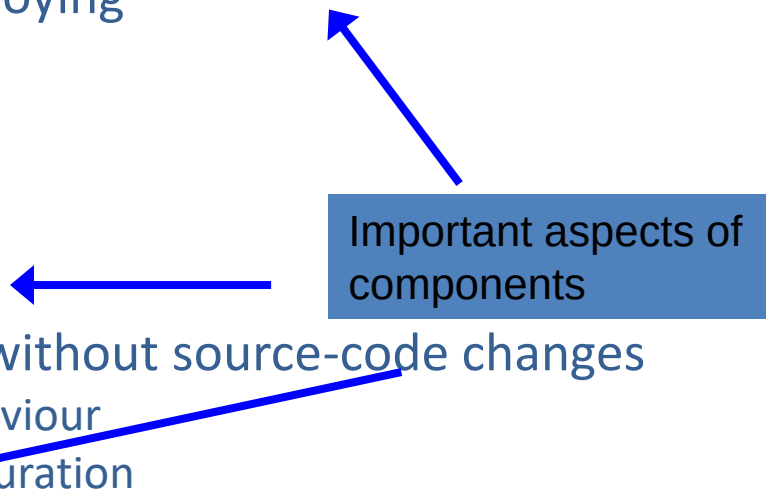
EJB parties



Enterprise Java Beans

EJB Component characteristics

- Contain typically business logic or business process modeling operating on data
- The Container manages the whole life cycle
- Declarative customization when deploying
  - Transaction management
  - Security management
  - Concurrency management
  - Component configuration
  - .....
- Can be reused in other applications without source-code changes
  - Fx. with a different transactional behaviour
  - Fx. with a different component configuration



Enterprise Java Beans

EJB Component characteristics

- Client access always enter through the Container
- The Client view to an Enterprise Java Bean are not affected by the Container interception (transparent to the Client)
- This in fact ensures that Enterprise Java Beans can be deployed in dífferent vendor Container implementations without the need for source code changes and recompilation

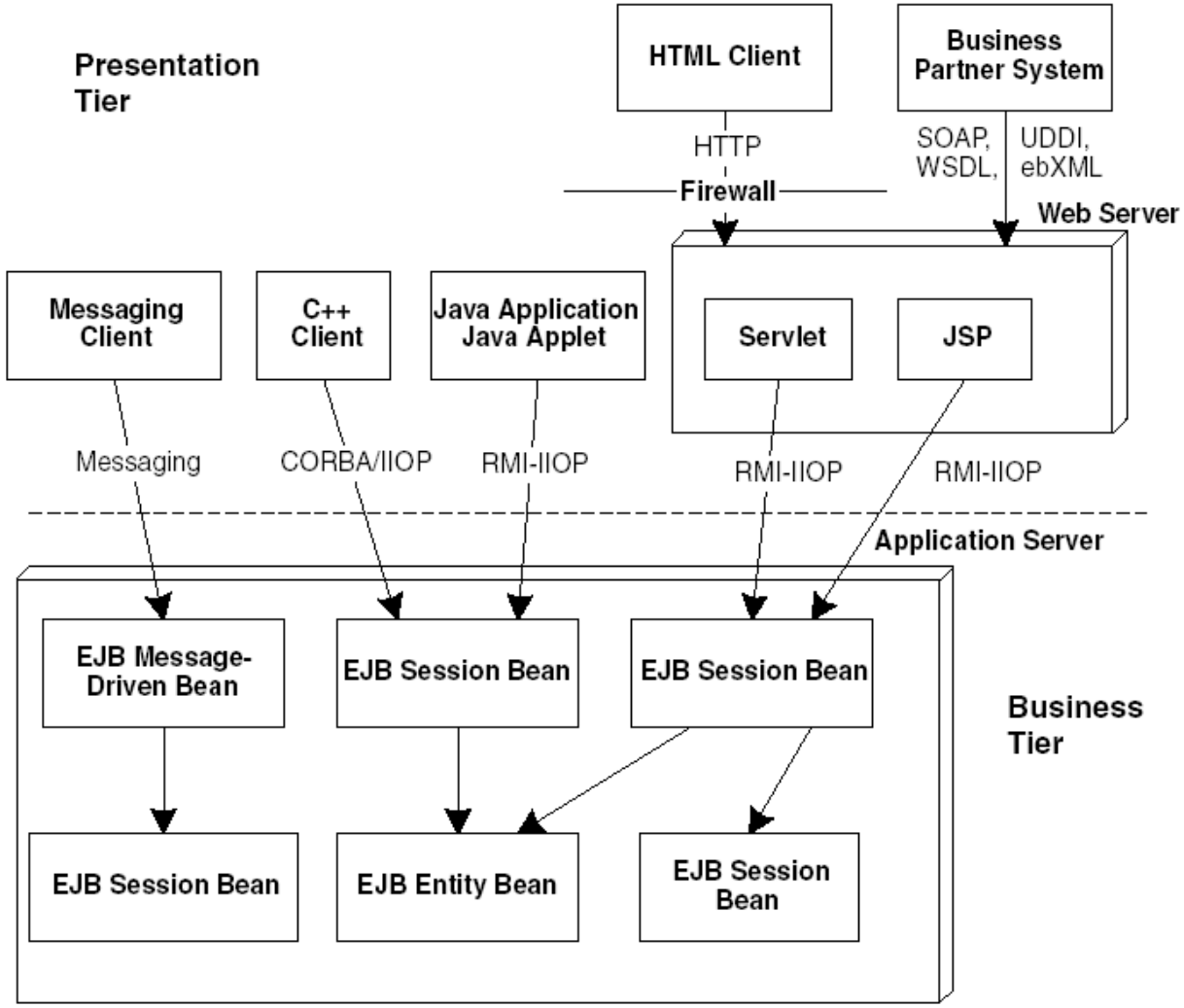
Enterprise Java Beans

Client view contract

- The contract between the Client and the Container
- Different types of clients:
  - Another Enterprise Java Bean deployed in the same Container or another vendors Container
  - Java Application, Servlet or Applet
  - The Client View for an Enterprise Java Bean can be mapped to non-Java environments like CORBA clients written in a non-Java programming language
    - The RMI/IIOP subset is used
  - Both local and remote -> different parameter semantic

Enterprise Java Beans

Types of clients



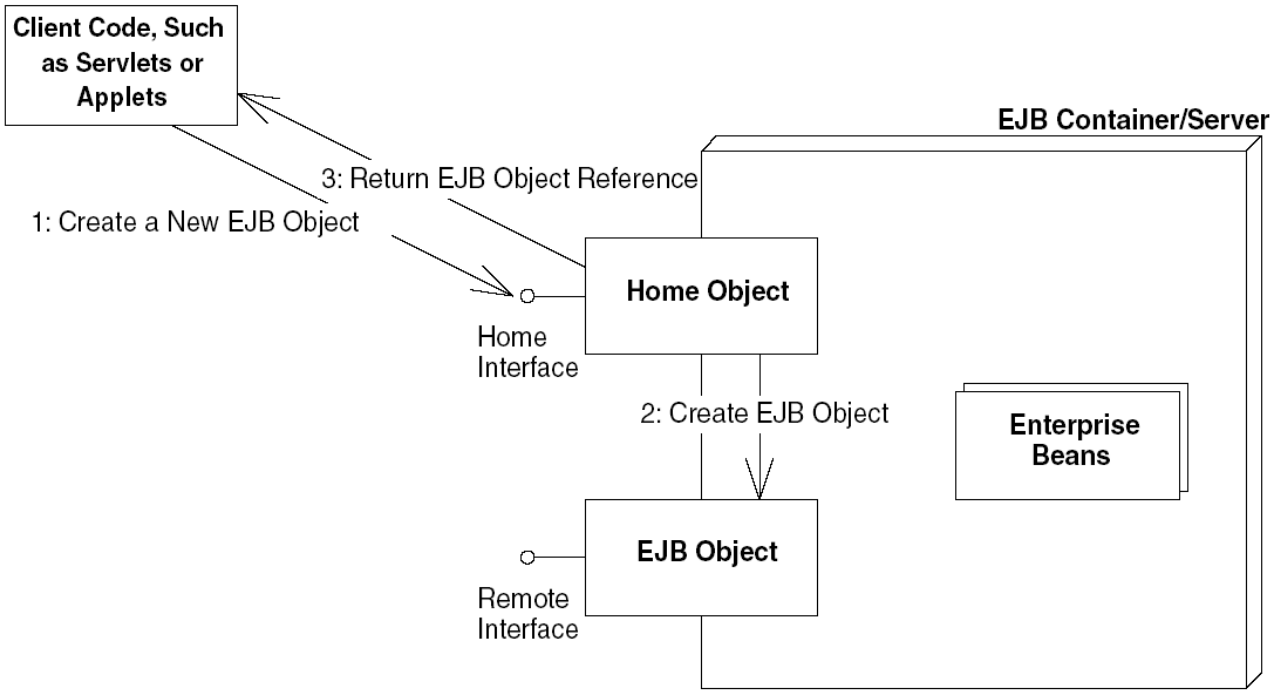
Enterprise Java Beans

Client view contract

- The Client View contract contains:
  - **Home interface**
    - Contains factory-/life cycle methods for EJB objects of the same type as the Enterprise Java Bean
    - Specificed by the Bean Provider
    - The EJB Container creates the class implementing the home interface
    - Clients use JNDI to obtain an remote reference to the Home Interface
  - **Remote Interface**
    - For remote clients (may be all EJB types + servlets, applets, Java apps etc.)
    - Based on the Java RMI interface
    - Provides location transparency
  - **Local Interfaces**
    - For local clients (may be all EJB bean types)
    - Based on the normal Java non-distributed interface (pass-by-reference)
    - Provides NO location transparency

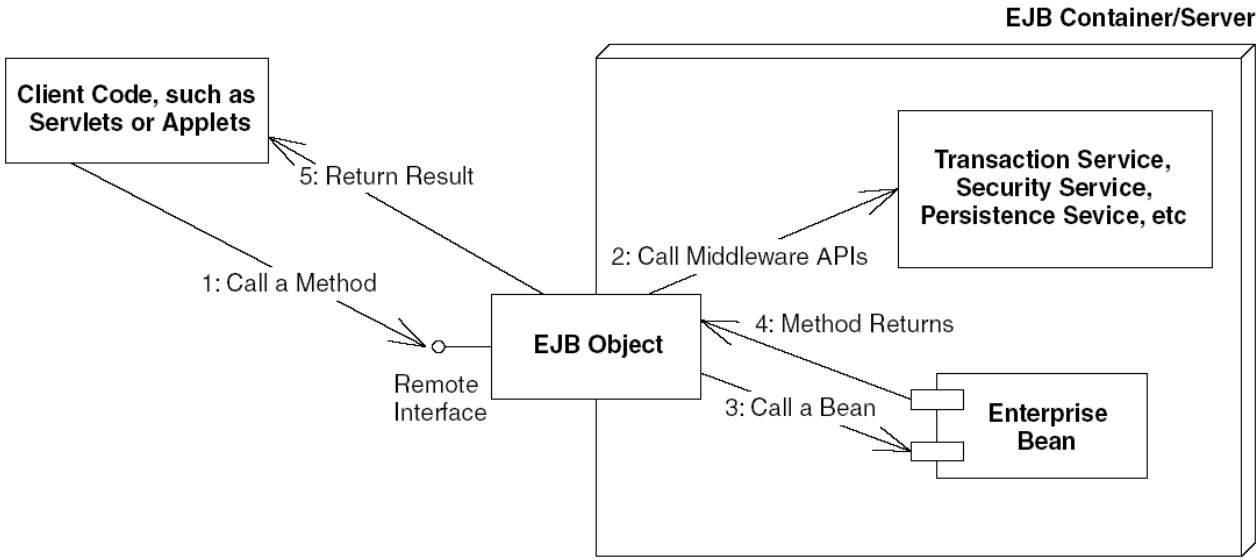
Enterprise Java Beans

Home interface



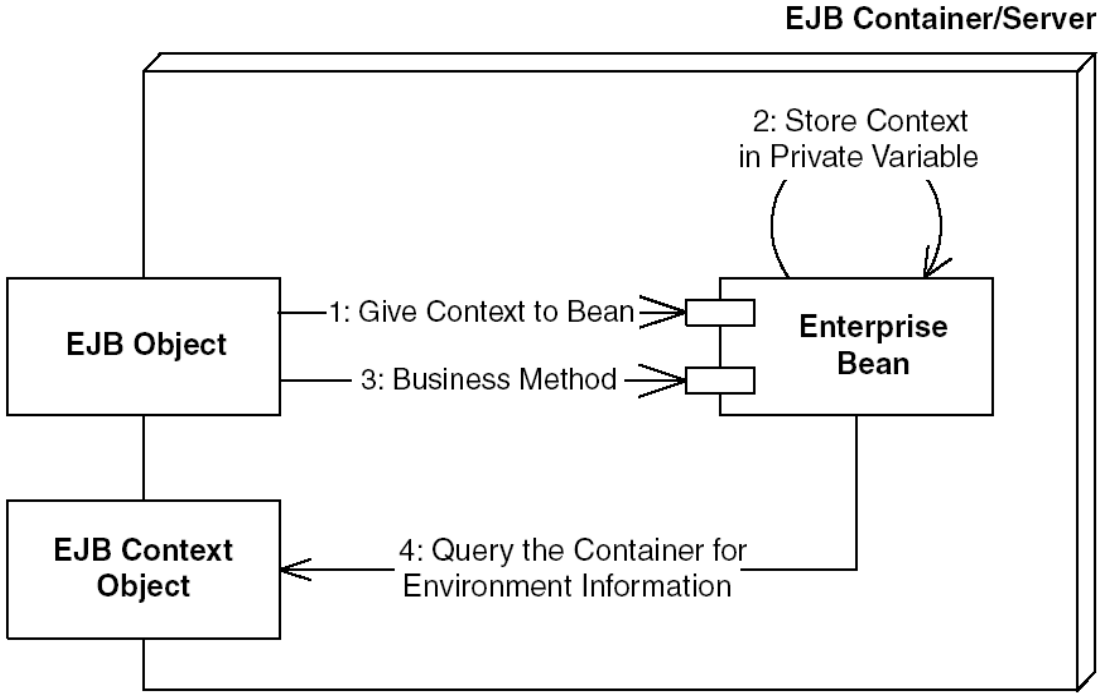
Enterprise Java Beans

EJB object



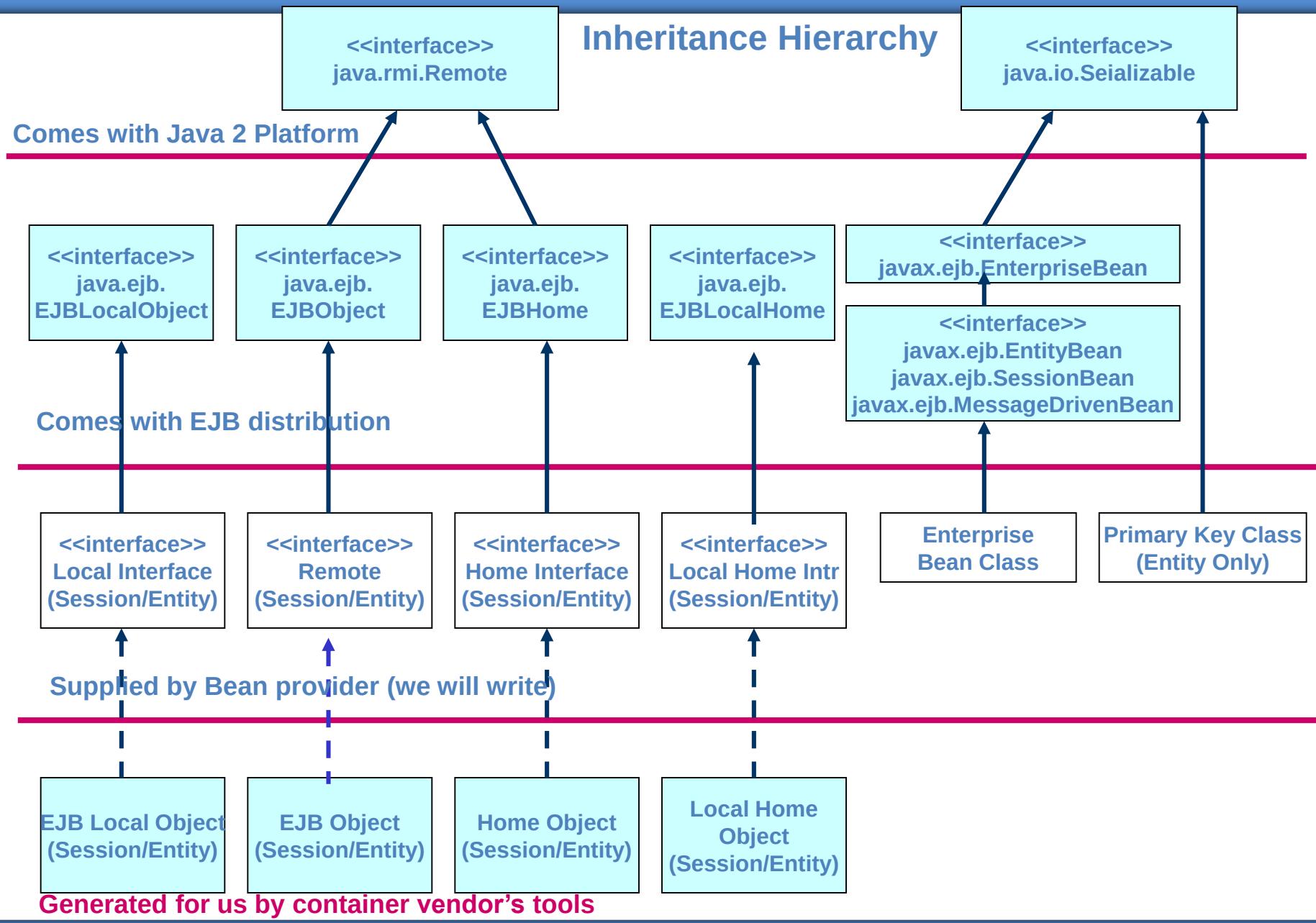
Enterprise Java Beans

Context



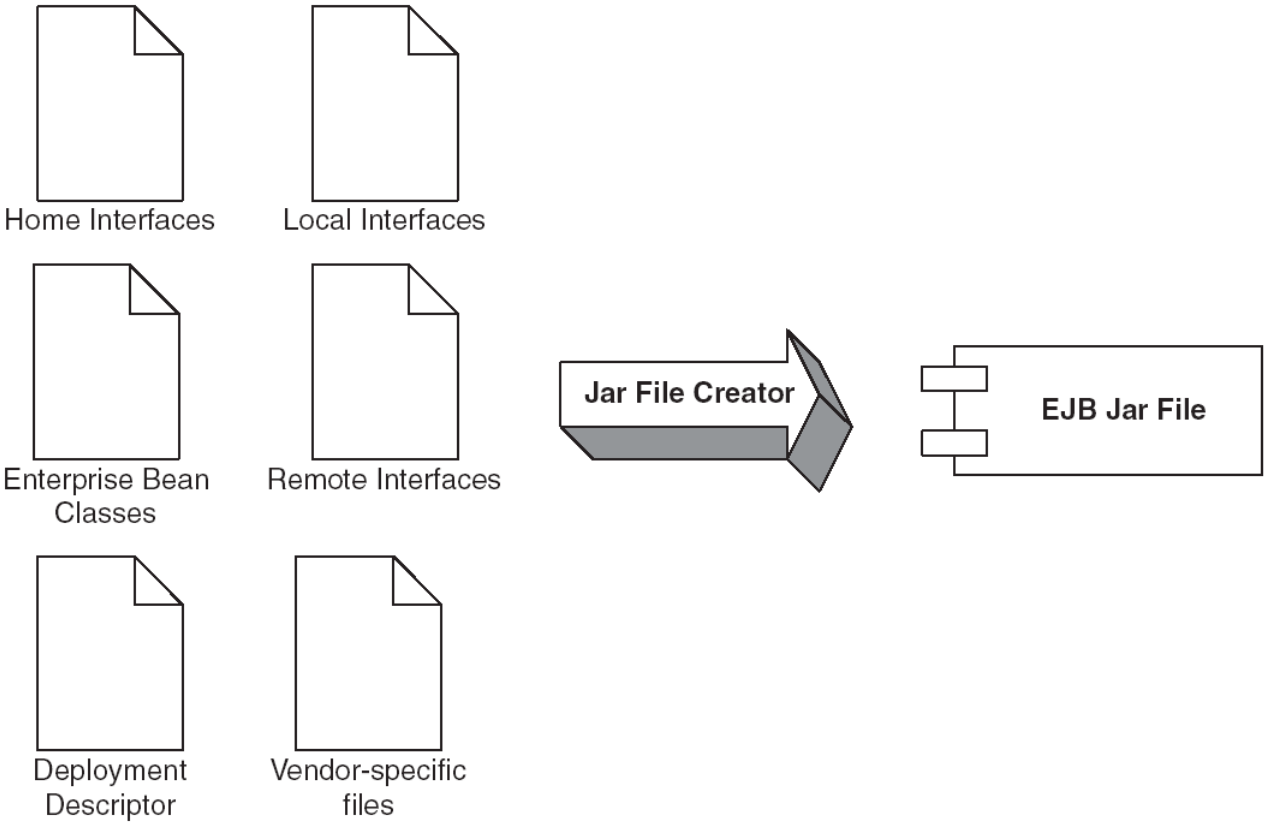
EJBContext = Gateway to the container

Inheritance Hierarchy



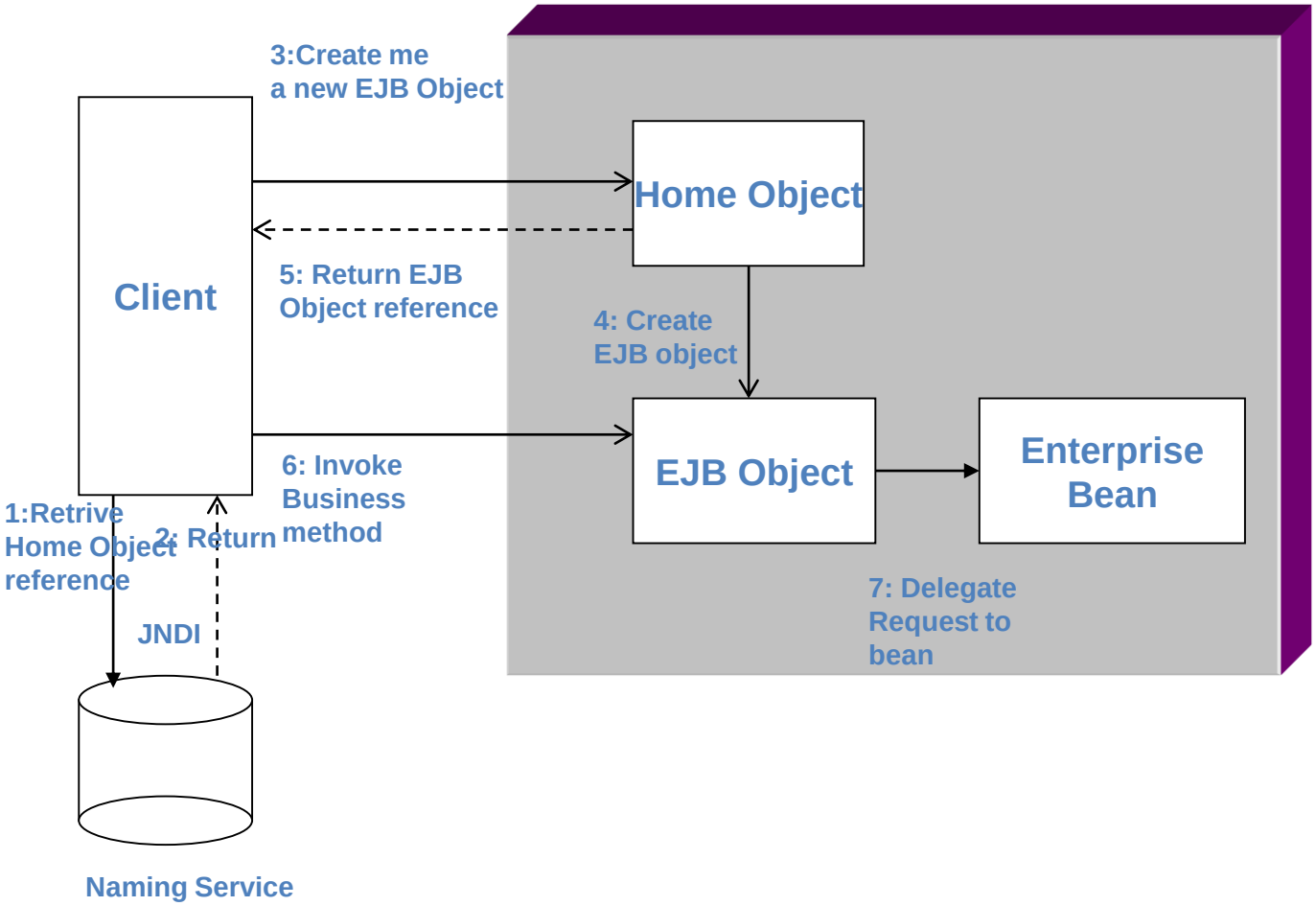
Enterprise Java Beans

EJB Jar File



Enterprise Java Beans

Invocation model



Introduction to Service Oriented Architecture (SOA)

Attributes of physical services

- Well defined, easy-to-use, somewhat standardized interface
- Self-contained with no visible dependencies to other services
- (almost) Always available but idle until requests come
- "Provision-able"
- Easily accessible and usable readily, no "integration" required
- Coarse grain
- Independent of consumer context,
  - but a service can have a context
- New services can be offered by combining existing services
- Quantifiable quality of service
  - Do not compete on "What" but "How"
  - Performance/Quality
  - Cost
  - …

Context, Composition and State

- Services are most often designed to ignore the context in which they are used
  - It does not mean that services are stateless they are rather context independent !
  - This is precisely my / the definition of “loosely coupled”
    - Services can be reused in contexts not known at design time
- Value can be created by combining, i.e. “composing” services
  - Book a trip versus book a flight, car, hotel, …

Service Interfaces

- Non proprietary
  - All service providers offer somewhat the same interface
- Highly Polymorphic
  - Intent is enough
- Implementation can be changed in ways that do not break all the service consumers
  - Real world services interact with thousands of consumers
  - Service providers cannot afford to “break” the context of their consumers

Intents and Offers

- Service consumer expresses “intent”
- Service providers define “offers”

- Sometimes a mediator will:
  - Find the best offer matching an intent
  - Advertise an offer in different ways such that it matches different intent
- Request / Response is just a very particular case of an Intent / Offer protocol

Service Orientation and Directories

- Our society could not be “service oriented” without the “Yellow Pages”
- Directories and addressing mechanisms are at the center of our service oriented society
- Imagine
  - Being able to reach a service just by using longitude and latitude coordinates as an addressing mechanism?
  - Only being able to use a service if you can remember its location, phone or fax number?

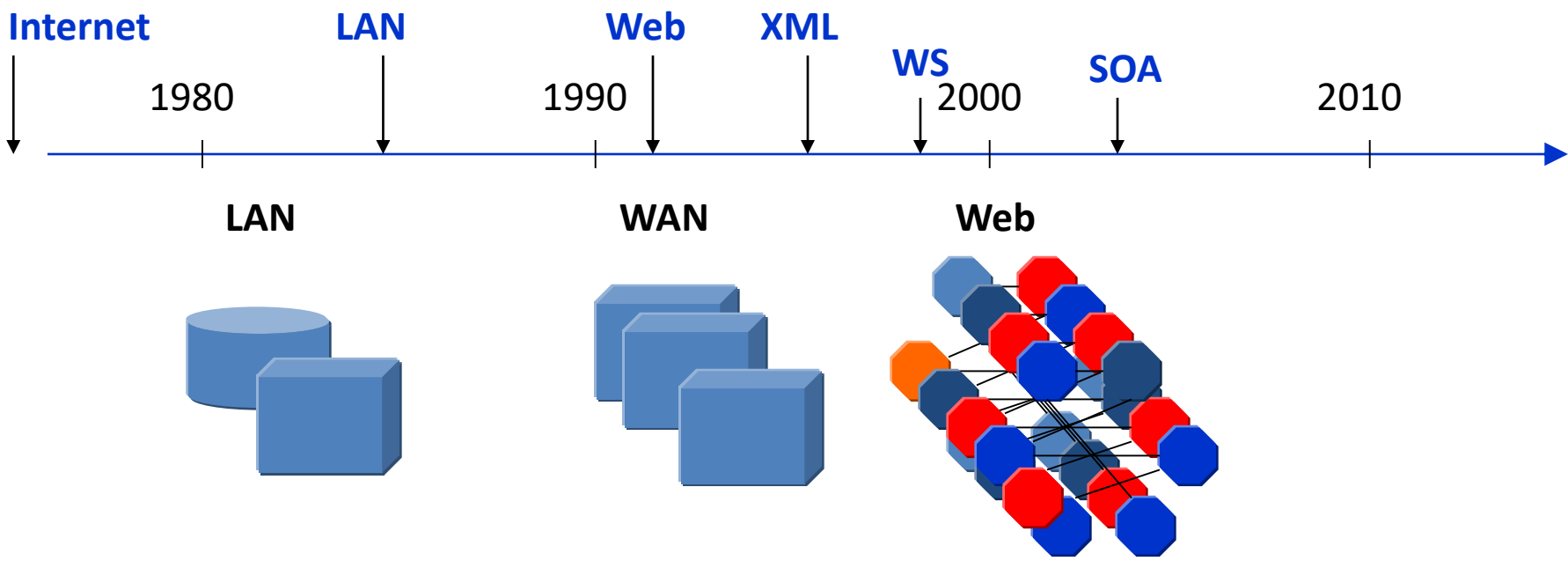
Service Orientation is scalable

- Consumers can consume and combine a lot of services since they don't have to know or "learn" how to use a service
- Service providers can offer their services to a lot more consumers because by optimizing
  - The user interface
  - Access (Geographical, Financial, …)
  - They were able to provide the best quality of service and optimize their implementations

So…

- Service orientation allows us
  - Complete freedom to create contexts in which services are uses and combined
  - Express intent rather than specific requests
- Our society should be a great source of inspiration to design modern enterprise systems and architectures or understand what kind of services these systems will consume or provide

Connectivity Enables Global Processes and Information Access



Seamless Connectivity enables “On Demand” Computing

- Use software as you need
- Much lower setup time, forget about
  - Installation
  - Implementation
  - Training
  - Maintenance
- Scalable and effective usage of resources
  - Provision
  - Billed on true usage
  - Parallelism (CPU, Storage, Bandwidth…)

But Seamless Connectivity is also questioning all our beliefs…

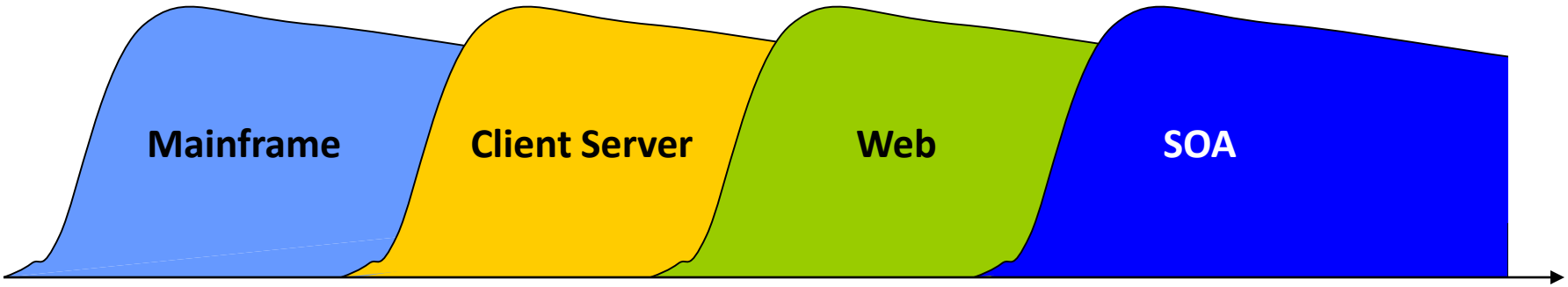
- An application is NOT a single system running on a single device and bounded by a single organization
- Continuum Object … Document
- Messages and Services
  - As opposed « distributed objects »
  - Exchanges becomes peer-to-peer
- Asynchronous communications
- Concurrency becomes the norm while our languages and infrastructures did not plan for it

…we are reaching the point of maximum confusion

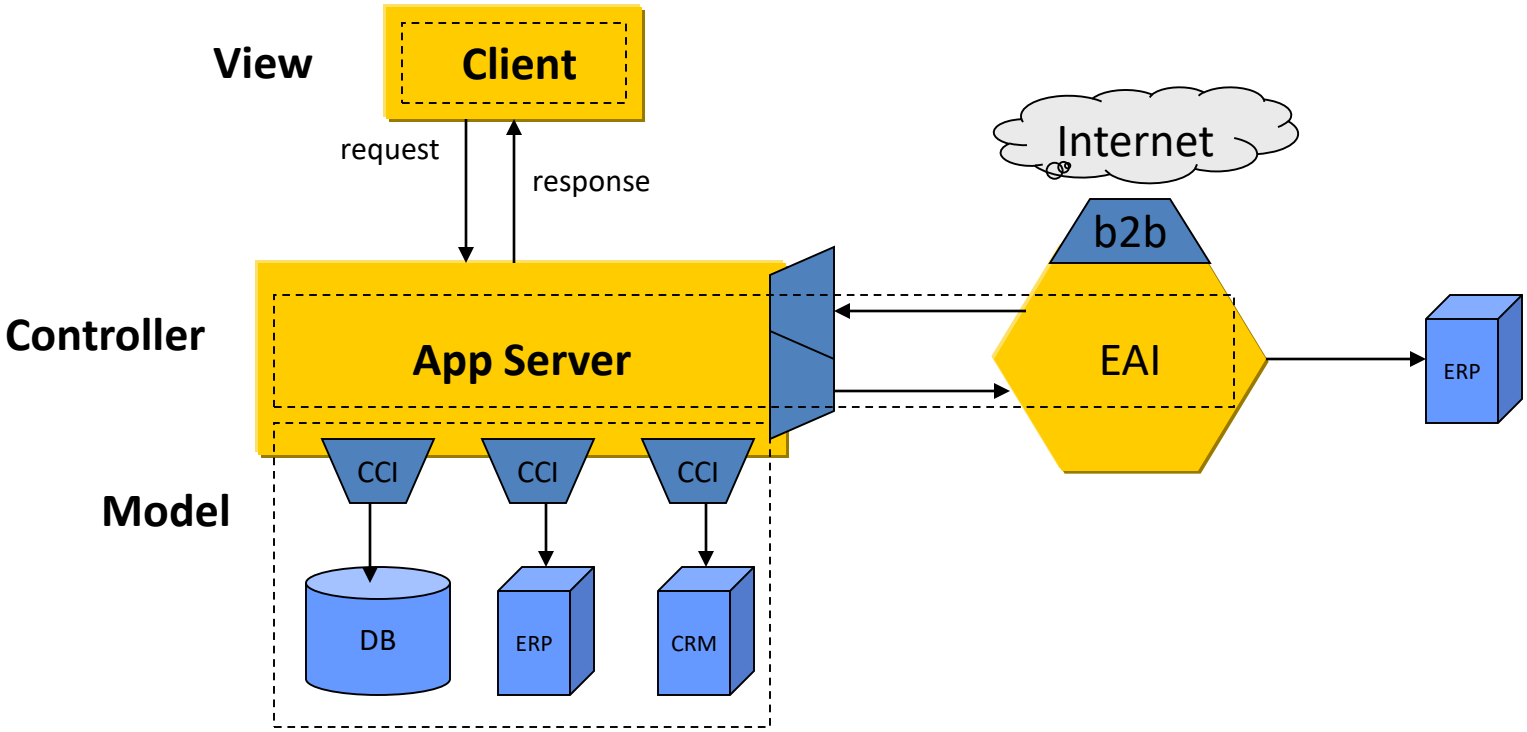
- Federation and Collaboration
  - As opposed to « Integration »
- Language(s)
  - Semantic (not syntactic)
  - Declarative and Model driven (not procedural)
- Licensing and Deployment models
- …

So…

- Today, the value is not defined as much by functionality anymore but by connectivity
  - However, we need a new programming model
- Why SOA ?
  - We are reaching a new threshold of connectivity and computing power



Constructing software in the web era (J2EE, .Net, …)

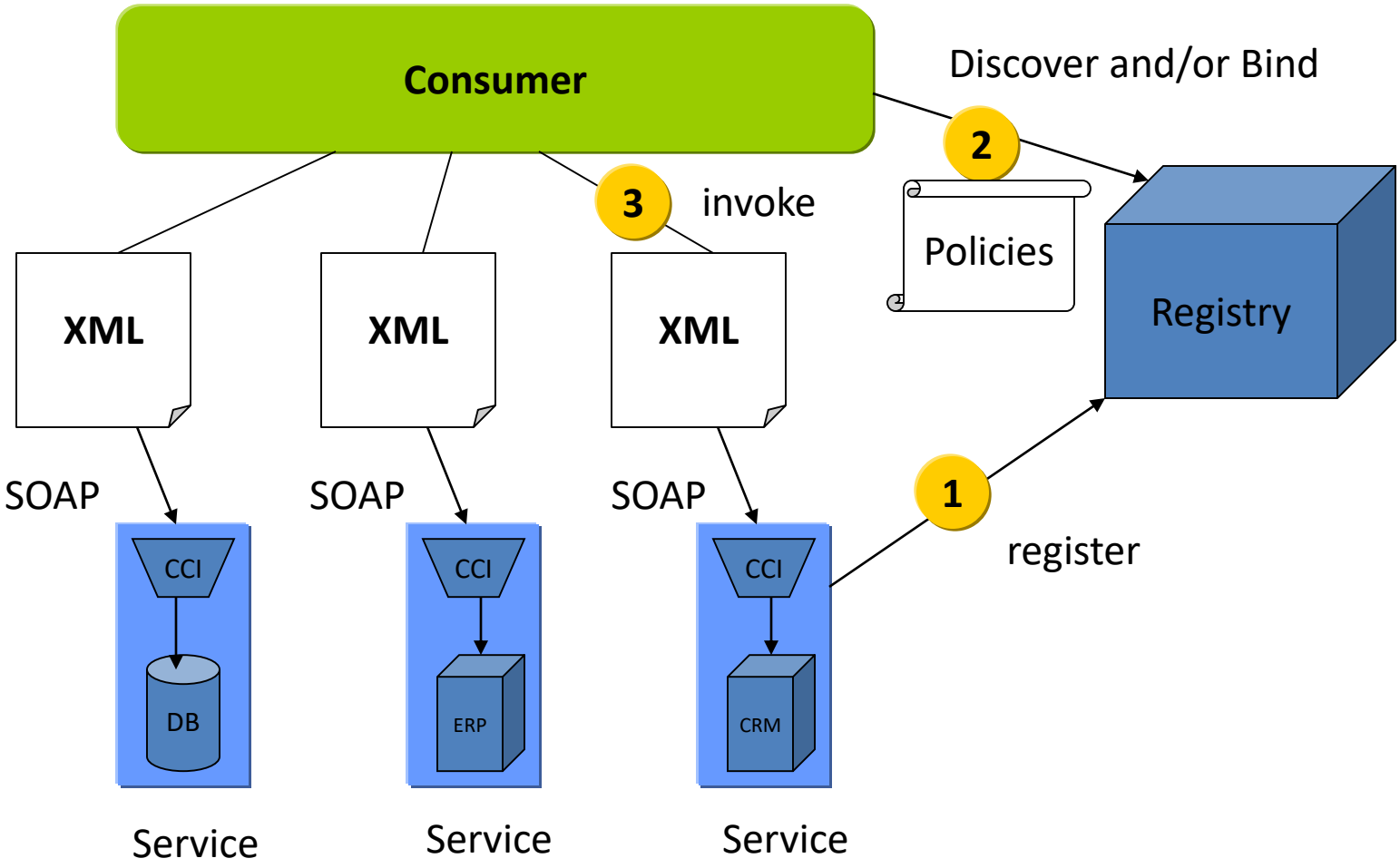


CCI: Client Communication Interface

Why do we Want to Move to a New Application Model Today?

- We are moving away precisely because of connectivity
  - J2EE, for instance was designed to build 24x7 scalable web-based applications
  - Job well done
- But this is very different from, “I now want my application to execute business logic in many other systems, often dynamically bound to me”
  - JCA (J2EE Connector Architecture) is not enough
  - EAI infrastructures are not enough

A Component now Becomes a Service Running Outside the Consumer Boundaries



From Components to Services

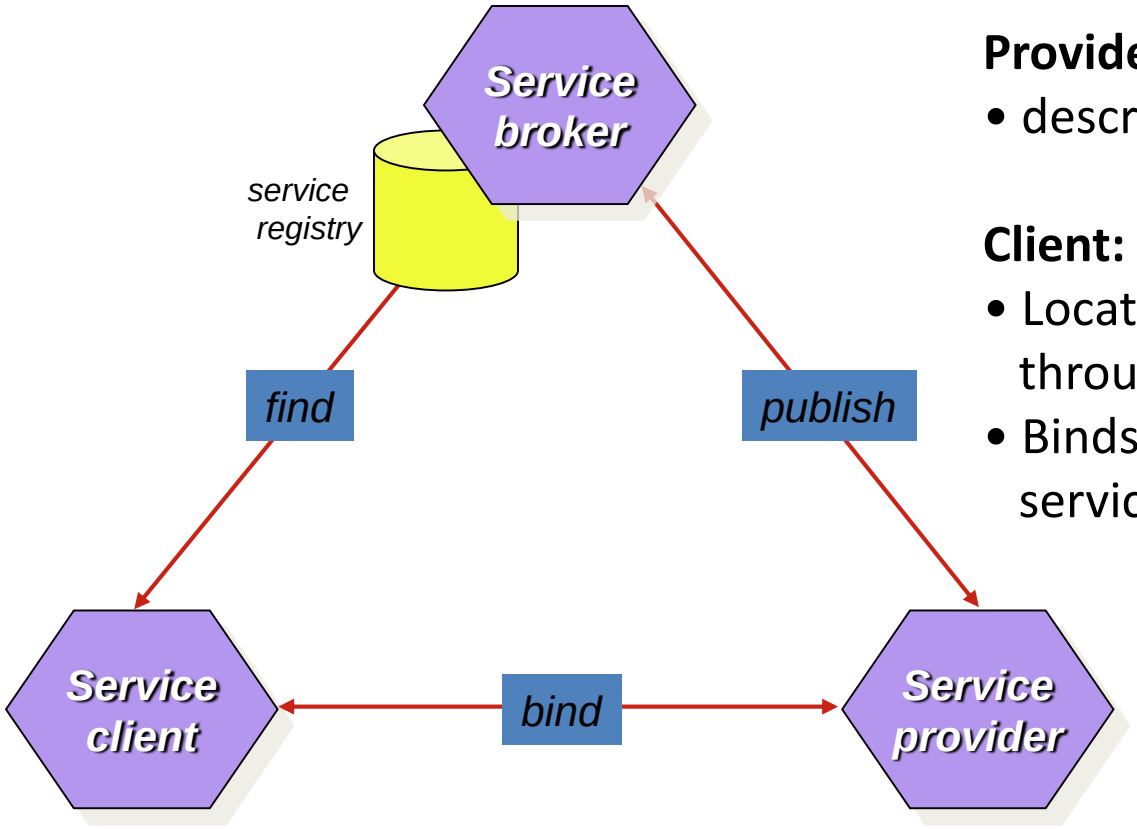
- Requires a client library
- Client / Server
- Extendable
- Stateless
- Fast
- Small to medium granularity

- Loose coupling via
  - Message exchanges
  - Policies
- Peer-to-peer
- Composable
- Context independent
- Some overhead
- Medium to coarse granularity

What is a Service Oriented Architecture?

- Service-oriented architecture (SOA) is an approach to loosely coupled, protocol independent, standards-based distributed computing where software resources available on a network are considered as services.
- The service is designed in such a way that it can be invoked by various service clients and is logically decoupled from any service caller.
- Creates service level abstractions that map to the way a business actually works.
- Leverages investment in existing application assets.

Service Oriented Architecture: Roles & Functions



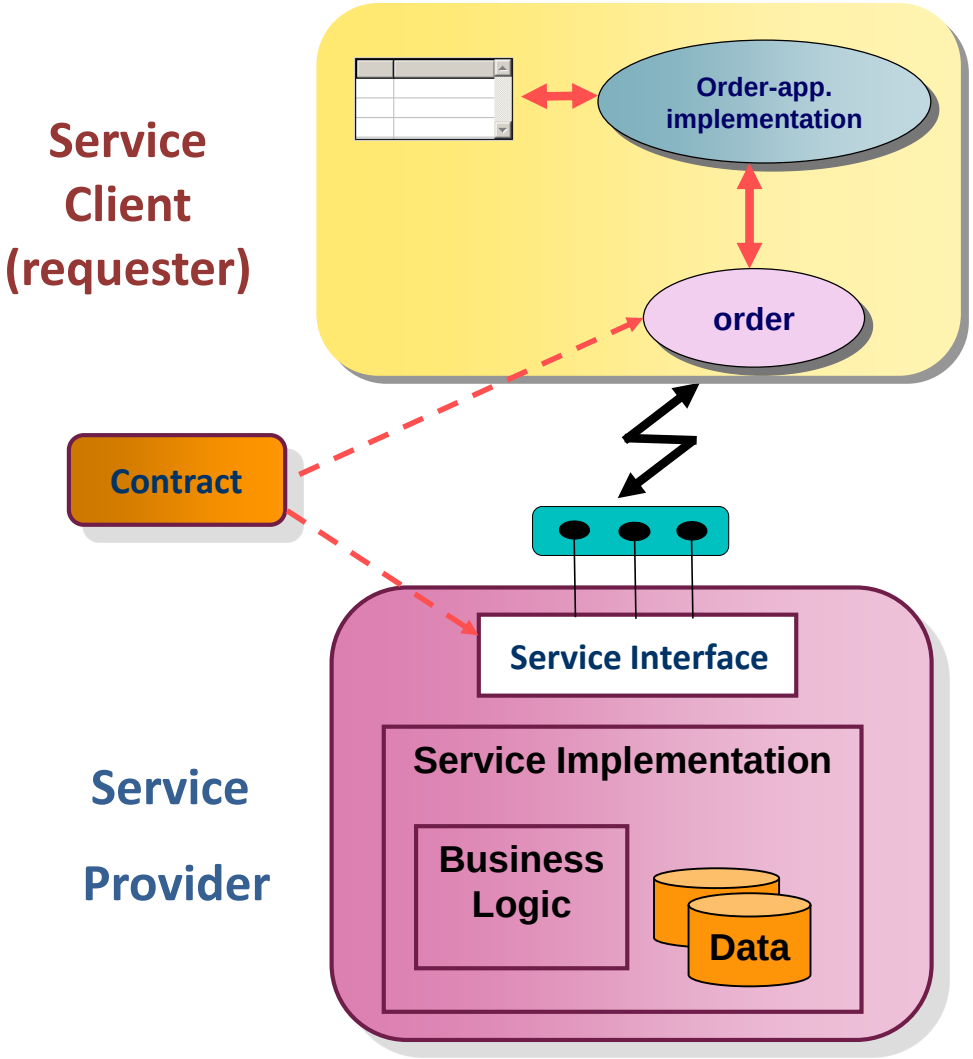
Provider:

- describes & publishes services

Client:

- Locates service provider through service registry
- Binds to & invokes service based service interface

Service Clients and Providers



What Happens to the Technical Services Typically Provided by an Application Server?

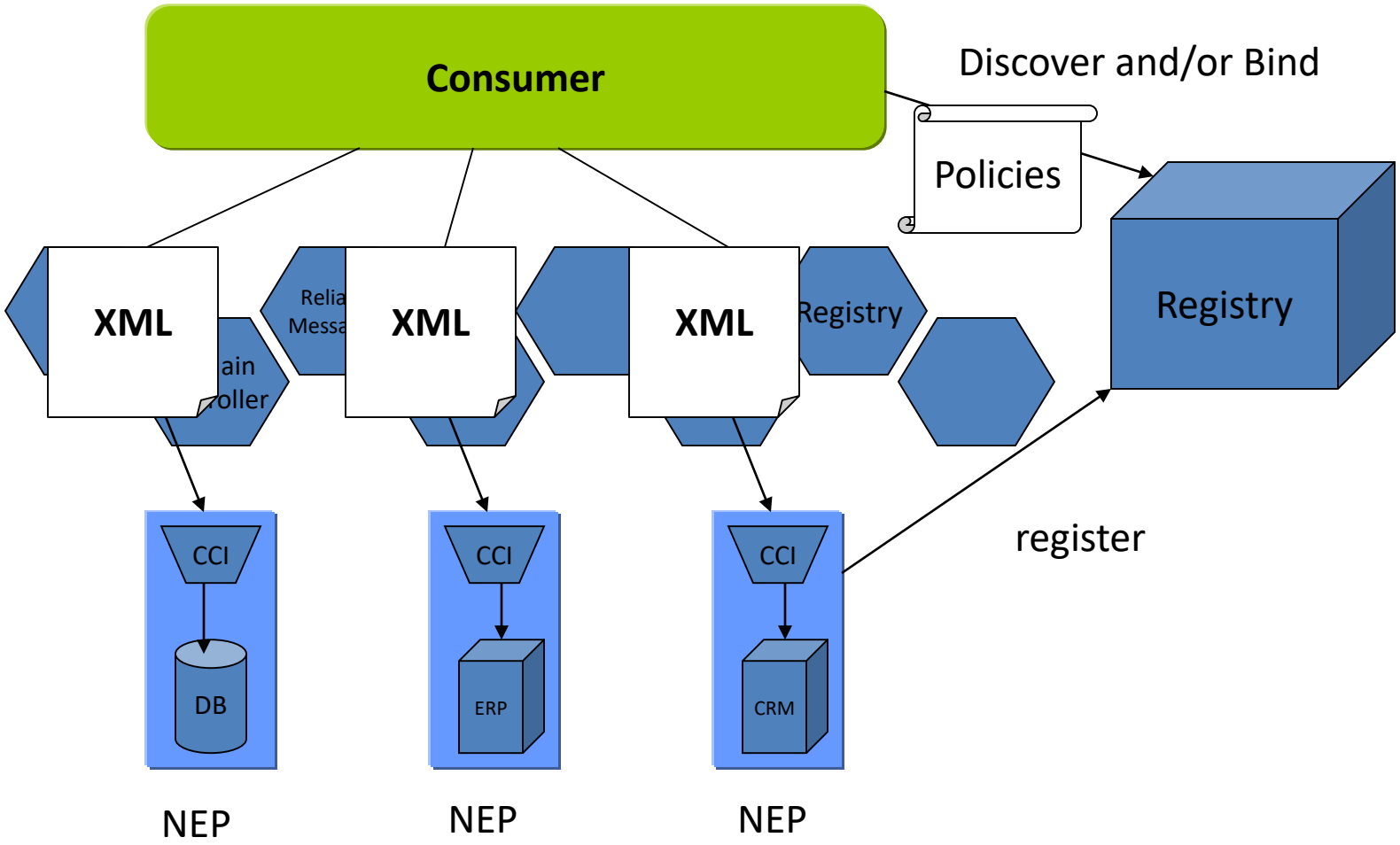
- Transaction
- Security
- Connection pooling
- Naming service
- Scalability and failover
- …

- They become the “Service Fabric”

What about the notion of “Container”? They become Service “Domains”

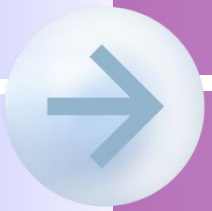
- The notion of “container” shifts to the notion of “Domain Controller”
  - A domain is a collection of web services which share some commonalities like a “secure domain”
  - A container is a domain with one web service
  - Web Services do not always need a container
- Consumers invoke the domain rather than the service directly
- This concept does not exist in any specification…

A Service Fabric can be more than a Bus with a series of Containers / Adapters



Shift To A Service-Oriented Architecture

From	To
• Function oriented • Build to last • Prolonged development cycles	• Coordination oriented • Build to change • Incrementally built and deployed
▪ Application silos ▪ Tightly coupled ▪ Object oriented ▪ Known implementation	▪ Enterprise solutions ▪ Loosely coupled ▪ Message oriented ▪ Abstraction



Source: Microsoft (Modified)

So Migrating to SOA

- Would entail
  - Organizing the business logic in a context independent way
    - Typically, model oriented business logic is wrapped behind (web) services
- Re-implementing the controller with “coordination” technologies
- …The view must be completely re-invented

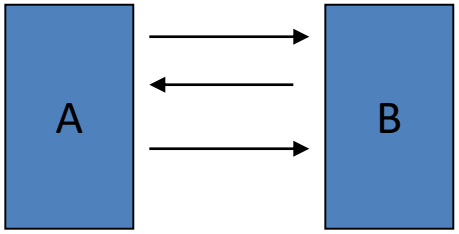
From this point on…

- We will focus on 3 key aspects of the controller
  - The coordination layer
  - Information Entities
  - The relationship between BPM and SOA

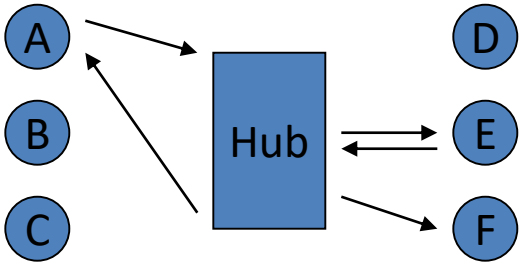
The “Coordination” Layer

- Many, many, many overlapping concepts not layered, not architected
  - Composition
  - Orchestration
  - Choreography
  - Collaboration
  - …
- What is the relationship between these concepts?

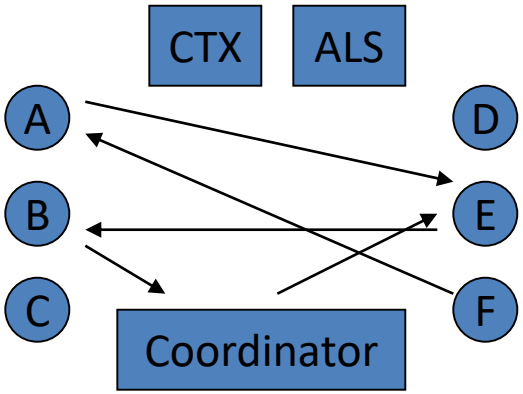
What are the Coordination Topologies?



Binary relationship
- Context and Activity are most often implicit
- Self-coordination



Hub and Spoke relationship
- Context and Activity are handled by the hub
- Coordination is handled by the hub exclusively



Multi party peer-to-peer relationship
- Context and Activity are explicit
- Context, ALS and Coordination are handled by the fabric

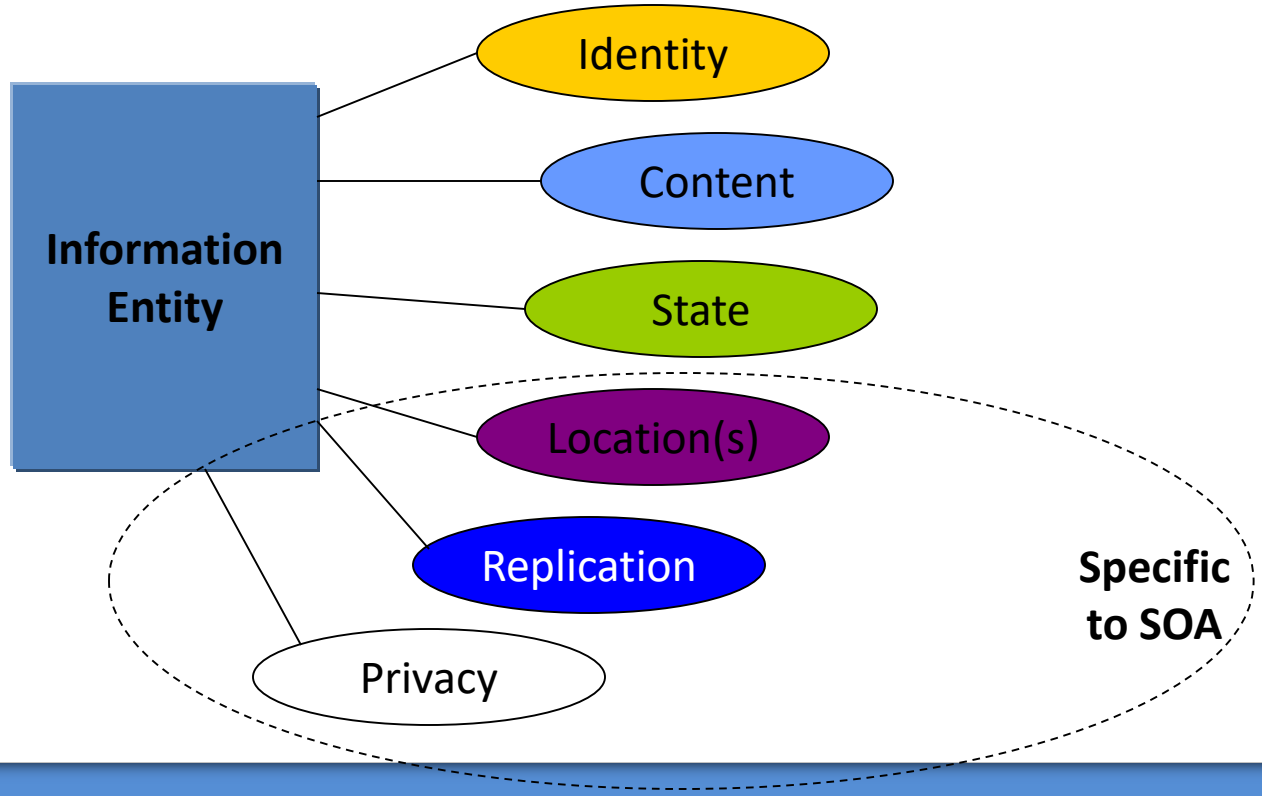
Information Entity in SOA

- “at the heart of SOA services is a very complex problem: with distributed applications comes the need for distributed data sharing”
  - Identification and equivalence
  - authentication
  - Authorization and privacy
  - mediation
  - synchronization

Source: The Dataweb: An Introduction to XDI, Drummond Reed et al.

Information Entities in SOA

- Several dimensions appear when managing an Information Aggregate in a SOA

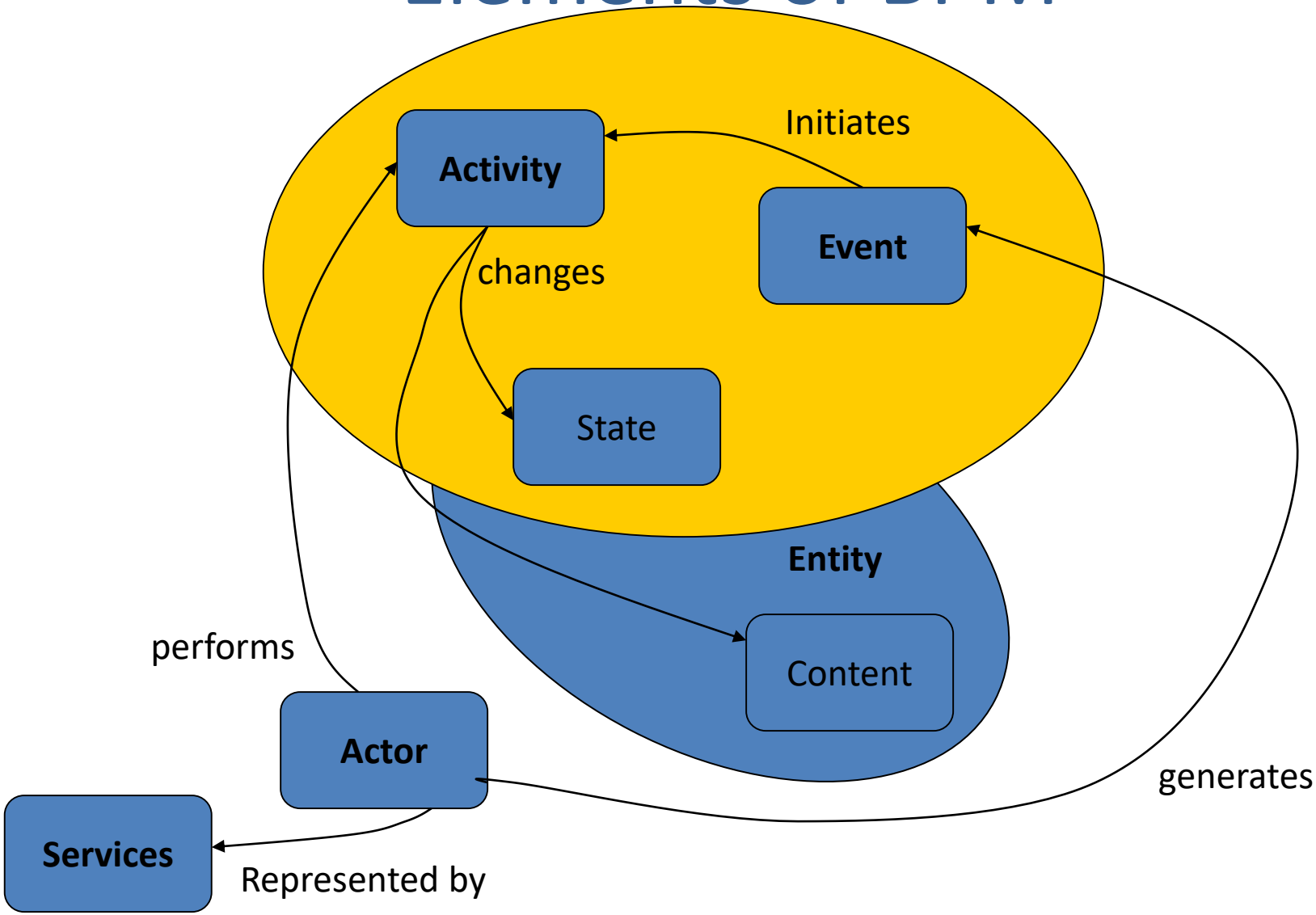


SOA and BPM

- SOA is about constructing software components that can be reused in context unknown at design time
  - Composition versus Extension (OO)
- BPM is about being able to precisely model and possibly change the context in which enterprise components are used

- But how the two meet?

Elements of BPM



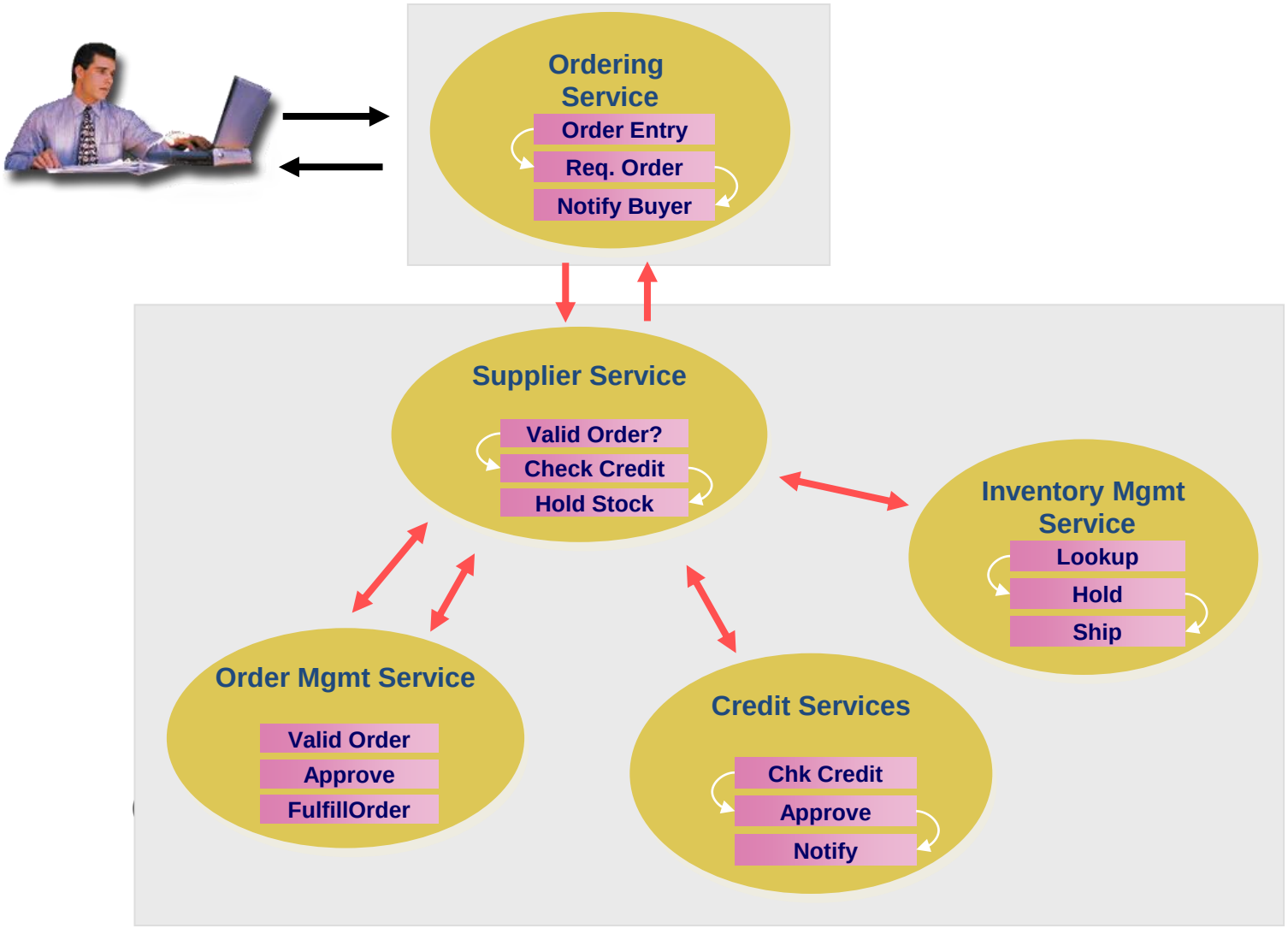
Bringing BPM and SOA together

- The foundation is becoming sound with strong theoretical support
  - A big piece still missing: “state”
  - Shift from Orchestration to Business Process Definition
- Once the foundation is in place we should see Domain Specific Languages (DSL) appear that are a lot closer to the way the users (not the programmers) think about a Business Process

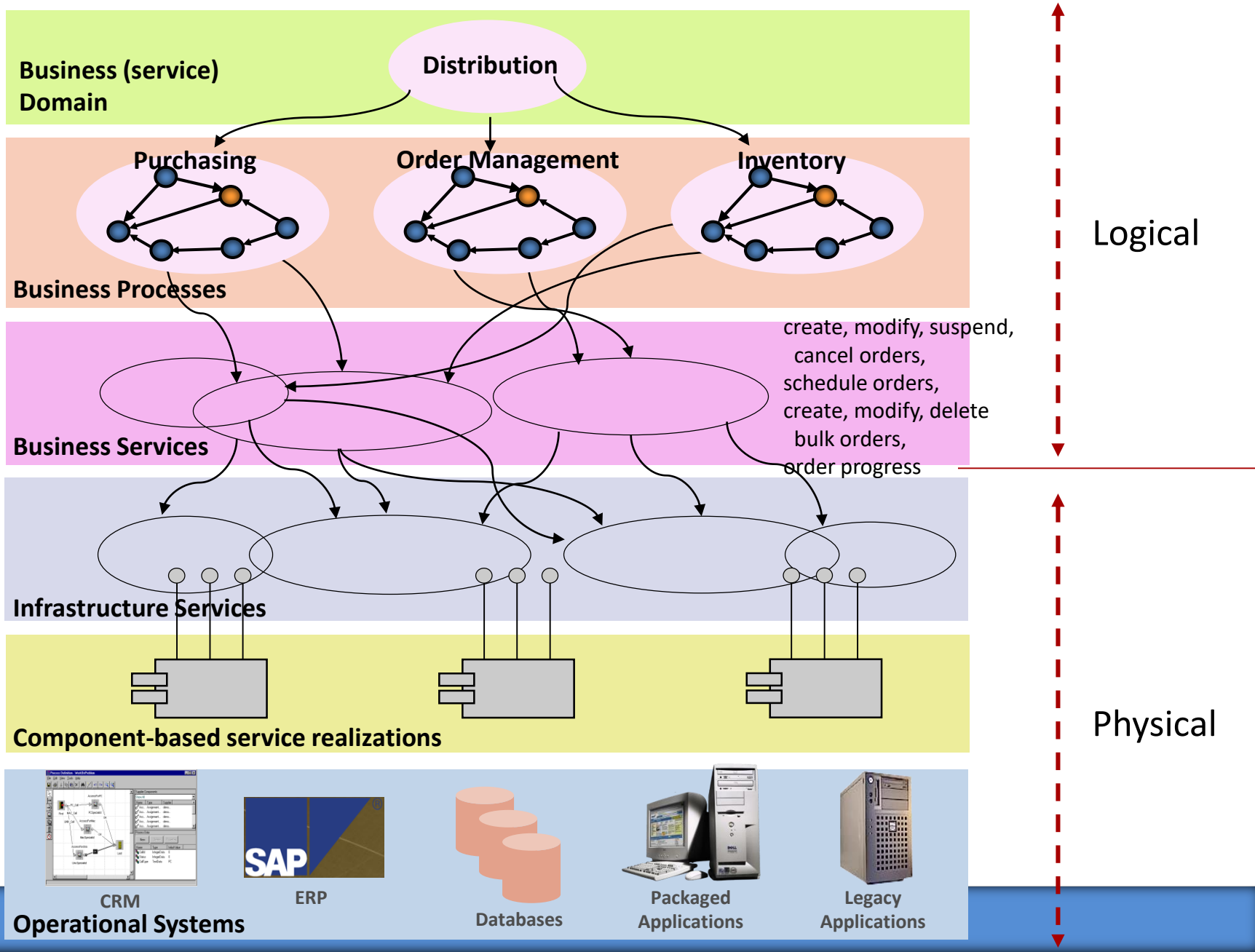
Service Orientation is a New Computing Paradigm

- Not as a new name for
  - API
  - Component
- A genuine set of concepts with which we can construct new kinds of software
  - This is as significant if not more than Object Orientation
- In particular SO forces us to think about enabling the same piece of code to be leveraged
  - by large numbers of consumers
  - in unforeseen context

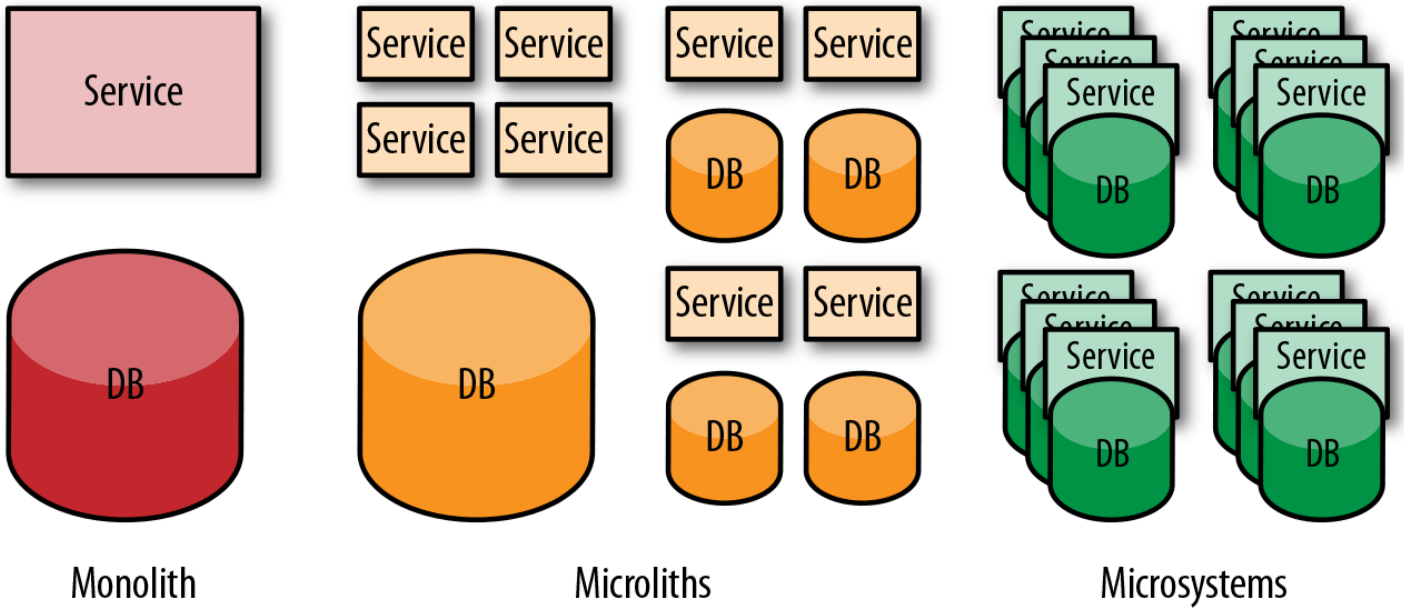
Composite Services



Towards an SOA Reference Architecture



The Evolution of Scalable Microservices



Essential Traits of an Individual Microservice

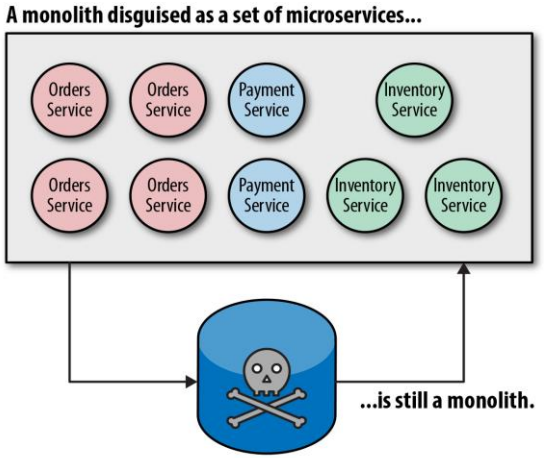
Isolate All the Things - Isolation between services makes it natural to adopt Continuous Delivery (CD). This makes it possible for you to safely deploy applications and roll out and revert changes incrementally, service by service.

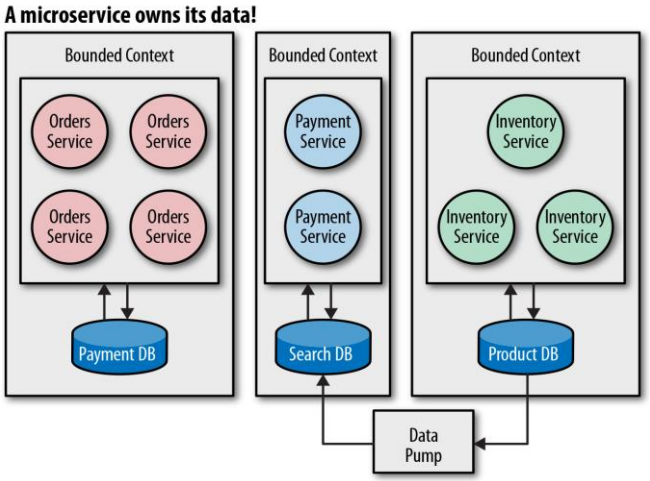
Act Autonomously - Isolation is a prerequisite for *autonomy*. Only when services are isolated can they be fully autonomous and make decisions independently, act independently, and cooperate and coordinate with others to solve problems.

Single Responsibility - This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.

Own Your State, Exclusively - It means that data can be strongly consistent only *within* each service but never *between* services, for which we need to rely on eventual consistency and abandon transactional semantics.

A monolith disguised as a set of microservices is still a monolith

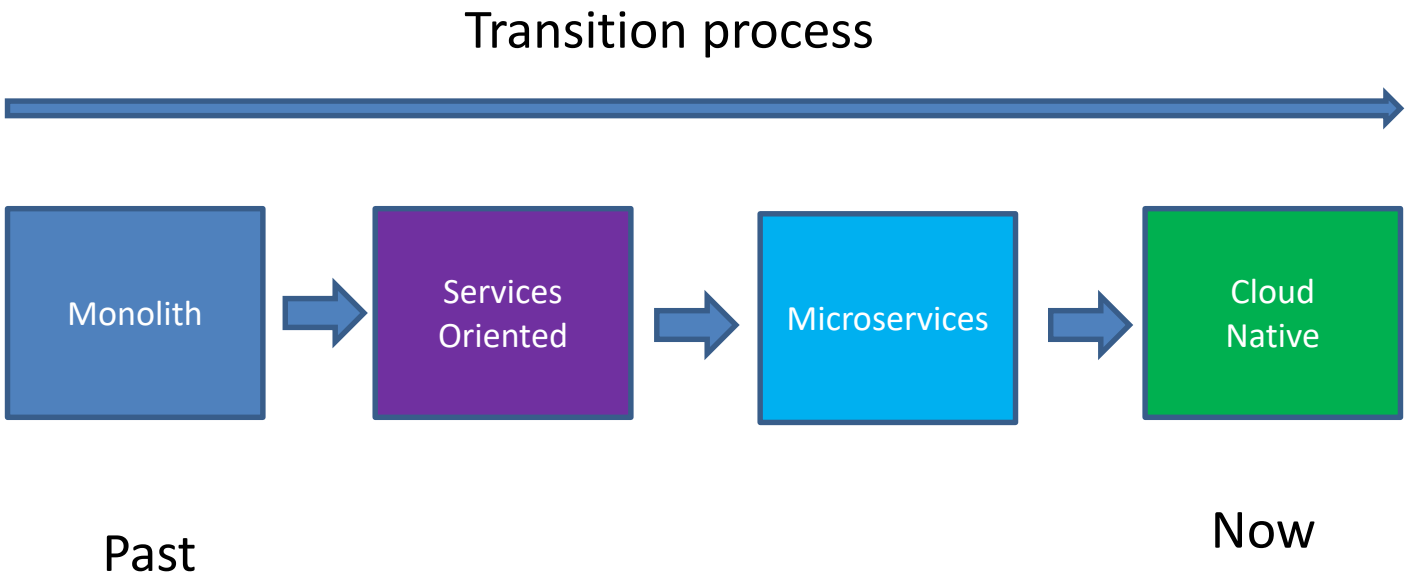




What can be considered “micro”?

How many lines of code can it be and still be a microservice? These are the wrong questions. Instead, **“micro” should refer to scope of responsibility**, and the guiding principle here is the Unix philosophy of SRP: let it do one thing, and do it well.

Application Architecture



Cloud Computing Properties

Cloud is a **business model of services offering** with following properties:

1. access via network,
2. on-demand self-service,
3. measured service (pay-per-use),
4. resource pooling and
5. rapid elasticity

NIST cloud definition

Cloud Native

Applications

A cloud-native application is an application that has been designed and implemented to run on a Platform-as-a-Service installation and to embrace horizontal elastic scaling

Platforms

A Cloud Native Platforms support distributed cloud applications. They also called, enterprise PaaS, private PaaS.

Cloud native platfoms are essentially the next generation of middleware that are built to deliver all the functionality needed to implement distributed applications in an enterprise setting.

Cloud Native

Applications

Essence of cloud native Apps:

- Ability to rapidly deliver software that simply does not fail.
- Applications are able to dynamically scale to deal with volumes of data previously unheard of
- Software that can handle massive load, remain responsive, and change as rapidly as the market.

SUPPORTS

Platforms

A Cloud Native Platforms support distributed cloud applications. They also called, enterprise PaaS, private PaaS.

Cloud native platfoms are essentially the next generation of middleware that are built to deliver all the functionality needed to implement distributed applications in an enterprise setting.

Cloud Native

Applications

Essence of cloud native Apps:

- Ability to rapidly deliver software that simply does not fail.
- Applications are able to dynamically scale to deal with volumes of data previously unheard of
- Software that can handle massive load, remain responsive, and change as rapidly as the market.

SUPPORTS

Platforms

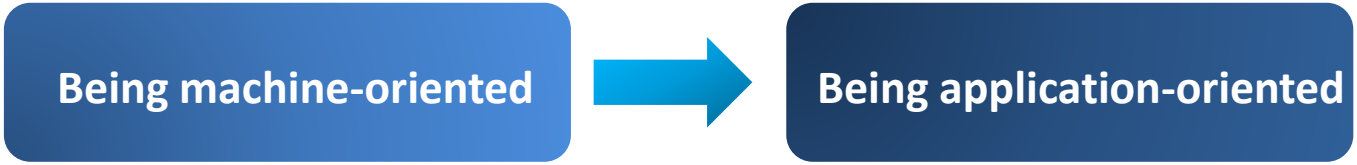
Building on the foundations of :

- containers and container orchestration
- logging, monitoring, auditing, security, policies,
- standard container image repo
- role-based access,
- Infrastructure tools abstraction e.g. virtualization

Native cloud Platforms supports Cloud Native and legacy Applications

Cloud Native Platform as Result of Evolution

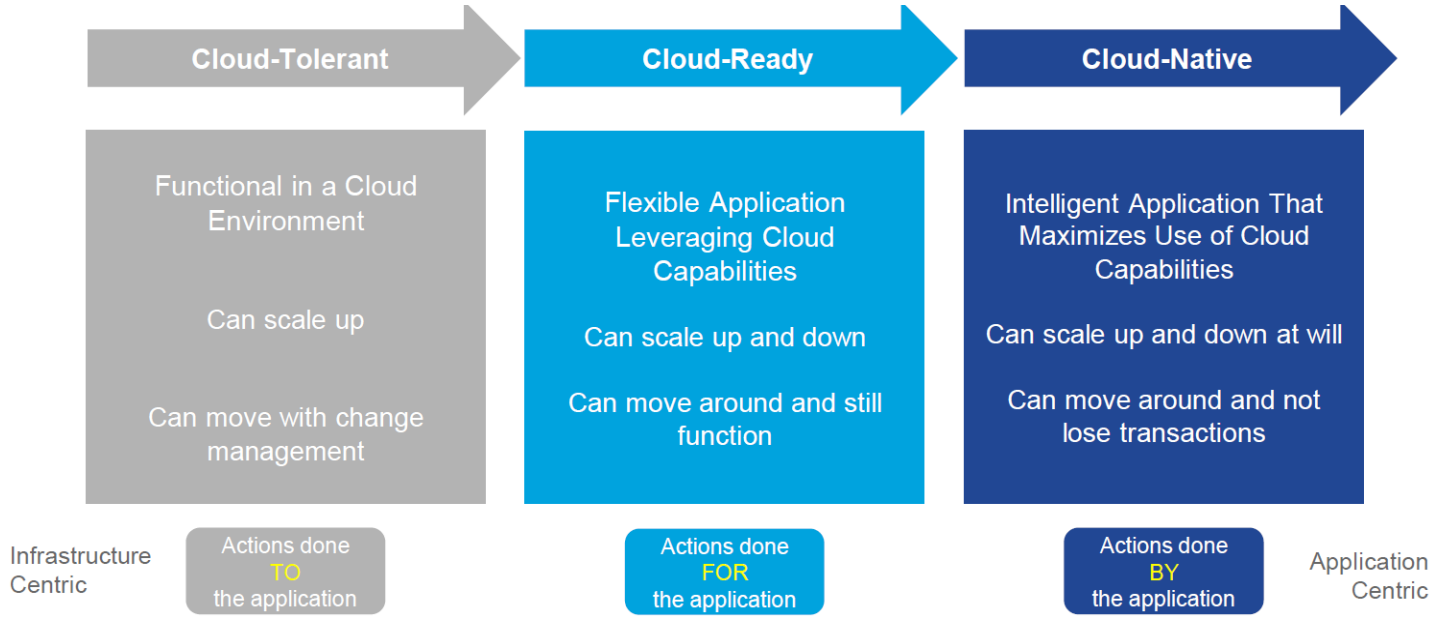
Container as a unit of Management



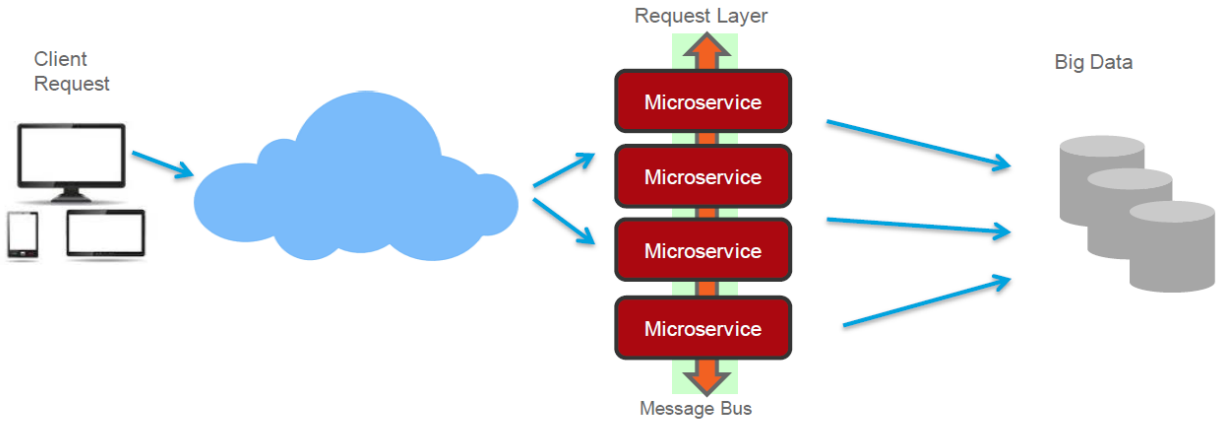
Containerization

e.g. Load balancers don’t balance traffic across machines; they balance across application instances

App Transformation: the road to cloud native



Microservices Application Architecture



Ever Changing Type and Number of Users

Large Number of Simple Components

Many Sources, Types, and Locations of Data

END