Enterprise Functionalities: 2

- Data management
  - Ensuring integrity, e.g., entity and referential integrity (richer than storage-level integrity)
  - Enabling access under various kinds of problems, e.g., network partitions
  - Supporting recovery, e.g., application, operating system, or hardware failures

Enterprise Functionalities: 3

Bases for choosing the above three-way partitioning as opposed to some other

- Size of implementations
- Organizational structure: who owns what and who needs what
- Staff skill sets
  - User Interface: usability and design
  - Programming
  - Database
  - Policy tools
- Products available in the marketplace
One-Tier and Two-Tier Architectures

- One tier: monolithic systems; intertwined in the code base
  - Historically the first
  - Common in legacy systems
  - Difficult to maintain and scale up
- Two-tier: separate data from presentation and business logic
  - Classical client-server (or fat client) approaches
  - Mix presentation with business rules
  - Change management

Three-Tier Architecture: 1

- Presentation tier or frontend
  - Provides a view to user and takes inputs
  - Invokes the same business logic regardless of interface modalities: voice, Web, small screen, ... 
- Business logic tier or middle tier
  - Specifies application logic
  - Specifies business rules
    - Application-level policies
    - Inspectable
    - Modifiable
Three-Tier Architecture: 2

- Data tier or backend
  - Stores and provides access to data
  - Protects integrity of data via concurrency control and recovery

Multitier Architecture

Also known as n-tier (sometimes treated synonymously with three-tier)

- Best understood as a componentized version of three-tier architecture where
  - Functionality is assembled from parts, which may themselves be assembled
  - Supports greater reuse and enables greater dynamism
  - But only if the semantics is characterized properly
- Famous subclass: service-oriented architecture
Architectural Tiers Evaluated

The tiers reflect logical, not physical partitioning

- The more open the architecture the greater the decoupling among components
  - Improves development through reuse
  - Enables composition of components
  - Facilitates management of resources, including scaling up
  - Sets boundaries for organizational control
- In a narrow sense, having more moving parts can complicate management
- But improved architecture facilitates management through divide and conquer

XML-Based Information System

Let’s place XML in a multitier architecture
How About Database Triggers?

- **Pros**: essential for achieving high efficiency
  - Reduce network load and materializing and serializing costs
  - Leave the heavy logic in the database, under the care of the DBA

- **Cons**: rarely port well across vendors
  - Difficult to introduce and manage because of DBA control
  - Business rules are context-sensitive and cannot always be applied regardless of how the data is modified

Implementational Architecture: 1

Centered on a Web server that
- Supports HTTP operations
- Usually multithreaded
Implementational Architecture: 2

Application server
- Mediates interactions between browsers and backend databases: runs computations, invoking DB transactions as needed
- Provides a venue for the business logic
- Different approaches (CGI, server scripts, servlets, Enterprise JavaBeans)

Implementational Architecture: 3

Database Servers
- Hold the data, ensuring its integrity
- Manage transactions, providing
  - Concurrency control
  - Recovery

Transaction monitors can manage transactions across database systems, but within the same administrative domain
Data Center Architecture

- Demilitarized zone (DMZ)
  - External router
  - Load balancer
- Firewall: only the router can contact the internal network
  - Internal network
  - Web servers
  - Application servers
  - Database servers

Application Servers

Architectural abstraction separating business logic from infrastructure

- Load balancing
- Distribution and clustering
- Availability
- Logging and auditing
- Connection (and resource) pooling
- Security

Separate programming from administration roles
Middleware: 1

Components with routine, reusable functionality
- Abstracted from the application logic or the backend systems
- Any functionality that is being repeated is a candidate for being factored out into middleware
- Enables plugging in endpoints (e.g., clients and servers) according to the stated protocols
- Often preloaded on an application server
- Simplify programmer’s task and enable refinements and optimizations

Middleware: 2

Software components that implement architectural interfaces, e.g., transaction, persistence, ...

- **Explicit:**
  - Invoke specialized APIs explicitly
  - Difficult to create, maintain, port
- **Implicit:**
  - Container invokes the appropriate APIs
  - Based on declarative specifications
  - Relies on request interceptions or reflection
Containers

- Architectural abstraction geared for hosting business components
  - Remote method invocation
  - Threading
  - Messaging
  - Transactions
- Implementations for JEE and .NET

Message-Oriented Middleware: 1

- **Queues**: point to point, support posting and reading messages
- **Topics**: logical multicasts, support publishing and subscribing to application-specific topics; thus more flexible than queues
- Can offer reliability guarantees of delivery or failure notification to sender
  - Analogous to store and forward networks
- Some messages correspond to event notifications
Message-Oriented Middleware: 2

- Varies in reliability guarantees
- Usually implemented over databases
- Can be used through an invocation-based interface (i.e., registered callbacks)

Message-Driven Beans

A standardized receiver for messages
- Clients can’t invoke them directly; must send messages to them
- No need for specialized interfaces, such as home, remote, . . .
- Easy interface to implement: mainly onMessage(), but limited message typing
- Stateless: thus no conversations
Methods for Message-Driven Beans

- **onMessage()**: define what actions to take when a message arrives on the destination this bean is watching

Peer-to-Peer Computing

- **Symmetric client-server**: (callbacks) each party can be the client of the other
- **Asynchrony**: while the request-response paradigm corresponds to pull, asynchronous communication corresponds to push
  - Generally to place the entire intelligence on the server (pushing) side
- **Federation of equals**: (business partners) when the participants can enact the protocols they like