**Creational**

SimpleFactory

Factory Pattern

AbstractFactory

Singleton

Builder

Prototype

**Structural**

Adapter

Façade

Proxy

Bridge

Composite

Decorator  
Flyweight

**Behavioral**

Command

Observer

Chain of responsibility

Visitor

Strategy

State

Template

Iterator

Mediator

**Creational Patterns**

**SimpleFactory**

**IDatabase => SqlDatabase/OracleDatabase**

DatabaseFactory

Public IDatabase GetDatabase(string type)

{

If (type=”sql” )

{

return new SqlDatabase();

}

else

{

return new OracleDatabase();

}

}

**FactoryMethod**

**IDatabase => SqlDatabase/OracleDatabase**

**IDatabaseFactory =>SqlFactory/OracleFactory**

Public class SqlFactory

{

Public IDatabase GetDatabase()

{

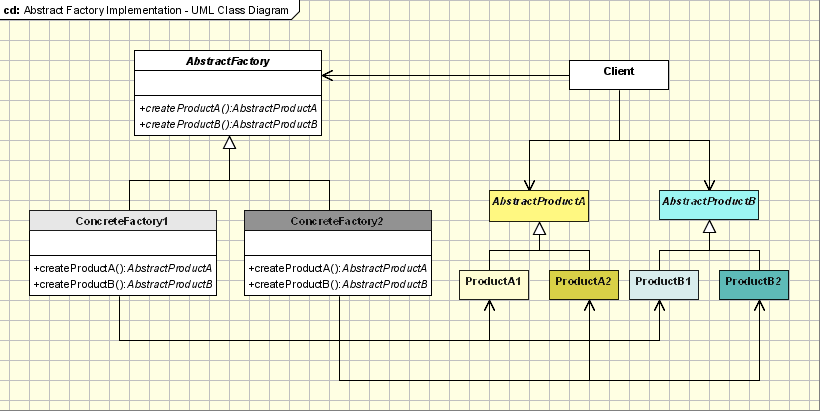
return new SqlDatabase();

}

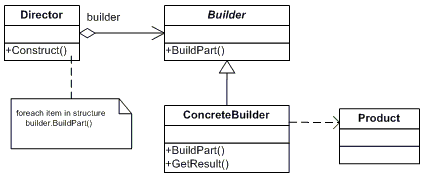
}

**Singleton**

**Abstract Factory**



**Builder**



VehicleBuilder -> BikeBuilder /CarBuilder

-buildBody()

-buildEngine()

-buildParts()

Class Director

{

Public Filebuilder fileBuilder;

Public void Construct(Filebuilder filebuilder)

{

fileBuilder= filebuilder;

}

Public void Execute()

{

fileBuilder. ReadFile ()

fileBuilder. ProcessFile ()

fileBuilder. UploadFile ()

fileBuilder. EmailFile ()

}

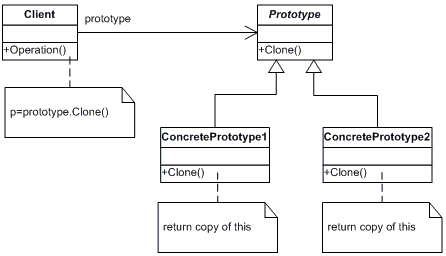
}

FileBuilder (ReadFile,ProcessFile,UploadFile,EmailFile)=> ExcelBuilder & Text Builder

Director

**Prototype**

Used to create a clone of an object using this pattern

****

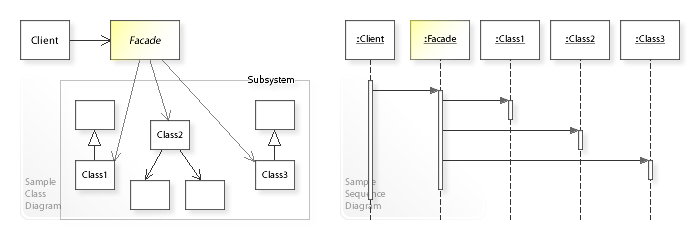
**Structural Patterns**

**Adapter**

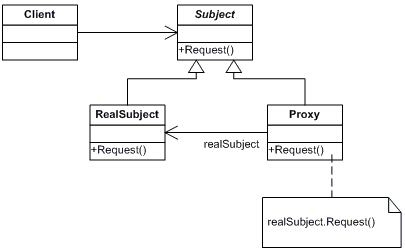
Used to call a third party external component.



**Façade**

****

**Proxy**



**Behavioural Patterns**

**Command**



**Invoker =>** SetCommand(Command command),executeCommand(){ command.execute()}

**Command** => ConcreteComand1,ConcreteCommand2

Public class ConcreteCommand1: Command

{

ConcreteCommand1 (Receiver receiver)

{

this.Receiver=receiver;

}

Public override Execute()

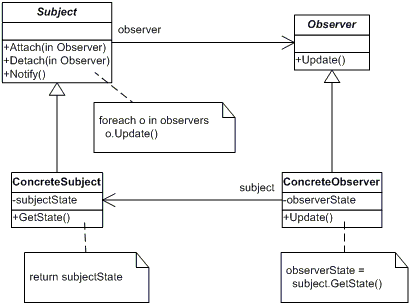
{

this.Receiver.Execute()

}

}

**Observer**



**SOLID Principles**

1. Single Responsibility:- Class should have one and only one responsibility and and only one reason to change

Customer class doing getCustomer() & writing to Log logic.

Writing to Log should go to Logger class

1. Open/Close :- One class is closed once it is done and should be able to extend its features without modifying it.

Customer => PlatinumCustomer, GoldCustomer, SilverCustomer

Discount () logic has different implementation with if / else logic. A new customer comes, we should not modify this logic again.

So create an interface and different implementation by creating separate implementation classes with its on Discount implementation.

1. Liskov Substitution **:- Derived classes should be substitutable for their base classes**

Customer

-Add

-GetDiscount

Enquiry

-Discount

1. Interface Segregation:-Create fine grained interfaces which are client specific

Clients should not be forced to depend upon interfaces that they don't use.

Customer : ICustomer

-Add

-Print

SavingsCustomer : wanted delete also

-delete

Create : ISpecialCustomer : ICustomer

{

Delete}

1. Dependency Inversion:-

**DRY (Don’t Repeat Yours)**

* Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
* Do not have repetitive code, repetitive task, data normalization t avoid redundancy.

**KISS (Keep it Simple Stupid)**

* Most of the systems work best if they kept simple rather than made complicated.

**YAGNI ( You are not gonna need it )**

* Principle of [extreme programming](https://en.wikipedia.org/wiki/Extreme_programming) (XP) that states a [programmer](https://en.wikipedia.org/wiki/Programmer) should not add functionality until deemed necessary.

**12 Factor App Principle**

1. **Codebase : -** One codebase and multiple deploys. This means you must not have various codebase for various versions. Branches is okay, different repos are not.
2. **Dependencies:**- Dependencies should be through packages or easy installation.
3. **Configuration :** Strict separation of config from code.  Never commit your environment-specific configuration (most importantly: password) in the source code repo
4. **Backing Services**:- A backing service is a service that app consume during its operation. Ex:- data store (my sql,Couch), Message queue (Kafka, Rabbit MQ), caching (mem cached)

There also could be third party services like New-Relic , AppDynamics, Splunk, SMTP Services.

The code for a twelve-factor app makes no distinction between local and third party services

and both are resources.

1. **Build,Release,Run** :- Separate build and Run stages.

The twelve-factor app uses strict separation between the build, release, and run stages.

Builds are initiated by apps developers when the new code is deployed.

1. **Processes** :- Execute application as one more more stateless process . with persisted data on a backing service.
2. **PortBinding** :- self contained services to be available for other services by specified ports
3. **Concurrency** :- concurrency by scaling each process
4. **Disposability** :-Fast startup and shut down
5. **Dev/Prod parity** :- All environments should be similar
6. **Logs** :- produce logs as event streams and execution environment to aggregate
7. **Admin Processes** :- admin tasks to be kept in source control and packaged with application.

**4+1 view model**

* **Logical View:-**functionality to end users.

**UML:** block diagram, class diagram,object diagram, state diagram, interaction diagram(eg:- sequence diagram)

* **Physical View**: - components on a physical layer.

**UML:**Deployment diagram.

* **Development view (Implementation view):-** programmers view and represents modules and sub domains. **UML:** package and component diagram.
* **Process View**:- how systems communicate and run time behaviours. Address concurrency, performance, scalability.

**UML:** Activity diagram.

* **Scenarios (use case view):**-small set of use case and scenarios that describe interaction between objects. Used to identify architectural elements.

[https://dzone.com/articles/“41”-view-model-software](https://dzone.com/articles/)

**Architecture Principle**

**ViewPoints:-**

**=========**

**1. Requirements Viewpoint :-**

-Identify stake holders

-Identify allFunctional & Non Functional

- Gather Risks & Assumptions

**tools :** Usecase, User stories

**2. Functional View point:-**

identify functional elements , interfaces and interactions with them

**tools :-** Component Diagram, Activity Diagram

**3. Information Viewpoint**

E-R model, Entity-Relation ship, Document model.

**tools:-** ER Diagram.

**4. Concurrency Viewpoint**

-identifies intra/inter process and system communication.

-ensures integrity

**5. Development Viewpoint**

- Project structure / Module structure

- environment setup

- pocs

-Design principles

- HLD, LLD

- unit tests, performance planning

- CI-CD

**6. Deployment Viewpoint**

- functional elements to hardware devices

**7. Operational Viewpoints**

any operational improvements.

- automate administrative & repetitive & error prone process

- deployment process improvements.

**8. Validation Viewpoints**

identify the stake holders & mechanism to validate the architeture

**Perspective**

**==========**

**1. Security Perpective**

- Aware of Security risks to application

-OWASP

- Identify the security threats

- Authentication

-Identity,Active Directory, SSO, Federated Identity

- role based security ( spring security, identity )

-classify data & sensitive resources

- security & compliance standards

- proper encryption and hashing

- auditing and intrusion monitoring systems

**-**

**2. Performance and Scalaility Perspective**

- Performance SLAs

-identify performance bottlenecks

-performance test plans & tools (Jmeter, UI testing etc)

-caching, clustering,resource pooling

- stress, load, soak and spike testing

- profiling

- plan the hardware to meet the requirement

- cloud system and services analysis

- horizontal scaling

- threading,parallel programming, NOSQL,MapReduce

**3. Availability and Resilience Perspective**

- SLA for uptime, allowed downtime

- monitoring & alerting to monitor availability SLAs

- Design for fault tolerance and appropriate fail-over mechanisms.

**4. Regulation Perspective**

- regulatory requirements

-ITAR, SOX, EU Data Retension,HIPPA

**5. Evolution Perspective**

-system aspects most likely to evolve.

-Assess impacts to system.

- evaluate need of desinging for change.(SOLID)

**-**

**6. Resource and Cost Perspective**

- estimate effort, duration and cost for technical delivery

- estimation techniques(story points,function points)

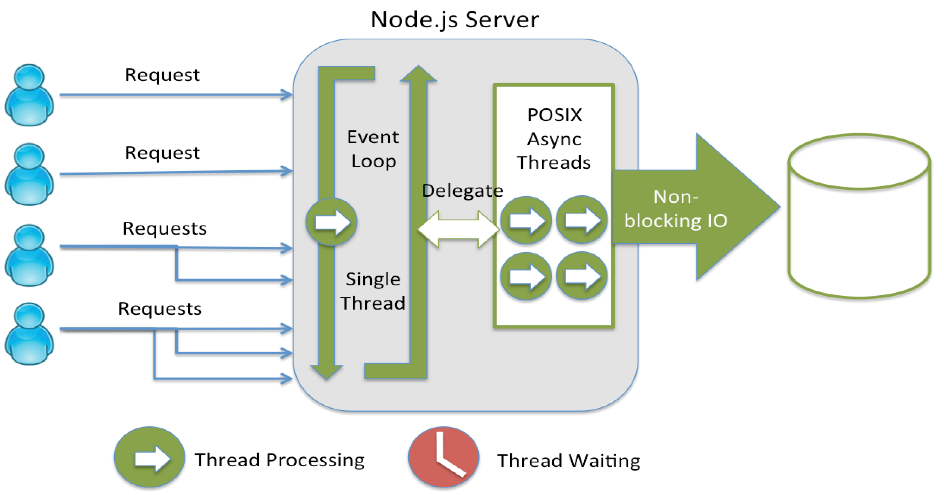
- POC for cost estimation,

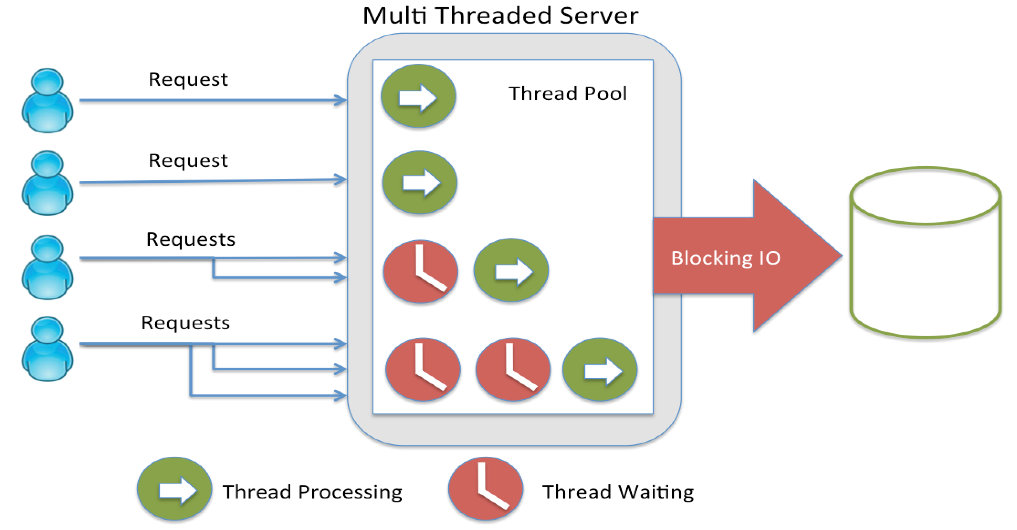
**Consideration while converting Requirements to Design**

**Performance Considerations while design**

**NodeJS vs RestfulAPIs ///Why WebAPI/Spring boot vs NodeJS??**

NodeJs provides single threaded asynchronous non-blocking IO. But can only utilize one Core.

[](https://camo.githubusercontent.com/87a6cdbb3801eb4d1a5a712a70739da0fea31844/68747470733a2f2f692e696d6775722e636f6d2f6f7579504b6b662e706e67) -



**Nodejs express**

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications.

**NodeJs pros and cons**

**Pros=>**

* 1. Faster for small applications with async single thread model
  2. Advantages if both client and server works on java based framework. Better serialization etc.
  3. Good for connected application like chat, games,socket connection,web application framework etc
  4. Anything that is not CPU intensive
  5. Easy packaging

**Cons=>**

* 1. Other framework provide better features and easier development.
  2. CPU intensive operation runs very slow.
  3. Callback hell
  4. Code is not typesafe and compiled.
  5. Framework is changing fast without backward compatibility
  6. Requires clustered design to take advantages of multicore.

**NodeJs vs WebAPI**

* + 1. Everything works out of the box in webapi. But tailoring the default behavior needs more effort.
    2. WebAPI is Great for large application but bloated for simple tasks.
    3. Node.js is created for fast request handling without heavy computations. Requiring heavy computations Node.js will certainly loose to ASP.NET.
    4. JavaScript is also more ubiquitous than .NET
    5. WebAPI is more type safe.(strongly typed)
    6. Nodejs is more portable. .NET Core is also portable.
    7. WebAPI is more readable.
    8. Nodejs is more testable.
    9. Max Memory supported is 1-1.7Gb for nodejs in a 64 bit system. 2Gb in ASP.NET
    10. Nodejs is good for IO load and MVC is better for CPU load.

**Performance comparison of WebAPI vs NodeJS**

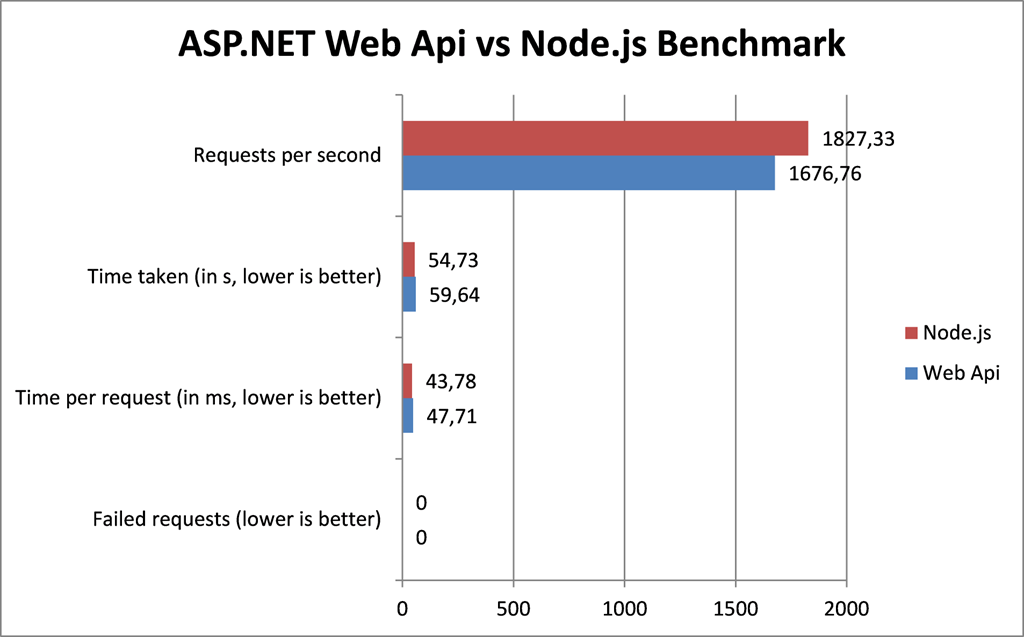
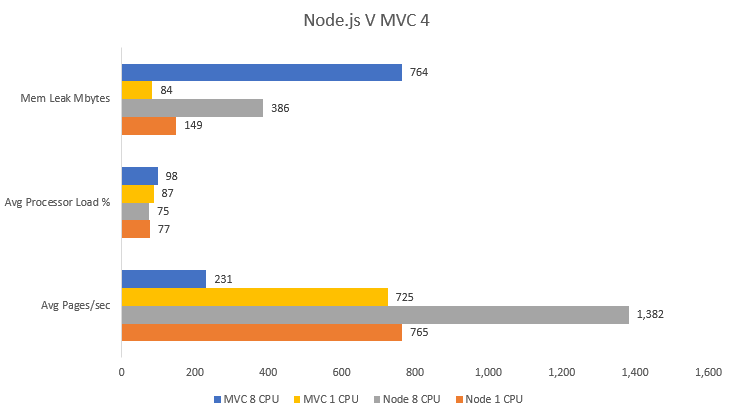
Here is a chart benchmarking ASP.NET Web API and Node.js for asynchronously reading POST body, 2012, [source](http://mikaelkoskinen.net/post/asp-net-web-api-vs-node-js-benchmark-take-2).[](https://camo.githubusercontent.com/3a048230c780e41940b29670fde73b4043030578/68747470733a2f2f692e696d6775722e636f6d2f4e356b3058695a2e706e67)

Chart below compares performance of Node.js and ASP.NET MVC 4 for asynchronously reading json file from filesystem, 2012, [source](http://rarcher.azurewebsites.net/Post/PostContent/19).

[](https://camo.githubusercontent.com/a2e9cf0d224a57ecdac5d435dc8f35a3759ac5e4/68747470733a2f2f692e696d6775722e636f6d2f4e7042314638422e706e67)

**Security consideration for Architects**

**Things to be considered (Concerns)**

* Principals: - Identify a user or access.
* Policies , Standards and Compliances :- Leads to security rules
* Threats / Vulnerabilities
* Resources & Data classifications: - sensitive part of the system.
* Accountability
* Availability
* Confidentiality/Integrity
* Recovery/Disaster recovery
* Risk

**Security Implementation:**

* Authentication
* Authorization
* Asset protection
* Audit (Logs and Alerts)
* Assurance
* Administration
* Risk Management

**Web Application Security:**

* Broken Authentication
* Broken Access controls
* Injection
* XSS
* XSRF
* Information leakage

**Design Guidelines for Application Performance**

**High Level**

* **Set performance objectives**
* **Validate architecture and design** early. Use POCs
  + Authentication and authorization
  + Exception management
  + Data access strategies
  + State management and caching
* **Validate Deployment Architecture** :- deployment topology, load balancing, network bandwidth,
* **Cut the dead wood** :- Remove un necessary process/flow
* **Tune end to end performance** :-a single bottleneck in a subsystem within your application can affect overall application performance
* **Plan and Measure performance thorugh out life cycle** :- As it can degrade.
* **Use a Layered design**
* **Stateless components (minimize sessions,**
* **Avoid Server affinity**

**Low Level**

* **Design fine grained Services**
* **Build stateless components**
* **Reduce round trips**
* **Acquire late and release early**
* **Processing should be closer to the resource it needs**
* **Pool shared resources**
* **Progressive processing**
* **Process independent tasks concurrently**
* **Consider asynchronous communication, fire and forget**
* **Consider message queues.**
* **Implement caching (cache transformed data)**

|  |  |
| --- | --- |
| **Performance profile category** | **Guidelines** |
| Coupling and Cohesion | Design for loose coupling.  Design for high cohesion.  Partition application functionality into logical layers.  Use early binding where possible.  Evaluate resource affinity. |
| Communication | Choose the appropriate remote communication mechanism.  Design chunky interfaces.  Consider how to pass data between layers.  Minimize the amount of data sent across the wire.  Batch work to reduce calls over the network.  Reduce transitions across boundaries.  Consider asynchronous communication.  Consider message queuing.  Consider a "fire and forget" invocation model. |
| Concurrency | Reduce contention by minimizing lock times.  Balance between coarse-grained and fine-grained locks.  Choose an appropriate transaction isolation level.  Avoid long-running atomic transactions. |
| Resource Management | Treat threads as a shared resource.  Pool shared or scarce resources.  Acquire late, release early.  Consider efficient object creation and destruction.  Consider resource throttling. |
| Caching | Decide where to cache data.  Decide what data to cache.  Decide the expiration policy and scavenging mechanism.  Decide how to load the cache data.  Avoid distributed coherent caches. |
| State Management | Evaluate stateful versus stateless design.  Consider your state store options.  Minimize session data.  Free session resources as soon as possible.  Avoid accessing session variables from business logic. |
| Data Structures/Algorithms | Choose an appropriate data structure.  Pre-assign size for large dynamic growth data types.  Use value and reference types appropriately. |

<https://docs.oracle.com/cd/B19306_01/server.102/b14211/design.htm#g34949>

**ACID**

**Replication in SQL Server**

**NOSQL Vs RDBMS**

RDBMS follow [**ACID**](http://en.wikipedia.org/wiki/ACID) (**A**tomicity, **C**onsistency, **I**solation, **D**urability).

**CAP Theorem /** [**Eric Brewer**](https://en.wikipedia.org/wiki/Eric_Brewer_(scientist)) **(Brewer's theorem)**

NoSQL follow **CAP (C**onsistency, Availability, Partition-tolerance):-

* CAP-Theorem states that only two of the three different aspects of scaling out are can be achieved fully at the same time.
* Many NOSQL database compromised on Consistency and achieved Availability and partitioning. This resulted in **BASE** transaction ( Basically Available, Soft-state, Eventually consistent). Theoretical foundation of BASE does not exist yet (it seems more of an informal approach

**Note** :- ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux

A BASE system gives up on consistency.

* **Basically available** indicates that the system *does* guarantee availability, in terms of the CAP theorem.
* **Soft state** indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
* **Eventual consistency** indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

**"design pattern" for building correct systems that (in a way) offer both CAP and BASE qualities**

* Process reads from database or cache if needed for availability.
* Optimistic locking with versioning or other mechanisms.
* Updates by clients (eg:- order request) are **queued** for execution.
* Queued updates are done when the number of partitions are less. The queued updates are applied as a cluster-wide distributed transaction across all replicas.

More refined way is to do the **quorum-based** or smart way of replication.

* Version information in the update is used to validate.
* The results are sent to client asynchronously through mail , message queue or any other async way.

<http://guysblogspot.blogspot.in/2008/09/cap-solution-proving-brewer-wrong.html>

**Solutions** : Expiration-based cahing, Quorum/majority alogorithms ,Two-Phase commit

Q : Can we achieve consistency with a replicated RDBMS?

<http://www.julianbrowne.com/article/brewers-cap-theorem>

<https://www.infoq.com/presentations/availability-consistency>

<https://queue.acm.org/detail.cfm?id=1394128>

**Architecture Diagrams/ Representations**

**High level Architecture:-**

**UML :**

**Structural Diagrams:**

-Component Diagrams => Logical Architecture

**-**Deployment Diagram => Physical architecture

**-**Class Diagram => Logical Architecture ( with details of High level class interactions)

**Behavioral Diagrams:**

-Usecase diagram => High level Usecases

-Sequence diagram => High level sequence

-Activity Diagram => High level sequence

- Collaboration diagram

**Others :-**

-Block diagram

-Flowchart

**Low Level Architecture:-**

**UML :**

**Structural Diagrams:**

-Class diagam

- object diagram

**Behavioral Diagrams:**

**-**  Use case diagrams

- Sequence diagram

- State chart diagram: - event driven state changes of a system. State change of

a class/Interface etc.

- Collaboration diagram:- structural organization of a system and the messages

Sent/received. Similar to Sequence diagram but used for capturing objects and

Interactions or messages.

**Note :** Marchitecture diagrams ( marketing and Architecture diagrams)

**Layered design approach**

<https://medium.com/slalom-engineering/black-slope-a-net-core-reference-architecture-5a7bf8695fc8>

**Reference:**

<https://www.tutorialspoint.com/uml/uml_basic_notations.htm>

<https://www.edrawsoft.com/architecture-diagram.php>

**Transaction**

http://www.dotnetfunda.com/articles/show/3087/how-to-work-with-transactions-in-aspnet-mvc

- Are you calling an external Payment gateway system that deducts money, but could fail on your callback?

- Do you have multiple sign-on mechanisms (like email or OAuth)?

- Are you calling third-party SaaS products that don’t have a rollback option?

- Are you leveraging Cloud APIs and storage buckets which don’t respect your transaction boundary?

- Do you have workflows spanning multiple request lifecycles to the same service?

**Transaction in Web API**

<http://geogig.org/docs/interaction/web-api-transactions.html>

**2 phase commit:**

<https://dzone.com/articles/xa-transactions-2-phase-commit>

<http://www.jguru.com/faq/view.jsp?EID=20929>

**SAGA**

<https://www.youtube.com/watch?v=YPbGW3Fnmbc>

**Distributed Transaction**

<https://code.msdn.microsoft.com/Distributed-Transactions-c7e0a8c2>

<https://code.msdn.microsoft.com/Distributed-Transactions-c7e0a8c2>

**-2phase commit, Sagas**

**Reference Architecture**

<http://mowebs.net/architecture/billing-system-architecture/attachment/billing-system-architecture-elegant-conducting-ecommerce/>

<https://www.google.com/imgres?imgurl=https%3A%2F%2Fimage.slidesharecdn.com%2Fthesisdefensepresentation-pluggableecommerceplatform-mohammadanggastaparamartha-finalversionv3-140926232640-phpapp01%2F95%2Fdesign-and-instantiation-of-reference-architecture-for-pluggable-service-platform-in-ecommerce-12-638.jpg%3Fcb%3D1411774103&imgrefurl=https%3A%2F%2Fwww.slideshare.net%2Fslidesharemania%2Fdesign-of-reference-architecture-for-pluggable-ecommerce-platform&docid=WbHOk9XLzwOM6M&tbnid=M_2CQEK8gtlw2M%3A&vet=10ahUKEwjb-uCektXaAhUDN48KHX-_B4IQMwimAigBMAE..i&w=638&h=479&bih=649&biw=1366&q=ecommerce%20reference%20architecture&ved=0ahUKEwjb-uCektXaAhUDN48KHX-_B4IQMwimAigBMAE&iact=mrc&uact=8>

<https://www.google.com/imgres?imgurl=https%3A%2F%2Fwww.researchgate.net%2Fprofile%2FSachchidanand_Singh3%2Fpublication%2F282447418%2Ffigure%2Ffig2%2FAS%3A306991017807873%401450203770758%2FReference-Architecture-for-Large-E-commerce-Enterprise-The-reference-architectures-for.png&imgrefurl=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FReference-Architecture-for-Large-E-commerce-Enterprise-The-reference-architectures-for_fig2_282447418&docid=Bf8FDCOW8IjS7M&tbnid=0sp33ZgftnaKfM%3A&vet=10ahUKEwjb-uCektXaAhUDN48KHX-_B4IQMwioAigDMAM..i&w=850&h=1152&bih=649&biw=1366&q=ecommerce%20reference%20architecture&ved=0ahUKEwjb-uCektXaAhUDN48KHX-_B4IQMwioAigDMAM&iact=mrc&uact=8#h=1152&imgdii=NtbUZPUaWHFO2M:&vet=10ahUKEwjb-uCektXaAhUDN48KHX-_B4IQMwioAigDMAM..i&w=850>

**Kafka**

**Splunk**

**AppD**

**Estimation Techniques**

Expert estimation:

* + - 1. WBS Based(bottom up) :- PMP, Company template etc
      2. Group Estimation :- Planning Pocker
      3. Judgmental :- Expert Judgement
      4. Combination based :- analogy+WBS or Expert Judegement+ parametric

Formal Estimation:

* + - 1. Parametric Model :- COCOMO, SLIM
      2. Size Based:- Function point, Story points , Use case based.
      3. Analogy based (comparison):- ANGEL , Weighed Micropoints