REST Architectural Constraints

**REST** stands for REpresentational State Transfer , It’s a style of a software architecture that basically exploits the existing technology and protocols of the web. RESTful is typically used to refer to web services implementing such an architecture.

REST defines 6 architectural constraints which make any web service – a truly RESTful API.

1. *Uniform interface*
2. *Client–server*
3. *Stateless*
4. *Cacheable*
5. *Layered system*
6. *Code on demand (optional)*

1. **Uniform interface** - A resource in system should have only one logical URI, and then should provide way to fetch related or additional data. It’s always better to synonymize a resource with a web page. All resources should be accessible through a common approach such as HTTP GET.

2. **Client–server** - Client application and server application must be able to evolve separately without any dependency on each other. Client should know only resource URIs and that’s all.

3. **Stateless** - Server will not store anything about latest HTTP request client made. It will treat each and every request as new. No session, no history. Client is responsible for managing the state of application.

4. **Cacheable** - Caching can be implemented server side or client side. Can improve performance.

5. **Layered system** - Supports layered architecture. Develop APIs on different layers, client cannot confirm whether it is connected directly to the end server, or to an intermediary along the way.

6. **Code on demand (optional)** - Majorly REST works with static data in return i.e., XML or JSON. But it can also return executable code.

This is optional Refer :

<https://restfulapi.net/rest-architectural-constraints/>

**REST**:

1.Rest(Representational State Transfer)is a style of software architecture.

2.The Rest Service is simple to use it HTTP protocol for all operations.

3.Its lightweight and faster(it use the.JSON and .XML formats).

**RESTFULL**:

1.RestFULL(Representational State Transfer)is a style of software architecture.

2.It uses the Complex Mechanism like SOAP for Communication.

3.WCF service will allows to make calls and exchange the data using SOAP protocol over different protocols (HTTP, TCP, MSMQ etc.).

SOLID Principles

**S: Single Responsibility Principle (SRP).** SRP says "Every software module should have only one reason to change". This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose.

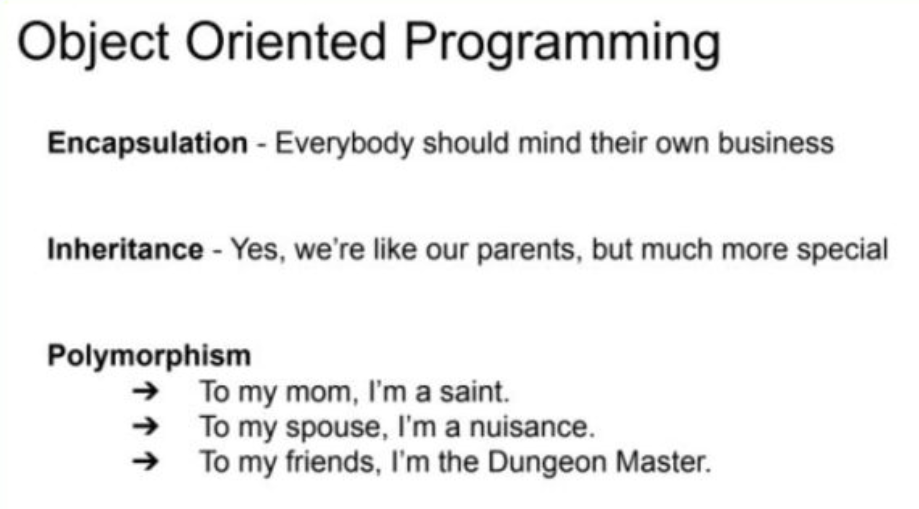
**O: Open closed Principle (OCP).** The Open/closed Principle says "A software module/class is open for extension and closed for modification". Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this.

**L: Liskov substitution Principle (LSP).** LSP states that "you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification". It ensures that a derived class does not affect the behavior of the parent class, in other words,, that a derived class must be substitutable for its base class.

**I: Interface Segregation Principle (ISP).** The Interface Segregation Principle states "that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.". An interface should be more closely related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs rather than which methods the class implements.

**D: Dependency Inversion Principle (DIP).** The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

3 Main Pillars of OOP



**Encapsulation**: This is the idea of wrapping everything up about a particular thing, whether a Checking Account or Armadillo, into a defined object with features and behaviors. Once we do, we can ask the object itself to do what it is supposed to do, whether that is DepositMoney or DefendYourself. But nobody outside the object needs to worry about how it does its jobs. We just tell it to do it and go about our day. If everybody, and by this I mean every object, simply minds its own business and stays out of the business of other objects, all is good with the world.

**Inheritance**: This is the idea that we don’t have to define absolutely everything about an object over and over again if it shares features and behaviors with other objects. We can define a class for Accounts and then let our Checking Account or Savings Account inherit all the stuff in common. Likewise, we can define a class for Animals, and let our Armadillo inherit features like NumberOfLegs and Weight as well as behaviors such as Breathe and Sleep. We call these overarching classes parent classes, and the ones that inherit from them, child classes. We can then inherit from the child classes and so on. But our Checking Account is more specialized than our Accounts because we can WriteACheck, which we can’t do with a Savings Account. Our Armadillo can RollIntoABall, but other animals such as a Giraffe don’t have that behavior.Since we go from more general to more specialized, I like to say that a child is like its parents, but much more special.

**Polymorphism**: This fancy name just means that we can treat the same object as different things depending on how we need it at different times, and we can treat groups of different objects that share an ancestor or trait as if they were that ancestor or trait.So, we could have a set of different Checking, Savings, and Credit Accounts and ask each to GetBalance so we can figure out how much we have to spend on vacation this year. Or we could ask a queue of animals to MoveQuickly, and not care how the Porpoise or Eagle or Armadillo would handle that shared behavior.I like to think that we are different things to different people, so even if not every Dungeon Master has a spouse to think him or her a nuisance, we can ask any of them to organize a game for Saturday night.

RESTful web API design

# RESTful web API design

Most modern web applications expose APIs that clients can use to interact with the application. A well-designed web API should aim to support:

* Platform independence. Any client should be able to call the API, regardless of how the API is implemented internally. This requires using standard protocols, and having a mechanism whereby the client and the web service can agree on the format of the data to exchange.
* Service evolution. The web API should be able to evolve and add functionality independently from client applications. As the API evolves, existing client applications should continue to function without modification. All functionality should be discoverable so that client applications can fully use it.

This guidance describes issues that you should consider when designing a web API.