Restful Web Services

What is an API?

- API is the acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other.
- The application that interprets the data sent from UI and presents you with the information you wanted in a readable way.

RESTful web service implementation defines the method of accessing various resources which are required by the client. The important aspects of this implementation include:

- Resources
- Request Headers
- Request Body
- Response Body
- Status codes

- Modern API's adhere to standards (typically HTTP and REST) that are developer-friendly, easily accessible and understood broadly
- They are treated more like products than code. They are designed for consumption for specific audiences (e.g., mobile developers), they are documented, and they are versioned in a way that users can have certain expectations of its maintenance and lifecycle.
- Because they are much more standardized, they have a much stronger discipline for security and governance, as well as monitored and managed for performance and scale
- As any other piece of productized software, the modern API has its own software development lifecycle (SDLC) of designing, testing, building, managing, and versioning. Also, modern APIs are well documented for consumption and versioning.

Features of webservice:

- Based on the Client Server representation.
- Use of HTTP protocol for performing functions like fetching data from the web service, retrieving resources, execution of any query, etc.
- The communication between the server and client is performed through the medium known as 'messaging'.
- Addressing of resources available on the server through URIs.
- Based on the concept of statelessness where every client request and the response is independent of the other with complete assurance of providing required information.
- Uses the concept of caching.
- Works on Uniform interface.

Stateless
Advantages:

- Every method required for communication is identified as an independent method i.e. there are no dependencies to other methods.
- Any previous communication with the client and server is not maintained and thus the whole process is very much simplified.
- If any information or metadata used earlier in required in another method, then the client sends again that information with HTTP request.
- HTTP protocol and REST web service, both shares the feature of statelessness.

Disadvantages:

- In every HTTP request from the client, the availability of some information regarding the client state is required by the web service.

Core components that come under HTTP Request:

- Verb: Includes methods like GET, PUT, POST, etc.
- Uniform Resource Identifier for identifying the resources available on the server.
- HTTP Version for specifying the HTTP version.
- HTTP Request header for containing the information about the data.
- HTTP Request body that contains the representation of the resources in use.

Explain Stateless w.r.t Restful Web Service:

- In REST, ST itself defines State Transfer and Statelessness means complete isolation.
- This means, the state of the client's application is never stored on the server and is passed on.
- In this process, the clients send all the information that is required for the server to fulfill the HTTP request that has been sent.
- Thus every client request and the response is independent of the other with complete assurance of providing required information.
- Every client passes a 'session identifier' which also acts as an identifier for each session.

Important Constraints of Restful web services:

- There should be separate concerns for each server and client which will help to maintain the modularity within the application. This will also reduce the complexity and increase the scalability.
- The client-server communication should be stateless, which means no previous information is used and the complete execution is done in isolation. In cases of failure, it also helps the client to recover.
- In client-server communication, the HTTP response

What is a Resource?

- 'Resource' is defined as an object of a type which can be an image, HTML file, text data, and any type of dynamic data. There are varieties of representation formats available in order to represent a resource.
- These are identified by logical URLs; it is the key element of a RESTful design.
    - JSON
    - YAML
    - XML
    - HTML

should be cacheable so that when required cached copy can be used which in turn enhances the scalability and performance of the server.

- The fourth constraint is the uniform interface which allows client-server interaction to be easily understood.
- Client-server communication should be done on a layered system and thus the client should only have knowledge about the intermediate level with which communication is being done.

Best practices that are followed while designing RESTful web services:

- Every input on the server should be validated.
- Input should be well formed.
- Never pass any sensitive data through URL.
- For any session, the user should be authenticated.
- Only HTTP error messages should be used for indicating any fault.
- Use message format that is easily understood and is required by the client.
- Unified Resource Identifier should be descriptive and easily understood.

Payload:

- The request data which is present in the body part of every HTTP message is referred as 'Payload'. In Restful web service, the payload can only be passed to the recipient through POST method.
- There is no limit of sending data as payload through POST method but the only concern is that more data with consuming more time and bandwidth.

JAX-RS:

- JAX-RS is defined as the Java API for RESTful web service. Among multiple libraries and framework, this is considered as the most suitable Java programming language based API which supports RESTful web service.
- Implementations:
    - Jersey
    - RESTEasy
    - Apache CFX
    - Play

Difference between AJAX and REST?

AJAX	REST
• In Ajax, the request are sent to the server by using XMLHttpRequest objects. The response is used by the JavaScript code to dynamically alter the current page • Ajax is a set of technology; it is a technique of dynamically updating parts of UI without having to reload the page • Ajax eliminates the interaction between the customer and server asynchronously • REST requires the interaction between the customer and server	• REST have a URL structure and a request/response pattern the revolve around the use of resources • REST is a type of software architecture and a method for users to request data or information from servers • REST requires the interaction between the customer and server

Differences between SOAP and REST?

SOAP	REST
• SOAP is a protocol through which two computer	• Rest is a service architecture and design for

communicates by sharing XML document
- SOAP permits only XML
- SOAP based reads cannot be cached
- SOAP is slower than REST
- It runs on HTTP but envelopes the message

network-based software architectures
- REST supports many different data formats
- REST reads can be cached
- A REST client is more like a browser; it knows how to standardized methods and an application has to fit inside it
- REST is faster than SOAP
- It uses the HTTP headers to hold meta information

RESTTemplate:
- The RestTemplate class is an implementation of Template method pattern in Spring framework.
- Similar to other popular template classes e.g. JdbcTemplate or JmsTemplate, it also simplifies the interaction with RESTful Web Services on the client side.
- You can use it to consume a RESTful Web Service.

HttpMessageConverter:
- An HttpMessageConverter is a Strategy interface that specifies a converter that can convert from and to HTTP requests and responses. Spring REST uses this interface to convert HTTP response to various formats e.g. JSON or XML.

@RequestMapping:
- The @RequestMapping annotation is used to map web requests to Spring Controller methods.
- You can map request based upon HTTP methods e.g. GET and POST and various other parameters.
- For examples, if you are developing RESTful Web Service using Spring then you can use produces and consumes property along with media type annotation to indicate that this method is only used to produce or consumers JSON as shown below:

```
@RequestMapping (method = RequestMethod.POST,
consumes="application/json")
public Book save(@RequestBody Book aBook) {
  return bookRepository.save(aBook);
}
```

@Controller and @RestController:
- Both @Controller and @RestController are stereotypes. The @Controller is actually a specialization of Spring's @Component stereotype annotation.
- This means that class annotated with @Controller will also be automatically be detected by Spring container as part of container's component scanning process.
- @RestController is a specialization of @Controller for RESTful web service.
- It not only combines @ResponseBody and @Controller annotation but also gives more meaning to your controller class to clearly indicate that it deals with RESTful requests.

@ResponseBody
- The @ResponseBody annotation can be put on a method to indicates that the return type should be written directly to the HTTP response body (and not placed in a Model, or interpreted as a view name).
- Alternatively, you can also use @RestController annotation instead of @Controller annotation. This will remove the need for using @ResponseBody

@PathVariable:
- It's one of the useful annotations from Spring MVC which allows you to read values from URI like query parameter. It's particularly useful in case of creating RESTful web service using Spring because in REST resource identifiers are part of URI.
- For example, in the URL http://myapp.com/books/101 if you want to extract 101 the id, then you can use @PathVariable annotation of Spring MVC.

@EnableWebMVC
- The @EnableWebMvc annotation is required to enable Spring MVC when Java configuration is used to configure Spring MVC instead of XML. It is equivalent to <mvc: annotation-driven> in XML configuration.
- It enables support for @Controller-annotated classes that use @RequestMapping to map incoming requests to handler methods.

@ResponseStatus:
- The @ResponseStatus annotation is required during error handling in Spring MVC and REST. Normally when an error or exception is thrown at server side, web server return a blanket HTTP status code 500 - Internal server error.
- This may work for a human user but not for REST clients. You need to send them proper status code e.g. 404 if the resource is not found. That's where you can

Securing REST:

- Security is a broad term, it could mean security of message which is provided by encryption or access restriction which is provided using authentication and authorization.
- REST is normally not secure but you can secure it by using Spring security.
- At the very least you can enable HTTP basic authentication by using HTTP in your Spring security configuration file.
- Similarly, you can expose your REST API using HTTPS if the underlying server supports HTTPS.

Transport Layer Security (REST):

- TLS or Transport Layer Security is used for secure communication between client and server. It is the successor of SSL (Secure Socket Layer). Since HTTPS can work with both SSL and TLS, REST can also work with TLS.
- Actually, REST says anything about Security, it's up to the server which implements that. Same RESTful Web Service can be accessed using HTTP and HTTPS if the server supports SSL.

use @ResponseStatus annotation, which allows you to send custom HTTP status code along with proper error message in case of Exception.

- In order to use it, you can create custom exceptions and annotated them using @ResponseStatus annotation and proper HTTP status code and reason.
- When such exceptions are thrown from controller's handler methods and not handled anywhere else, then appropriate HTTP response with the proper HTTP status code, which you have set is sent to the client.

```
@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No such Book")  // 404
public class BookNotFoundException extends RuntimeException { ...}
```

Caching Overview

What is Caching?

- A cache is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data's primary storage location. Caching allows you to efficiently reuse previously retrieved or computed data.

How does Cache work?

- The data in a cache is generally stored in fast access hardware such as RAM (Random-access memory) and may also be used in correlation with a software component. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer.
- Trading off capacity for speed, a cache typically stores a subset of data transiently, in contrast to databases whose data is usually complete and durable.

Applications:

- Caches can be applied and leveraged throughout various layers of technology including Operating Systems, Networking layers including Content Delivery Networks (CDN) and DNS, web applications, and Databases.
- You can use caching to significantly reduce latency and improve IOPS for many read-heavy application workloads, such as Q&A portals, gaming, media sharing, and social networking.
- Cached information can include the results of database queries, computationally intensive calculations, API requests/responses and web artifacts such as HTML, JavaScript, and image files.
- Compute-intensive workloads that manipulate data sets, such as recommendation engines and high-performance computing simulations also benefit from an In-Memory data layer acting as a cache.
- In these applications, very large data sets must be

Design Patterns:

- In a distributed computing environment, a dedicated caching layer enables systems and applications to run independently from the cache with their own lifecycles without the risk of affecting the cache.
- The cache serves as a central layer that can be accessed from disparate systems with its own lifecycle and architectural topology.
- This is especially relevant in a system where application nodes can be dynamically scaled in and out.
- If the cache is resident on the same node as the application or systems utilizing it, scaling may affect the integrity of the cache.
- In addition, when local caches are used, they only benefit the local application consuming the data.
- In a distributed caching environment, the data can span multiple cache servers and be stored in a central location for the benefit of all the consumers of that data.

accessed in real-time across clusters of machines that can span hundreds of nodes.
- Due to the speed of the underlying hardware, manipulating this data in a disk-based store is a significant bottleneck for these applications.

Caching Best Practises:
- When implementing a cache layer, it's important to understand the validity of the data being cached.
- A successful cache results in a high hit rate which means the data was present when fetched.
- A cache miss occurs when the data fetched was not present in the cache.
- Controls such as TTLs (Time to live) can be applied to expire the data accordingly.
- Another consideration may be whether or not the cache environment needs to be Highly Available, which can be satisfied by In-Memory engines such as Redis.
- In some cases, an In-Memory layer can be used as a standalone data storage layer in contrast to caching data from a primary location.
- In this scenario, it's important to define an appropriate RTO (Recovery Time Objective--the time it takes to recover from an outage) and RPO (Recovery Point Objective--the last point or transaction captured in the recovery) on the data resident in the In-Memory engine to determine whether or not this is suitable.
- Design strategies and characteristics of different In-Memory engines can be applied to meet most RTO and RPO requirements.

DNS Caching:
- Every domain request made on the internet essentially queries DNS cache servers in order to resolve the IP address associated with the domain name.
- DNS caching can occur on many levels including on the OS, via ISPs and DNS servers.

Sticky Sessions:
- When your website is served by only one web server, for each client-server pair, a session object is created and remains in the memory of the web server. All the requests from the client go to this web server and update this session object. If some data needs to be stored in the session object over the period of interaction, it is stored in this session object and stays there as long as the session exists.
- Now, if the requests are being served from (physically) 3 different servers, each server has created a session object for you and because these session objects sit on three independent boxes, there's no direct way of one knowing what is there in the session object of the other. In order to synchronize between these server sessions, you may have to write/read the session data into a layer which is common to all - like a DB. Now writing and

Benefits of Caching:
- Improve application performance.
- Reduce Database Cost
- Reduce the Load on the Backend
- Predicate Performance
- Eliminate database hotspots
- Increase read throughput

Database Caching:
- Database caching allows you to dramatically increase throughput and lower the data retrieval latency associated with backend databases, which as a result, improves the overall performance of your applications. The cache acts as an adjacent data access layer to your database that your applications can utilize in order to improve performance.

CDN Caching:
- A CDN provides you the ability to utilize its global network of edge locations to deliver a cached copy of web content such as videos, webpages, images and so on to your customers.
- To reduce response time, the CDN utilizes the nearest edge location to the customer or originating request location in order to reduce the response time.
- Throughput is dramatically increased given that the web assets are delivered from cache.

Session Management:
- HTTP sessions contain the user data exchanged between your site users and your web applications such as login information, shopping cart lists, previously viewed items and so on.
- Critical to providing great user experiences on your website is managing your HTTP sessions effectively by remembering your user's preferences and providing rich user context.
- With modern application architectures, utilizing a centralized session management data store is the ideal solution for a number of reasons including providing, consistent user experiences across all web servers, better session durability when your fleet of web servers is elastic and higher availability when session data is replicated across cache servers.

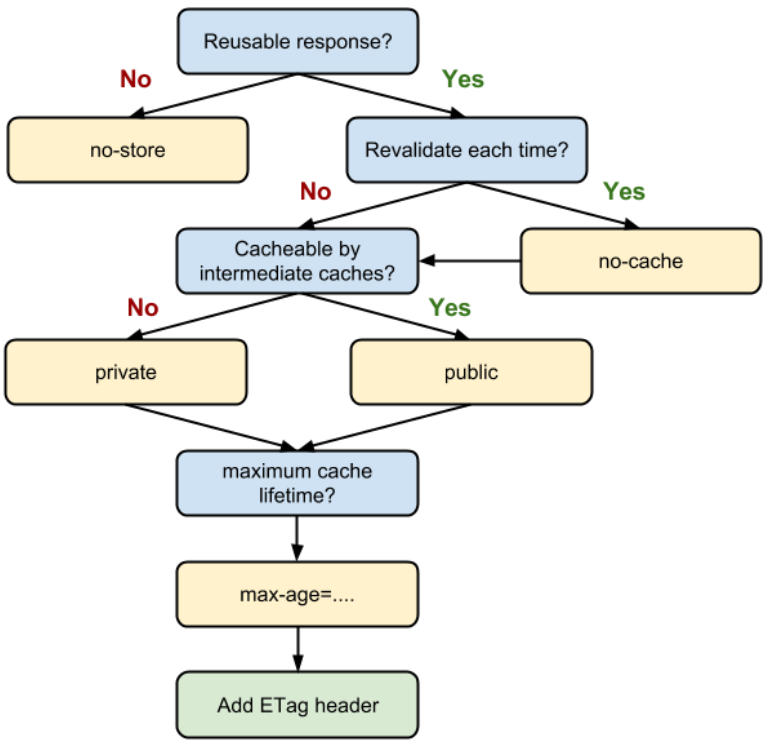
API's caching:
- An API generally is a RESTful web service that can be accessed over HTTP and exposes resources that allow the user to interact with the application.

reading data to/from a db for this use-case may not be a good idea. Now, here comes the role of sticky-session.

- If the load balancer is instructed to use sticky sessions, all of your interactions will happen with the same physical server, even though other servers are present. Thus, your session object will be the same throughout your entire interaction with this website.

Web Caching:

- When delivering web content to your viewers, much of the latency involved with retrieving web assets such as images, html documents, video, etc. can be greatly reduced by caching those artifacts and eliminating disk reads and server load.
- Various web caching techniques can be employed both on the server and on the client side.
- Server side web caching typically involves utilizing a web proxy which retains web responses from the web servers it sits in front of, effectively reducing their load and latency.
- Client side web caching can include browser based caching which retains a cached version of the previously visited web content.



Tips and techniques to keep in mind as you work on caching strategy:

- Use consistent URLs: if you serve the same content on different URLs, then that content will be fetched and stored multiple times.
- Ensure that the server provides a validation token (ETag): validation tokens eliminate the need to transfer the same bytes when a resource has not changed on

- When designing an API, it's important to consider the expected load on the API, the authorization to it, the effects of version changes on the API consumers and most importantly the API's ease of use, among other considerations.
- It's not always the case that an API needs to instantiate business logic and/or make a backend requests to a database on every request.
- Sometimes serving a cached result of the API will deliver the most optimal and cost-effective response.
- This is especially true when you are able to cache the API response to match the rate of change of the underlying data.

HTTP Caching:

- Every browser ships with an implementation of an HTTP cache. All you need to do is ensure that each server response provides the correct HTTP header directives to instruct the browser on when and for how long the browser can cache the response.
- When the server returns a response, it also emits a collection of HTTP headers, describing its content-type, length, caching directives, validation token, and more.

Cache-Control

- Each resource can define its caching policy via the Cache-Control HTTP header.
- Cache-Control directives control who can cache the response, under which conditions, and for how long.

no-cache and no-store

- "no-cache" indicates that the returned response can't be used to satisfy a subsequent request to the same URL without first checking with the server if the response has changed. As a result, if a proper validation token (ETag) is present, no-cache incurs a roundtrip to validate the cached response, but can eliminate the download if the resource has not changed.

- By contrast, "no-store" is much simpler. It simply disallows the browser and all intermediate caches from storing any version of the returned response

"public" vs. "private"

- If the response is marked as "public", then it can be cached, even if it has HTTP authentication associated with it, and even when the response status code isn't normally cacheable.

- By contrast, the browser can cache "private" responses. However, these responses are typically intended for a single user, so an intermediate cache is not allowed to cache them.

"max-age"

the server.
- Identify which resources can be cached by intermediaries: those with responses that are identical for all users are great candidates to be cached by a CDN and other intermediaries.
- Determine the optimal cache lifetime for each resource: different resources may have different freshness requirements. Audit and determine the appropriate max-age for each one.
- Determine the best cache hierarchy for your site: the combination of resource URLs with content fingerprints and short or no-cache lifetimes for HTML documents allows you to control how quickly the client picks up updates.
- Minimize churn: some resources are updated more frequently than others. If there is a particular part of a resource (for example, a JavaScript function or a set of CSS styles) that is often updated, consider delivering that code as a separate file. Doing so allows the remainder of the content (for example, library code that doesn't change very often), to be fetched from cache and minimizes the amount of downloaded content whenever an update is fetched.

- This directive specifies the maximum time in seconds that the fetched response is allowed to be reused from the time of the request. For example, "max-age=60" indicates that the response can be cached and reused for the next 60 seconds.

Load Balancer

Application Load Balancer:
- An Application Load Balancer makes routing decisions at the application layer (HTTP/HTTPS), supports path-based routing, and can route requests to one or more ports on each container instance in your cluster.
- Application Load Balancers support dynamic host port mapping.
- For example, if your task's container definition specifies port 80 for an NGINX container port, and port 0 for the host port, then the host port is dynamically chosen from the ephemeral port range of the container instance (such as 32768 to 61000 on the latest Amazon ECS-optimized AMI).
- When the task is launched, the NGINX container is registered with the Application Load Balancer as an instance ID and port combination, and traffic is distributed to the instance ID and port corresponding to that container.
- This dynamic mapping allows you to have multiple tasks from a single service on the same container instance.

Other load balancers are:
- Network load balancer
- Classic load balancer

Load Balancing Techniques:
- Round Robin
- Weighted Round Robin
- Least Connection
- Weighted Least Connection
- Agent-Based Adaptive Load Balancing
- Chained Failover (Fixed Weighted)
- Weighted Response Time
- Source IP Hash
- Software Defined Networking (SDN) Adaptive

Containers

- Containers provide a standard way to package your application's code, configurations, and dependencies into a single object.
- Containers share an operating system installed on the server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless

Benefits:
- Run anywhere:
  - Containers package your code with the configuration files and dependencies it needs to consistently run in any environment.
- Improve resource utilization:

<table>
<tr><td>of environment.
● The AWS Cloud offers infrastructure resources optimized for running containers, as well as a set of orchestration services that make it easy for you to build and run containerized applications in production.</td><td>○ Containers provide process isolation that lets you granularly set CPU and memory utilization for better use of compute resources.
● Scale quickly:
○ Each container runs as a separate process that shares the resources of the underlying operating system. This enables containers to start and stop quickly.</td></tr>
<tr><td>How they work:
● Build a container image
● Automate testing and deployment
● Run anywhere
● Scale rapidly</td><td></td></tr>
<tr><td colspan="2">Vagrant</td></tr>
<tr><td>● Vagrant is a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time.</td><td>● Vagrant will isolate dependencies and their configuration within a single disposable, consistent environment, without sacrificing any of the tools you are used to working with (editors, browsers, debuggers, etc.).
● Once you or someone else creates a single Vagrantfile, you just need to vagrant up and everything is installed and configured for you to work.
● Other members of your team create their development environments from the same configuration, so whether you are working on Linux, Mac OS X, or Windows, all your team members are running code in the same environment, against the same dependencies, all configured the same way.</td></tr>
<tr><td colspan="2">Swagger</td></tr>
<tr><td>What is Swagger?
● A set of rules to semantically describe an API
● Human and machine-friendly structured data format.
● Coded as a text file in YAML or JSON format.
● Generated by annotating source code or directly
● Often created before any code is written
● Enables sharing of API internally and externally
● Automatically generates API documentation.
● Now called the “Open API Initiative” or OAS</td><td>Why use Swagger?
● Integrating a REST API into your app can be troublesome and difficult
● REST semantics can vary widely by vendor and language used
● Documentation for developers is often an afterthought
● Negates the need for trial-and-error experimentation to discover how an API works.</td></tr>
<tr><td colspan="2">Rules Engine</td></tr>
<tr><td>● RuleBook also allows you to specify rules using an easy to use Lambda enabled Domain Specific Language or using POJOs that you define!
● Tired of classes filled with if/then/else statements? Need a nice abstraction that allows rules to be easily specified in a way that decouples them from each other?

Thread Safe
● RuleBooks are threadsafe. However, FactMaps and</td><td>Given-When-Then: The Basis of the RuleBook Language
● The RuleBook Java Domain Specific Language (DSL) uses the Given-When-Then format, popularized by Behavior Driven Development (BDD) and associated testing frameworks (e.g. Cucumber and Spock).
○ Given some Fact(s)
○ When a condition evaluates to true
○ Then an action is triggered

● Given methods can accept one or more facts in various</td></tr>
</table>

other implementations of NameValueReferableMap are not.
- This means that a single instance of a RuleBook can be run in different threads with different Facts without unexpected results.
- However, using the same exact FactMap across different threads may cause unexpected results.
- Facts represent data for individual invocations of a RuleBook, whereas RuleBooks represent reusable sets of Rules.

Facts:
- Using methods reduce the set of facts available to a then() method. Multiple using() methods can also be chained together if so desired. The aggregate of the facts with the names specified in all using() methods immediately preceding a then() method will be made available to that then() method.
- Stop methods break the rule chain. If a stop() method is specified when defining a rule, it means that if the when() condition evaluates to true, following the completion of the then() action(s), the rule chain should be broken and no more rules in that chain should be evaluated.
- Facts can be provided to Rules using the given() method.

Rule Chain Behavior
- By default, errors found when loading rules or exceptions thrown when running rules, remove those rules from the rule chain. In other words, rules that error are just skipped. Additionally, by default, a rule can only stop the rule chain if its condition evaluates to true and if its actions successfully complete.

different forms and are used as a collection of information provided to a single Rule. When grouping Rules into a RuleBook, facts are supplied to the Rules when the RuleBook is run, so the 'Given' can be inferred.

- When methods accept a Predicate that evaluates a condition based on the Facts provided. Only one when() method can be specified per Rule.

- Then methods accept a Consumer (or BiConsumer for Rules that have a Result) that describe the action to be invoked if the condition in the when() method evaluates to true. There can be multiple then() methods specified in a Rule that will all be invoked in the order they are specified if the when() condition evaluates to true.

Behavior Driven Development

- Behavior Driven Development (BDD) is a methodology for developing software through continuous example-based communication between developers, QAs and BAs.
- The primary purpose of BDD methodology is to improve communication amongst the stakeholders of the project so that each feature is correctly understood by all members of the team before development process starts.
- This helps to identify key scenarios for each story and also to eradicate ambiguities from requirements.
- In BDD, Examples are called Scenarios. Scenarios are written in a special format called Gherkin. The scenarios explain how a given feature should behave in different situations with different input parameters. They are called “Executable Specifications” because Gherkin is structural and it serves both specification and input into automated tests.

- Acceptance criteria should be written in terms of scenarios and implemented as classes:
- Given [initial context],
- when [event occurs],
- then [ensure some outcomes].

- The goal of BDD is a business readable and domain-specific language that allows you to describe a system’s behavior without explaining how that behavior is implemented.

E.g.
Feature: log into the system. User should be signed into the system when he provides valid username and password

Scenario: Successful login with Valid Credentials

Given User is at the Home Page
And Navigate to LogIn Page
When User enters credentials
| username | password |
| testuser_1 | Test@123 |
And User login into a system
Then user is on the main profile page

Domain Driven Development

- DDD is about trying to make your software a model of a real-world system or process. In using DDD, you are meant to work closely with a domain expert who can explain how the real-world system works. For example, if you're developing a system that handles the placing of bets on horse races, your domain expert might be an experienced bookmaker.
- Between yourself and the domain expert, you build a ubiquitous language (UL), which is basically a conceptual description of the system. The idea is that you should be able to write down what the system does in a way that the domain expert can read it and verify that it is correct. In our betting example, the ubiquitous language would include the definition of words such as 'race', 'bet', 'odds' and so on.
- The concepts described by the UL will form the basis of your object-oriented design. DDD provides some clear guidance on how your objects should interact, and helps you divide your objects into the following categories:
    - Value objects, which represent a value that might have sub-parts (for example, a date may have a day, month and year)
    - Entities, which are objects with identity. For example, each Customer object has its own identity, so we know that two customers with the same name are not the same customer
    - Aggregate roots are objects that own other objects. This is a complex concept and works on the basis that there are some objects that don't make sense unless they have an owner. For example, an 'Order Line' object doesn't make sense without an 'Order' to belong to, so we say that the Order is the aggregate root, and Order Line objects can only be manipulated via methods in the Order object

- DDD also recommends several patterns:
    - Repository, a pattern for persistence (saving and loading your data, typically to/from a database)
    - Factory, a pattern for object creation
    - Service, a pattern for creating objects that manipulate your main domain objects without being a part of the domain themselves

TDD

Rules:

- Write only enough of a unit test to fail.
- Write only enough production code to make the failing unit test pass.

Red Green Refactor cycle:

- Red Phase:
    - In the red phase, you have to write a test on a behavior that you are about to implement. Yes, I wrote behavior. The word “test” in Test Driven Development is misleading. We should have called it “Behavioral Driven Development“ in the first place. Yes, I know, some people argue that BDD is different from TDD, but I don’t know if I agree. So in my simplified definition, BDD = TDD.
- Green Phase:
    - This is usually the easiest phase, because in this phase you write (production) code. If you are a programmer, you do that all the time.
    - In this phase, you need to act like a programmer who has one simple task: write a straightforward solution that makes the test pass (and makes the alarming red on the test report becomes a friendly green). In this phase, you are allowed to violate best practices and even duplicate code. Code duplication will be removed in the refactor phase.

- Refactor Phase:
    - In the refactor phase, you are allowed to change the code, while keeping all tests green, so that it becomes better. What “better” means is up to you. But there is something mandatory: you have to remove code duplication. Kent Becks suggests in his book that removing code duplication is all you need to do.
    - Removing code duplication often results in abstraction.

Conclusion:

- T.D.D. requires much more time than “normal” programming!
- How many test do I have to write?
- The minimum amount that lets you write all the production code.
- TDD is not a substitution for the analysis/design phase.

Asynchronous programming:

- Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress.

When to Use?

- There are a large number of tasks so there is likely always at least one task that can make progress.
- The tasks perform lots of I/O, causing a synchronous program to waste lots of time blocking when other tasks could be running.
- The tasks are largely independent from one another so there is little need for inter-task communication.

Benefits:

- Performance
- Responsiveness

Threading vs Asynchronous programming:

- Threading is about workers; asynchrony is about tasks.
- In multithreaded workflows you assign tasks to workers.
- In asynchronous single-threaded workflows you have a graph of tasks where some tasks depend on the results of others; as each task completes it invokes the code that schedules the next task that can run, given the results of the just-completed task. But you (hopefully) only need one worker to perform all the tasks, not one worker per task.

Data Orchestration vs Choreography

- As we start to model more and more complex logic, we have to deal with the problem of managing business processes that stretch across the boundary of individual services.
- And with microservices, we’ll hit this limit sooner than usual. [...] When it comes to actually implementing this flow, there are two styles of architecture we could follow.
- With orchestration, we rely on a central brain to guide and drive the process, much like the conductor in an orchestra.
- With choreography, we inform each part of the system of its job, and let it work out the details, like dancers all finding their way and reacting to others around them in a ballet.

Choreography:

- With a choreographed approach, we could instead just have the customer service emit an event in an asynchronous manner, saying Customer created.
- The email service, postal service, and loyalty points

Orchestration:

- Here, probably the simplest thing to do would be to have our customer service act as the central brain.
- On creation, it talks to the loyalty points bank, email service, and postal service [...], through a series of request/response calls.
- The customer service itself can then track where a customer is in this process.
- It can check to see if the customer’s account has been set up, or the email sent, or the post delivered.
- We get to take the flowchart [...] and model it directly into code.
- We could even use tooling that implements this for us, perhaps using an appropriate rules engine.
- Commercial tools exist for this very purpose in the form of business process modeling software.
- Assuming we use synchronous request/response, we could even know if each stage has worked [...] The downside to this orchestration approach is that the customer service can become too much of a central governing authority.

bank then just subscribe to these events and react accordingly [...]
- This approach is significantly more decoupled.
- If some other service needed to reach to the creation of a customer, it just needs to subscribe to the events and do its job when needed.
- The downside is that the explicit view of the business process we see in [the workflow] is now only implicitly reflected in our system [...]
- This means additional work is needed to ensure that you can monitor and track that the right things have happened.
- For example, would you know if the loyalty points bank had a bug and for some reason didn't set up the correct account?
- One approach I like for dealing with this is to build a monitoring system that explicitly matches the view of the business process in [the workflow], but then tracks what each of the services does as independent entities, letting you see odd exceptions mapped onto the more explicit process flow.
- The [flowchart] [...] isn't the driving force, but just one lens through which we can see how the system is behaving.
- In general, I have found that systems that tend more toward the choreographed approach are more loosely coupled, and are more flexible and amenable to change.
- You do need to do extra work to monitor and track the processes across system boundaries, however.
- I have found most heavily orchestrated implementations to be extremely brittle, with a higher cost of change.
- With that in mind, I strongly prefer aiming for a choreographed system, where each service is smart enough to understand its role in the whole dance.

- It can become the hub in the middle of a web, and a central point where logic starts to live.
- I have seen this approach result in a small number of smart "god" services telling anemic CRUD-based services what to do.

Replication vs Partitioning vs Clustering vs Sharding

- Replication:
    - The strategy of duplicating data across more than one node. There are many varieties, simplest being Master-Slave(s) or Leader-Follower and can either be synchronous or asynchronous.
- Partitioning:
    - Really just a generic term for dividing a single data set into smaller sets. There are many different ways to do this, including sharding and clustering. It can also be single node, such splitting or fragmenting databases and tables into subsets to decrease index size for example.
- Clustering:
    - There are a couple ways to define this term.
        - Using multiple application servers to access the same database. Used for computation intensive, parallelized, analytical applications that work on non volatile data.
        - However, in database terms, clustering is another subset of partitioning, similar to sharding, though the nodes are aware of each other and function as a unit. Data typically is distributed automatically and rebalanced automatically.
- Sharding:
    - A subset of partitioning, meaning to horizontally partition, typically across independent nodes. Often, data is sharded by a shared key to ensure data locality. Typically, nodes are unaware of each other and are more resilient

to failures of their siblings.

Note: Information gathered in this document has been collected from various sources on Internet.