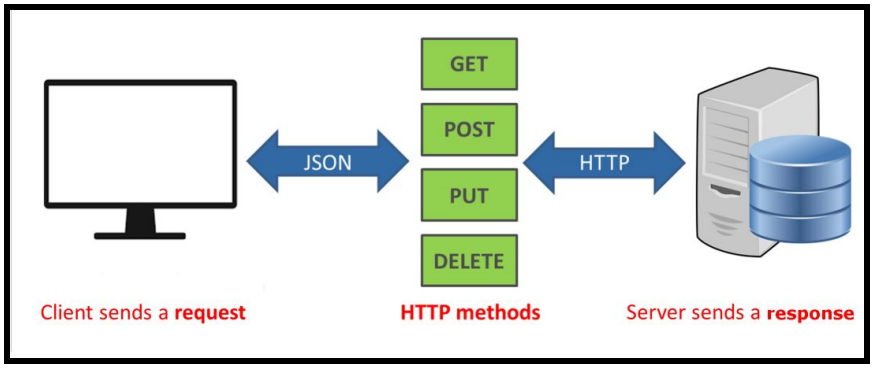
Software Architecture - Overview:

- There are two ways of constructing a software design:
  One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
- Definition: **Software architecture** is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is the result of the software development activity.
- **Architectural Design/High-Level Design:**
  - It provides an overview of a system.
    I.e. It provides an overview of the overall structure such as main modules and their connections.
  - It covers the main use-cases of the system.
  - It addresses the main non-functional requirements (e.g. throughput, reliability).
  - It is hard to change.
- **Detailed Design/Low-Level Design:**
  - It is created based on the high-level design.
  - It describes the class diagrams with the methods and relations between classes and program specs. It describes the modules so that the programmer can directly code the program from the document.
  - It is the inner structure of the main modules.
  - It may take the target programming language into account.
  - It is detailed enough to be implemented in the programming language.
- **Client Server Architecture:**
- The **Client Server Architecture** is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client.
- This type of architecture has one or more client computers connected to a central server over a network or internet connection. This system shares computing resources.
- The client-server architecture is an architecture of a computer network in which many clients (remote processors) request and receive service from a centralized server (host computer). Client computers provide an interface to allow a computer user to request the services of a server and to display the results the server returns. Servers wait for requests to arrive from clients and then respond to them. Ideally, a server provides a standardized transparent interface to clients so that clients need not be aware of the specifics of the system that is providing the service. Clients are often situated at workstations or on personal computers, while servers are located elsewhere on the network, usually on more powerful machines. This computing model is especially effective when clients and the server each have distinct tasks that they routinely perform.
- Advantages:
  - **Improved Data Sharing:** Data, which is retained by usual business processes and is manipulated on a server, is available for designated clients over an authorized access. The use of SQL supports open access from all client aspects and also transparency in network services depict that similar data is being shared among users.
  - **Integration of Services:** Every client is given the opportunity to access corporate information via the desktop interface eliminating the necessity to log into a terminal mode or another processor. Desktop tools like spreadsheet, powerpoint presentations, etc can be used to deal with corporate data with the help of database and application servers resident on the network to produce meaningful information.

    - **Shared Resources amongst Different Platforms:** Applications used for the client/server model are built regardless of the hardware platform or technical background of the clients.
    - **Inter-Operation of Data:** All development tools used for client/server applications access the back-end database server through SQL. Advanced database products enable users to gain a merged view of corporate data dispersed over several platforms. Rather than a single target platform this ensures database integrity with the ability to perform updates on multiple locations enforcing quality recital and recovery.
    - **Data Processing capability despite the location:** Through the client or server users can directly log into a system despite the location or technology of the processors.
    - **Easy maintenance:** Since client/server architecture is a distributed model representing dispersed responsibilities among independent computers integrated across a network, it's an advantage in terms of maintenance. It's easy to replace, repair, upgrade and relocate a server while clients remain unaffected. This unawareness of change is called encapsulation.
    - **Security:** Servers have better control access and resources to ensure that only authorized clients can access or manipulate data and server-updates are administered effectively.
- Disadvantages:
    - **Overloaded servers:** When there are frequent simultaneous client requests, servers severely get overloaded, forming traffic congestion.
    - **Impact of centralized architecture:** Since it is centralized, if a critical server fails, client requests will not be accomplished.

REST API:

- An **API (Application Programming Interface)** is a set of rules and mechanisms by which one application or component interacts with the others.
- **REST** stands for REpresentational State Transfer.
- Introduced by Roy Fielding in 2000.
- REST is a web standards based architecture and uses the HTTP Protocol.
- Every component is a resource and a resource is accessed by a common interface using HTTP standard methods.
- In REST architecture, a REST Server simply provides access to the resources and a REST client accesses and modifies the resources. Here each resource is identified by URIs or global IDs. REST uses various representations, such as text, JSON, XML, etc, to represent a resource. JSON is the most popular one.
  I.e.



- A JSON object is a key-value data format that is typically rendered in curly braces.

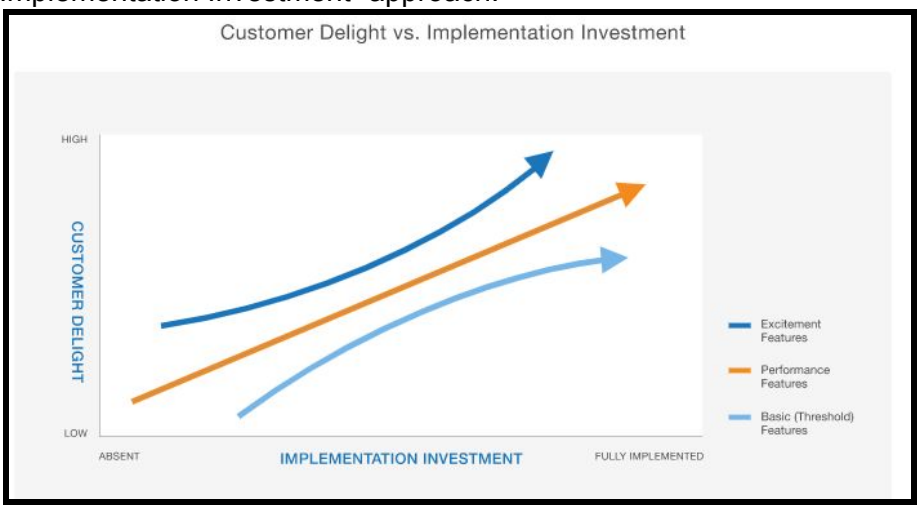
- When you're working with JSON, you'll likely see JSON objects in a .json file, but they can also exist as a JSON object or string within the context of a program.
- A JSON object looks something like this:
  {"first_name" : "Sammy",
  "last_name" : "Shark",
  "location" : "Ocean",
  "online" : true,
  "followers" : 987}
- Most REST implementations depend on HTTP as the protocol because it defines a communication protocol that's suitable for REST.
- **Note:** For something to be considered a RESTful API, it doesn't necessarily need to run over HTTP. It can use other transfer protocols such as SNMP, SMTP, etc.
- Remember, HTTP is a communication protocol while REST is an architectural design.
- Normally an API is created to solve a business problem of some sort.
- It's important that business entities not change throughout the API design.
- These entities don't necessarily need to be equal to a single data entity.
  E.g. A customer entity could be built up coming from a variety of data sources. When presented to the consumer of the API, the customer returns all the data in a single form. So if customer information is stored in two different databases and spread across three tables we can still build that data into a customer entity.
- Every resource created should have an identifier.
  This is what the industry calls a **URI (Uniform Resource Identifier)**.
- The most important thing to remember regarding these resource identifiers is that they should be unique.
  I.e. They should uniquely identify a given resource/entity.
- Accessing the resources should then be possible via these URIs. They usually have the following form:
  http://domain.com/departments
  Executing an HTTP GET request against this URI should retrieve all the departments.
  We may also think about this as collections.
  The departments collection contains all the single department entities.
- **Resource hierarchy** is how we want to organise and represent relationships between various entities.
- E.g. Let's say that we are developing an API for an HR system and suppose the entities are Departments, Employees and Salaries.
  If we want to retrieve information about an individual department we should be able to pass in a department ID and therefore make the following HTTP GET request:
  http://domain.com/departments/12.
  We could also get some information, such as the name or phone number, of a given employee working at a specific department by making the following GET call:
  http://domain.com/departments/12/employees/1568.
- We need to define the data structure that we'll return for requests.
  These days, it's safe to say that the data format used in RESTful APIs is JSON.
  Although it could be XML as well.
- It's important to remember that the JSON structure returned by the API should be decoupled from the underlying database. If we use a relational database, which makes heavy uses of schemas, a change in the schema should not affect the representation of the entity in the API.

- HTTP Methods:
    - **GET:** Returns a resource. Either an entire collection or an individual resource. Both are based on a URI.
    - **POST:** Creates a new resource at the URI specified as part of the request. This request requires a payload which contains the details of the resource to be added.
    - **DELETE:** Used to remove a resource at the specified URI.
    - **PUT:** Creates or replaces a resource at a URI specified. Again, the payload contains the details of the resource. The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly has side effects of creating the same resource multiple times.

Prioritizing the Product Backlog:

- Prior to each sprint, the product owner is allowed to change the priorities in the product backlog.
- Decisions can be based on the following criteria:
    - **Value:** What value does the story add and how does it help maximize Return on Investment (ROI).
    - **Cost:** Some features may prove too costly to implement and affect ROI.
    - **Risk:** Uncertainty about value/cost.
    - **Knowledge:** If the product owner doesn't have enough information about the features to do a proper estimate they can introduce a **spike** to explore it and get more info. A spike is a user story for which the team cannot estimate the effort needed. In such a case, it is better to run time-boxed research, exploration to learn about the issue or the possible solutions. As a result of the spike, the team can break down the features into stories and estimate them. Thus, you may consider a spike as an investment for a product owner to figure out what needs to be built and how the team is going to build it. The product owner allocates a little bit of the team's capacity now, ahead of when the story needs to be delivered, so that when the story comes into the sprint, the team knows what to do.
- Kano Model:
    - Developed by Noriaki Kano in the 1970s and 1980s while studying quality control and customer satisfaction.
    - It is an approach to prioritizing features on a product roadmap based on the degree to which they are likely to satisfy customers. Product teams can weigh a high-satisfaction feature against its costs to implement, to determine whether or not adding it to the roadmap is a strategically sound decision.
    - Using the Kano Model, product teams pull together a list of potential new features vying for development resources and space on the roadmap. The team will then weigh these features according to two competing criteria:
        1. Their potential to satisfy customers.
        2. The investment needed to implement them.

In fact, you can also think of the Kano Model as the “Customer Delight vs. Implementation Investment” approach.



- The Kano Model identifies three types of initiatives product teams will want to develop:
    1. **Basic features:**
       These are features your product needs to be competitive. Customers expect these features and take them for granted. This means they must be included. And, if they don’t work as expected they may lead to dissatisfaction.
       Basic features are what a user expects to be there and work but will never score high on satisfaction. They can take inordinate amounts of effort to build and maintain.
    2. **Performance features:**
       These are features that give you a proportionate increase in customer satisfaction as you invest in them. These are also features customers know they want and weigh heavily when deciding whether to choose your product or your competitor’s.
    3. **Excitement features**
       Excitement features yield a disproportionate increase in customer delight as you invest in them. If you don’t have these features, customers might not even miss them; but if you include them, and continue to invest in them, you will create dramatic customer delight. You can also think of these features as the unique innovations and surprises you include in your product. These are features that delight the user. These score very highly on satisfaction and in many cases may not take as much investment. Small incremental improvements here have an outsized impact on customer satisfaction.
- **Satisfiers** are requirements that customers are not expecting, but the more of them that are included, the happier the customer.

- Kano proposed determining the category of a feature by asking two questions:
    1. How the user would feel if the feature were present in the product.
    2. How the user would feel if it were absent.

  The answers to these questions are on the same five-point scale:
    1. I like it that way.
    2. I expect it to be that way.
    3. I am neutral.
    4. I can live with it that way.
    5. I dislike it that way.
- The Kano Model can be a helpful framework for product teams with limited time and resources who want to make sure they are prioritizing the appropriate mix of features to work on next.