Week1

- *JAMStack* is a technology stack comprising of JavaScript, API and Markup. It separates out the frontend from the backend, and they both can be updated/modified independently without affecting the other. The term is coined by *Mathias Billmann*.
- JavaScript can be used on the client/browser or server (node) side. It helps reduce the load on the server.
- Most JavaScript engines at the browser use UTF-16 for encoding.
- Comparison using "===" operator checks the equality of value and type of data. This helps avoid implicit "coercion".
- *var* has a function-level scope, while *let* and *const* has a block-level scope. It implies variables declared using *let* and *const* don't exist outside the block. However *var* can change a global variable. To create a global variable, don't use any keyword.

```
function quiz() {
    a = 10  // Global
    var b = 20 //Function scope.
    console.log(b)
    let c = 20 //Block scope
    console.log(c)
}
quiz()
console.log(a) //prints 10
console.log(b) //ReferenceError
console.log(c)  //ReferenceError
```

- While defining *const* variables, initialization is mandatory. Else, *syntax error* ensues.
- AJAX stands for *Asynchronous JavaScript and XML.* The term was first used in 2005 by Garrett
- Original JS could be executed only from a browser, but Node.js allowed executing JS code in a command line environment.
- A polyfill is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

Week2

- To find length of an array x, use x.length
- To delete all elements of an array x, use x = []
- To delete an element from an array x, use delete x[<idx>]. However, this operation leaves a hole.
- To delete the last element of an array x, use x.pop()
- To delete the first element of an array x, use x.shift()
- To delete multiple elements of an array x, use x.splice(<start_idx>, <#elements>)
- To add element into an array x, use x.push()
- Assigning x.length creates holes in the array.
- In a C-style for loop, one cannot use a *const* to declare the variable, since *const* cannot be incremented. Use *let* or *var*.
- C-style for loop loops over all indices of the array. for…in also loops over array indices. for…of loops over values instead.

- C-style for loop loops over holes; Similarly, for…of also loops over holes. for…in does not loop over holes.
- Object.keys(x) returns a list with the indices of the array x. Note that all indices are of type string, not number. This is unlike Python.
- Object.entries(x) returns a list containing lists, each containing a key (as a string) and its corresponding value.
- Values in an object can be accessed as x['first'] or x.first, assuming *first* is a key in x. console.log(x) will print all its entries.
- C-style for loop loops over all indices of the object. for…of cannot be used with objects directly (because objects are not iterable), unless used on its entries like Object.entries(x).
- new Array() defines an array, all of whose elements are holes initially. Array() can also be used instead.
- Use …x to spread out the array x inside another data structure.
- If x is an array, [...x] creates a copy of x.
- y = x.find(t => t < 0) finds the first value in array x that's less than 0.
- y = x.filter(t => t < 0) finds all values in array x that's less than 0, and returns them in an array format.
- x.map(t => t > 0 ? '+' : '-') returns an array containing '+' for all positive values in array x, and '-' for all negative values.
- x.reduce((a, i) => a + i, 0) returns the cumulative sum of all elements in *x*. *a* gets initial value of 0, and is accumulated as it marches through each element in *x*(represented as *i*).
- x.sort() sorts all elements in the array, but does a lexical sort. x.sort((a, b) => a - b) sorts all elements in the ascending order. Similarly, x.sort((a, b) => b - a) sorts all elements in the descending order.
- Array de-structuring

```
let x = [1, 2, 3];
let [a, b] = x;
```

  a gets 1 and *b* gets 2, after the above code is run. Value 3 is ignored.
  NOTE: If the second line was a, b = x;, it would give *ReferenceError*.
- Object de-structuring

```
const person = {
  firstName: 'Albert',
  lastName: 'Pinto',
  age: 25,
  city: 'Mumbai'
};
const {firstName: fn, city: c} = person;
```

  fn is assigned 'Albert' and *c* is assigned 'Mumbai', after executing the above code.
- Object.keys(person).length returns the number of keys in the object *person*. Note that person.length will return *undefined*.
- Another example of destructuring

```
<script>
    fruit = {
        name: 'Apple',
        color: 'red',
    }
    let description = ({name, color, shape = 'Spherical'}) => {
        console.log(`${name} is ${color} and ${shape}`)
    }
    description (fruit) //Apple is red and Spherical
</script>
```

- type=" module" is mandatory when using the script inside an html file, if there're import/export statements in the script.
- import {c, energy} from './module1.js'; imports variables *c* and *energy* exported from module1.js. In order to export, use export c; in module1.js. Note that it's not necessary to export *internal* functions of module1.js.
- variables/functions can be renamed during *export* or *import* process.
- When importing, only a read-only view of the export variable is available. Hence, it's not possible to change the value of *c*, after importing.
- Implementation of class in JS involves setting *prototype* of one object to another (which will then act as its parent)
- JS doesn't support multiple inheritance. In the case of inheritance, the constructor must explicitly call super().
- *this* refers to the current object.

```
<script>
    obj = {
        first_name: "Anand",
        last_name: "Iyer",
    }

    obj.full_name = function () {
        return (this.first_name + " "+ this.last_name)
    }

    console.log(obj.full_name())  // ANand Iyer
</script>
```

NOTE: Instead of defining *full_name* as a function, it's possible to define it as a property by using *get* keyword. Similarly, *set* keyword can be used to set one or more attributes of the object.

```
<script>
    obj = {
        first_name: "Anand",
        last_name: "Iyer",
        get full_name() {
            return (this.first_name + " "+ this.last_name)
        },
        set full_name(full) {
            let [first, last] = full.split(' ')
            this.first_name = first
            this.last_name = last
        }
    }

    console.log(obj.full_name)  // ANand Iyer
    obj.full_name = "Aravind Krishnan"
    console.log(obj.first_name) // Aravind
</script>
```

- When executed as a script in the HTML, console.log(this) outputs Window object. However, on a *node.js environment/browser console* (REPL in general), output will be a global object.
- this refers to the calling object when used in a normal function, but since arrow functions work on the parent scope (global object), it will not work. Note that, in the following example, in order to add *name* to global object, it must be defined using *var* outside of the object definition *o*.

```
<script>
    o = {
        name: 'Anand',
        a: function() {
            console.log('hello', this.name)
        },

        b: () => {
            console.log('hello', this.name)
        }
    }

    o.a() //hello Anand
    o.b() //hello
</script>
```

  NOTE: In the case of execution from a *browser console/node environment*, the variable shows up in global object only when it's declared using *var* keyword, and doesn't when declared using *let* or *const*; however, In the case of a *script* execution, the variable shows up in global object, irrespective of the declaration mode (*var, let* or *const*)
- Here is an example of how *var/let* alters execution.

```
let x = 5;
let obj = {
   'x' : 10,
   func : (x) => {
       this.x = x;
   }
}
obj.func(20);
console.log(x, "and", obj.x);
```

  Above program will print 5 and 10. However, if *x* is declared using *var* instead of *let, x* is treated as a global variable and hence the program will print 20 and 10.
- Here is another example of how *var/let* alters execution.

```
for (var i = 1; i<=3; i++) {
    setTimeout(() => console.log(i), i*1000)
}
```

  Above program will print 4, 4, 4. This is because the logs are created inside *setTimeout* which executes only after the 1, 2, 3 seconds respectively. By the time, the logs are created, *i* is already 4. Note that *i* is declared as a global variable. However, if *i* were declared using *let* (instead of *var*), the above program would print 1, 2, 3 respectively.
- To make a copy of a variable, use the spread operator like z = ...x
- In JavaScript, a function has a method named *call*, which when called will in turn invoke the function. The first parameter for the *call* method is the *context* in which you're invoking it.
- *apply* method of the function allows to call it in the context of another object, while also passing a list of arguments as a list.
- *bind* method of the function allows to create a closure and creates another short-cut function with a pre-defined argument.

```
<script>
    let obj = {
        first_name: "Anand",
        last_name: "Iyer",
        set_first_name: function (first) {
            this.first_name = first
        },
    }

    let obj2 = {
        first_name: "Bhaskar",
        last_name: "Banerjee",
    }

    f = obj.set_first_name
    f.call(obj, 'Aravind')
    console.log(obj.first_name) //Aravind

    console.log(obj2.first_name) //Bhaskar
    f = obj.set_first_name
    f.call(obj2, 'Aravind')
    console.log(obj2.first_name) //Aravind

    f.apply(obj, ['Krishna'])
    console.log(obj.first_name) //Krishna

    f1 = f.bind(obj, 'Ramesh') //Creating a closure - a new function
    f1.call(obj)
    console.log(obj.first_name) //Ramesh
</script>
```

`

- When used with arrow functions, the first argument gets ignored in *call*, *apply* and *bind* and always refer to the enclosing context.
- Implementation of class in JavaScript uses *__proto__* keyword.

```
const x = {a: 1, inc: function() {this.a++;}};
console.log(x);
const y = {__proto__: x, b: 2};
console.log(y);
console.log(y.a);
y.inc();
console.log(y.a);
```

Here, *x* is a prototype of *y*, which means that *y* inherits all properties and functions defined by *x*. Thus, the last line in the above code will print *2* (increments value of a = 1)

- Here's an example of class implementation and inheritance.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  describe() {
    return `${this.name} makes a sound $
    {this.sound}`
  }
}
let x = new Animal('Jerry');
console.log(x.describe());

class Dog extends Animal {
  constructor(name) {
    super(name);
    this.sound = 'Woof';
  }
}
let d = new Dog('Spike');
console.log(d.describe());
```

The parent class *Animal describes* the object using its *name* and its *sound*. Although the *name* is defined, the *sound* is not defined until *Animal* class has been extended to create a *Dog* class.

Thus, in the above code, the first x.describe() prints "Jerry makes a sound undefined" whereas the last x.describe() prints "Spike makes a sound Woof"

- In order to get the name of the class from its object O, use it like O.constructor.name
- When arrow functions are used, it's not mandatory to use flower brackets if there's a single statement in the function block. No need to use *return* keyword in this case.
- Hoisting allows a user to use a named normal function before it is defined. Note that arrow functions aren't hoisted.
- All variables including those declared using *var*, *let* or *const* are hoisted. However, *let* and *const* variables are not accessible before they're defined, since they're "hoisted" into a temporary dead zone, unlike those declared with *var*. Accessing such variables (before they're defined) will result in *ReferenceError*.
- Whenever a property is accessed via an object, the output always comes as *undefined*, if the property accessed does not exist. However, if the variable is accessed directly (without referencing via an object), it causes *ReferenceError*, if the variable doesn't exist.

```
const Obj1 = {
    y: 15,
    getY: () => {
        return this.y;
    },
    obj3: {
        y: 45,
        getY: () => {
            return this.y;
        },
    }
}
console.log(Obj1.obj3.getY(), Obj1.getY())
```

The output is undefined undefined.

- To redefine an existing variable *xyz* as a function, it's not sufficient to define *xyz* as a function, but must also be explicitly assigned to the variable *xyz*. This is unlike in Python.
- A task from task queue will be pushed to call stack, only if the call stack is empty. This is the run-to-completion semantics in JS.
- How event loops work in JS? See https://vimeo.com/96425312

Week3

- System state and Application state are examples of persistent state/data
- State of the page which is currently visible to the user is an example of ephemeral state.
- Application state is everything that is present in the memory when the app is running.
- In Imperative programming, all steps involved in solving the problem needs to implemented.
- In Declarative programming, only what to do needs to be declared.
- The system state is typically a huge collection of data.
- The user interface is dependent on the application state.

Week4 (Vue.js)

- @click can be used as a shorthand for v-on:click during event binding.

- Computed property is generally derived from reactive properties of Vue instances, and are themselves reactive.
- Computed properties are by default *getters*.
- Computed properties are cached based on their reactive dependencies.
- Computed properties should not be directly mutated.
- Watcher is a function which is triggered when the property it refers to changes. The parameters of a *watch* function are *new_value* and *old_value* in the same order.
- An element with *v-show* directive is always rendered irrespective of the condition's truth value.
- When *v-if* and *v-for* is used on the same element, *v-for* is evaluated first.
- *ref* in Vue allows us to directly reference the DOM element, *and* can be accessed only after the element is mounted.
- Vue components must be named when defining it using Vue.component construct, or need to be registered with the parent object (See 158.js in scripts/Quiz)
- While defining data for the Vue components, it needs to be returned, rather than defined as an object.

```
const comp1 = {
    template: `<h4>This is {{name}}</h4>`,
    //Wrong way of defining component data
    data: {
        name: 'component1'
    },
    //Right way of defining component data
    data() {
        return {
            name: 'component1'
        }
    }
}
```

- *v-bind* directive is used for one way data binding.
- *v-model* directive is used for two-way data binding.

Week5

- *BeforeCreate* event gets called before the data, methods and watchers are setup.
- *Created* event gets called after the above are setup, but before the instance is mounted. *el* will not be available during this event.
- *BeforeMount* event gets called after the *el* is available, but before mounting process is completed.
- Lifecycle hooks can't be handled using arrow functions, but full functions.
- It's typical to *fetch* data from backend during *Mounted* event, but it can also be done during *Created* event. These events can be handled in async.
- The lifecycle hooks are triggered implicitly, depending on the state of the component.
- All events like mouse or button clicks are asynchronous. Apart from these, there're certain JavaScript functions like *setTimeOut* that're inherently asynchronous.
- Calling a function with an *async* prefix on its definition, delivers a *Promise* object. An *async* function implicitly returns a *Promise* that resolves with the value returned by the function. If the function throws an error, the *Promise* is rejected with the thrown error as its value.
- *Promise* is a proxy for some value not necessarily known during creation. *Promise* always runs asynchronously.

- *Promise* accepts a single parameter – a tuple with two callback functions. The first callback is called when the promise is resolved, and second is called when the promise is rejected
- *Promise* can be in any of 3 states = accepted, rejected or pending
- Using a *then* keyword on a Promise object, will resolve it (using the resolve_handler function passed its first/only parameter). But, the timing of resolution isn't predictable. Optionally, it can also take a reject_handler function as its second parameter.
- Catching an error from a *Promise* must be done using *.catch* syntax.
- The keyword *await* can only be used inside *async* functions, except Chrome/Firefox browser console.
- The promises in JavaScript can be resolved synchronously, but are designed to work asynchronously.

```
<script>
    const promise = Promise.resolve(42);
    promise.then(value => console.log(value)); // This will be executed synchronously
    console.log("This will be logged before the value of the promise");
</script>
```

- Here is a general technique to solve problems related to promises and async/await.
    - Follow path of execution. Look for the main program. Ignore function definitions.
    - The default path of execution inside a function or even a promise is always synchronous. This is true even when the function is defined as *async*.
    - When execution reaches *resolve/reject* phase of a promise, it skips and executes the next line. If *resolve/reject* occurs inside a function, it skips all remaining lines in the function.
    - setTimeOut inside an async function makes it asynchronous.
- *Fetch* method returns a promise object that's guaranteed to resolve, unless there's a network error. Resolution can be HTTP errors though.
- *Though fetch* API supports the use of the async/await syntax, which allows you to write asynchronous code in a synchronous-looking style, the underlying network request is still processed asynchronously.
- *Fetch* method has two parameters = *URL*, and an *init* object.
- *Cookies* are key/value pairs used by website servers to store state information on the browser. Typically, when the server responds to the client requests, it instructs the client to stores the cookies.
- Client automatically sends cookies in subsequent requests to the server so that the server is aware of the client state. The option "credentials : 'omit'" ensures that no cookies are sent with the request.
- A static method cannot be invoked by a class object in JavaScript.
- A child class constructor must call the parent class constructor to instantiate the child class.
- A function that accepts another function as its parameter, or one that returns another function is called a higher-order function
- *XMLHttpRequest*, *Fetch*, *SetTimeOut* are all examples of asynchronous APIs
- *Finally* block doesn't take an input, or return anything.
- *Catch* block catches exception raised from the previous block. If the previous block doesn't raise an exception, it'll be skipped.
- If *then* block does not handle rejection of a promise, *catch* block can handle it optionally.

Week6

- *LocalStorage* as well as *SessionStorage* are client-side storage mechanisms and local to a specific browser. Data is stored as key-value pairs in string format in both cases.
- The data stored by a specific domain in local storage of the browser cannot be directly accessed by its subdomain.
- Data stored in *LocalStorage* persists across app refreshes and even machine restarts. Data stored in *SessionStorage* survives page refreshes, but expires when page session ends.
- Use *novalidate* to prevent form validation by the browser
- SFC allows *scoped* CSS(<style>), component modularization (<script>)and *pre-compiled* templates.
- Jest, Chai, Mocha are some of the tools used to test Vue.
- Vuelidate and VeeValidate are libraries to perform Form validation with Vue.
- Browser provides simple validations for certain type of input elements. Server side validation must be performed, even when such client-side validation exists.
- Client side storage can be used to store web-generated documents for offline use, or personalized site preferences.

Week7

- *Props* are used to pass information from parent to child.
- *Events* are emitted to pass information from child to parent.
- Child can directly invoke parent function and modify parent data using *$parent*
- An SPA fetch and update parts of the DOM asynchronously as and when needed. It helps to improve speed of transaction on page, page loading speed etc.
- When the server is "thin", all states and updates are handled on the browser using JavaScript, and gives pure data API service.
- *Flux, NgRx and Redux* are state management solutions. *NgRx* is used by Angular applications, and *Flux/Redux* is used by React applications.
- *Vuex* is a *reactive* state management solution used in Vue applications. It provides a common solution for parent-child (components) communication. *Vuex* is preferred in cases where multiple views depend on the same state.
- *State* in a *Vuex* should be changed only through committing mutations, from within a method. *Mutations* are always synchronous.
- When state access/changes need to be asynchronous, use *Actions*.
- *Getters* in a *Vuex* is similar to computed properties, but limited to the *Vuex* store. Getters shouldn't change state.
- Store state can be accessed as *this.$store.state*, by all child components including its descendants.
- Vue DevTools lists all mutations requested, time of mutations, and which component requested mutation. It also records mutations and allows time travel debugging.
- Target path in router-link can be set using *to* attribute, which will render it as a hyperlink tag. To render it as any other tag, use *tag* attribute.
- *router-view* renders the component matched by the path in router-enabled app. *router-link* enables user navigation in router-enabled app
- SPA loads only a single web document and then updates body content asynchronously/dynamically as and when required.This results in improved speed of transaction on page, page loading speed etc.

- Web Manifests, service workers are some characteristics of a PWA.
- A Web worker is a script started by a web content that runs in the background, and can perform computation and fetch requests and communicate back messages to the origin web content.
- Though most PWAs are implemented as SPAs, they're fundamentally different. Not all PWAs are SPAs, and not all SPAs are PWAs.
- Search engine optimization, managing browser history are some challenges of SPAs.

Week8

- CORS allow servers to indicate which origins can load the resource from the server. For example, Flask APIs should be CORS-enabled to be used by a client. Note that CORS is disabled in Flask by default. However, it's not required if the request comes from the same origin.
- The CORS headers are generally prefixed with the value "Access-ControlAllow".
- JWT contains encoded JSON data and is used as an access token to share information between two parties.
- *PUT* APIs update the entire resource is updated, whereas *PATCH* APIs update only the necessary fields
- An HTTP method is *idempotent* if the intended effect on the server of making a single request is the same as the effect of making several identical requests. Examples include GET, PUT, UPDATE, DELETE. POST is not idempotent.
- An HTTP method is *safe* if it doesn't alter the state of the server. In other words, a method is safe if it leads to a read-only operation. Examples include GET, HEAD, OPTIONS.
- All *safe* methods are also *idempotent*, but not all *idempotent* methods are *safe*. For example, PUT and DELETE are both idempotent but unsafe.
- JSX (JavaScript XML) is a syntax extension to JavaScript that allows writing HTML in React.
- Token Based, OAuth and JSON Web Token (JWT) are the common methods of API authentication.
- OAuth is a protocol used to authorize access to resources, hosted on a different server, on behalf of a user. See https://stackoverflow.com/a/33704657 and https://oauth.net/articles/authentication
- In the REST architectural style, both the client and the server should be stateless.
- Following are two issues with REST APIs, which are solved using *GraphQL*
    - Construction of complex queries requires client side handling.
    - When data is from multiple data sources, the fusion of data becomes an issue.
- *GraphQL* can execute a complex query to return a complex data set with a single request, whereas, REST APIs may require to make multiple requests for the same.
- *GraphQL* is a query language for APIs with a single-entry point. It allows a type system to describe the schema for backend data. It is not tied to any specific database or storage engine.
- *GraphQL* resolvers help resolve queries in any programming language, including Python, Java and C++
- *GraphQL* supports a *ContentType* called *application/graphql*, which is well suited to query the server for multiple resources. *GraphQL* also allows to make POST requests using JSON body.
- *GraphQL* helps to fetch exactly the same data which is needed, and avoids over-fetching as well as under-fetching.
- In REST architecture, GET requests are generally considered cacheable, but POST requests aren't.
- The response to a *GraphQL* request is a JSON response, with the result stored in the "data" field.

- *JAM Stack* stands for JavaScript, API and Markup. It takes storage, logic, and presentation into the consideration
- *JAM Stack* can't guarantee high performance and stability of the application, it improves them.
- Jam stack applications are faster and more secure.
- *Jekyll, Hugo and MkDocs* are a few common static site generators.
- A permalink is a permanent link which is not expected to change throughout the lifetime of a resource, and identifies a resource uniquely.

Week9

- Flask works in threaded mode by default, but can operated in non-threaded mode too.
- Multiple threads can execute concurrently or in parallel.
- Concurrency is achieved using context switching in a single-core environment
- In Parallelism, multiple threads can be executed at the same time on multiple processors.
- Web server can *dispatch* tasks to be executed later. Use it when response to the user doesn't need to depend on the outcome. Cannot be used in the case of API fetch task.
- In the case of a large number of long-running tasks, employ task queues. User request handler pushes tasks into task queues and handled in FIFO mode. Examples include Python asyncio, JS async/await.
- How does task queue work?
    - Client pushes a task in the queue, provided pushing it into the queue is faster than executing it.
    - Server *polls* the queue at regular intervals. This is CPU/network intensive.
    - Alternatively, server keeps connection open with the client until it receives a response, and does a *long-poll.*
    - There should be enough workers to empty the queue regularly, else backlog will build up.
    - It can also potentially lead to deadlocks and related issues.
    - Celery is a task queue implementation for Python web applications. It supports RabbitMQ, Redis, Amazon SQS and Zookeeper for message queueing.
    - Google AppEngine, Tencent cloud, AWS are some high-end providers of task queues.
- Message queues, brokers are systems that enable communicate between servers – that manage tasks. Messaging typically works in FIFO mode.
- Message queues provide a buffer that temporarily stores messages.
- In general, a message in message queue is processed only once in point-to-point messaging.
- Advantages with using a message broker include scalability (can add servers), guaranteed delivery of messages (though delayed), ease in performance monitoring, batch processing of messages.
- Advanced Message Queueing Protocol (AMQP) has many standard implementations like RabbitMQ and Apache ActiveMQ.
- Redis
    - Redis uses RESP protocol for communication between client and server, and uses 6379 as the default port.
    - Redis was originally designed as a caching mechanism in the server side.
    - Redis can be used to store messages (in RAM) and used in a pub-sub mode, especially for smaller messages. Redis can also be used as a NoSQL database.

    - Disadvantage of Redis is the volatility of its store, and loses messages when restarted.
    - Redis supports multiple datatypes
    - It's possible to set expiration time of data in Redis

Week10

- One of the techniques widely used for messaging is to send HTTP GET/POST requests to certain endpoints (called web-hooks) exposed by a server, which can in turn initiate further operations. Such endpoints are also referred to as reverse APIs, since they're used to push information, and not to retrieve data.
- A webhook allows two different applications to communicate with each other, over HTTP protocol.
- The primary consumers of webhooks are machines (servers), not humans.
- A webhook notifies the client every time an event occurs to which client has subscribed.
- For example, Twilio can send messages or emails, when events occur on the platform. In this case, Twilio provides you with an immediate response (status) and calls you back once on the configured web-hook after all messages has been sent.
- Client can pull latest information from the server. Alternatively, the server can push data to the client (server-sent events), as far as the following conditions are met
    - Client and server maintains a persistent connection between them.
    - Web worker is kept running on the client, in the background. This is possible through JS push API, or web push protocol.
    - It's not required that the application is loaded on a user agent (browser).
- WebSocket is a protocol layered over TCP, in which connection between client and server is always open. It's used for 2-way communication.

Week11

- Lighthouse built into the Chrome browser gives the following metrics:
    - First Contentful paint (Something displayed on the screen)
    - Speed Index (Captures video of page loading and analysis)
    - Largest Contentful paint ("Most of the page" rendered)
    - Time to Interactive (Usability)
    - Total Blocking Time (Time blocked from responding to user)
    - Cumulative Layout Shift (Rearrangement of page after initial load)
- Lighthouse score considers the following while generating a score:
    - Performance
    - Accessibility
    - Best practices
    - Search Engine Optimization (SEO)
    - Progressive Web App (if relevant)
- In general, lighthouse score is not a good measure of the site's usability, but can be used as a guideline
- The function of a load balancer is just to forward the requests to the server, using one of round-robin, least-load or some such algorithm.

- HTTP pipelining allows client to send multiple request to the server over a single TCP connection without waiting for their response
- While establishing a proxy between client and server, it's ideal to have one nearer to the client for faster response.
- CDN is another form of proxy, but encoded in the URL.
- Reverse proxy servers can cache the response based on the request.
- Tools used for live monitoring include:
    - ElasticSearch / LogStash / Kibana (ELK stack)
    - Grafana + InfluxDB + Prometheus
- Caching may occur at one of more of server, proxy front-end, network router or client.
- Shared caching and Local (private) caching are two types of caching, wherein the latter is results in faster response than the former.
- 'no-cache' directive in the response header indicates that the client must revalidate the response in every subsequent request.
- Flask caching using *cache.cached* decorator

```
@app.route("/")
@cache.cached(timeout=50)
def index():
    return render_template('index.html')
```

  The server response to the web-root(index.html) results in caching the response, and times out after 50 seconds.
- The *memoize* decorator includes the function parameters in the cache key.
- Both the *cache.cached* and *memoize* decorators work the same way for functions not having any parameters.
- Caching done at browser level can be cleared by doing a hard-refresh (Ctrl+F5 in Chrome). Note that this can still result in a cached response being served from a proxy server.
- E-Tag is a response header to validate the current version of resource.
- E-Tag is a string of ASCII characters placed between double quotes.
- For get and head request, If-Match request header is sent to get the resource only if E-Tag value matches.