המרכז
האוניברסיטאי
אריאל בשומרון

# Node.js & Service Based Software Development

**Yossi Zaguri**
**Yossiza@ariel.ac.il**
**052-4668866**

# Topics



- What is Node.js
- Node.js ecosystem
- IoC, event loop and Callback methods
- HTTP & Hello world
- Express.js & Web API development
- Implementing MVC & Front Controller Patterns
- Implementing Pipes and Filters Pattern
- Incorporating MongoDB/DBaaS

# Topics



- Implementing O.O & Functional paradigms
- O.O prototype concepts
- “this” binding methods
- JavaScript Closures
- Asynchronous programming methods
- Implementing Namespace & IIFE Patterns
- Introduction to Node.js internals & code optimization

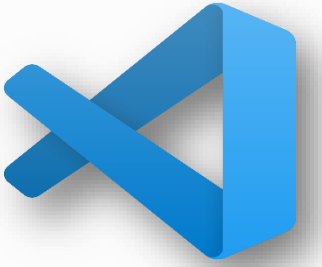
# Topics



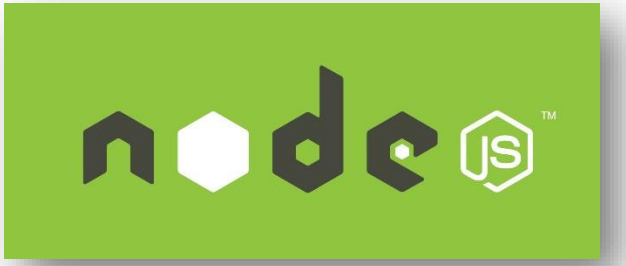
- Cloud & Service Orientation paradigm
- Web Services vs. Cloud Services
- In-Depth Micro Services Pattern
- Docker & building Node.js Image
- Serverless Development with Node.js
- Review of:
  - Message Broker
  - Gatekeeper
  - Gateway Routing
  - Gateway Aggregation
  - Event Sourcing
  - CQRS Patterns

# Before we start

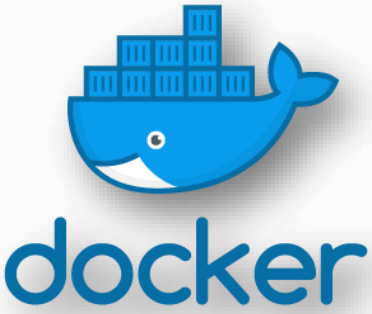




https://code.visualstudio.com/




https://nodejs.org/en/download/




https://docs.docker.com/desktop/windows/install/


https://www.mongodb.com/atlas/database

# Node.js

# VS Code and Node.js



https://code.visualstudio.com/docs/nodejs/nodejs-tutorial

# What is Node.js



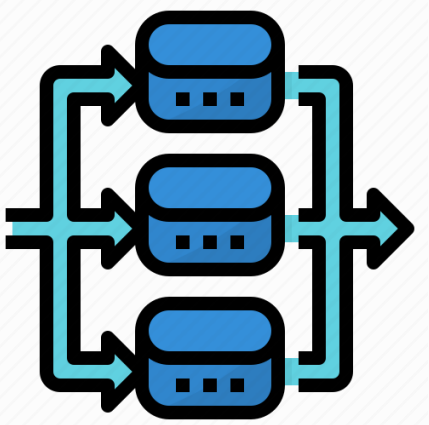
- Open-source & cross-platform JavaScript runtime environment, developed initially by Ryan Dahl.
- Runs the V8 JavaScript engine from Google Chrome.
- Runs in a single process, without creating a new thread for every request.
- Provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking.
- Libraries are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

# What is Node.js



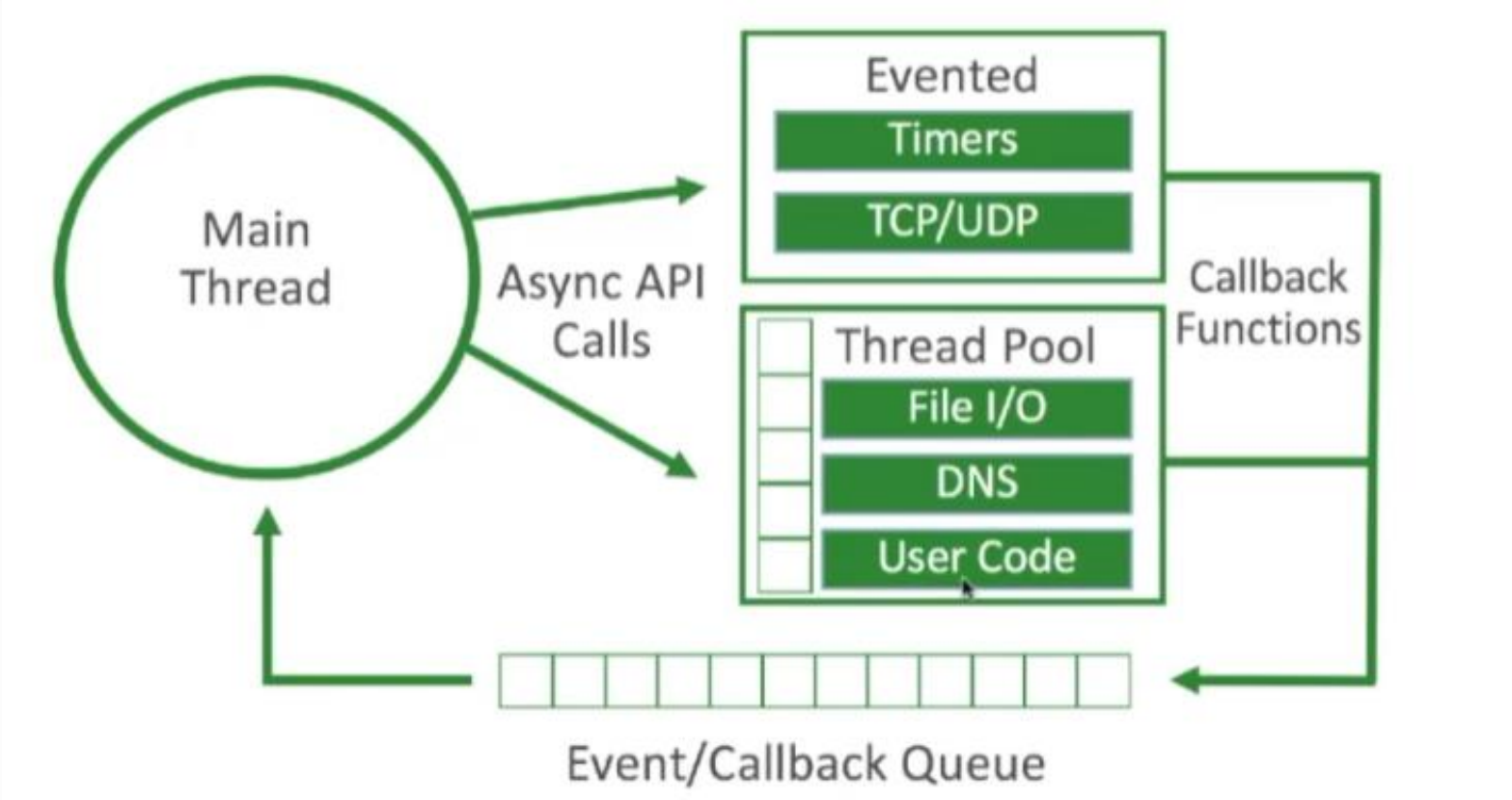
- I/O operations (network, access DB or FS), instead of blocking the thread and wasting CPU cycles waiting -> Node.js will resume operations ONLY when the response comes back.

- Thus Node.js can handle thousands of concurrent connections with a single server without the burdening programmers with thread concurrency management -> reducing this source of potential Bugs.



# What is Event Loop





# Event Loop & async

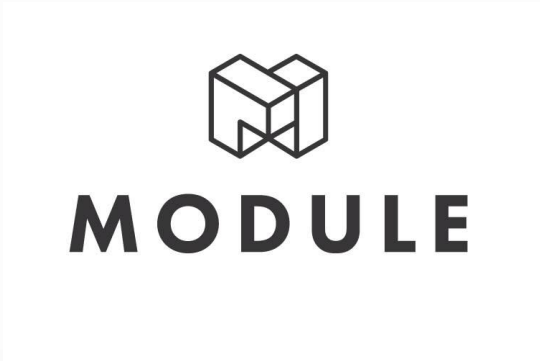


- When starting Node.js initializes an Event loop, allowing to perform non-blocking I/O operations despite the fact that it is single-threaded.
- Yet modern kernels are multi-threaded handling multiple operations by executing in the background.
- So Node.js Offers operations to the system kernel whenever possible
- When an operation completes, the kernel tells Node.js that the appropriate callback may be added to the **poll** queue to eventually be executed.

# Node.js Modules

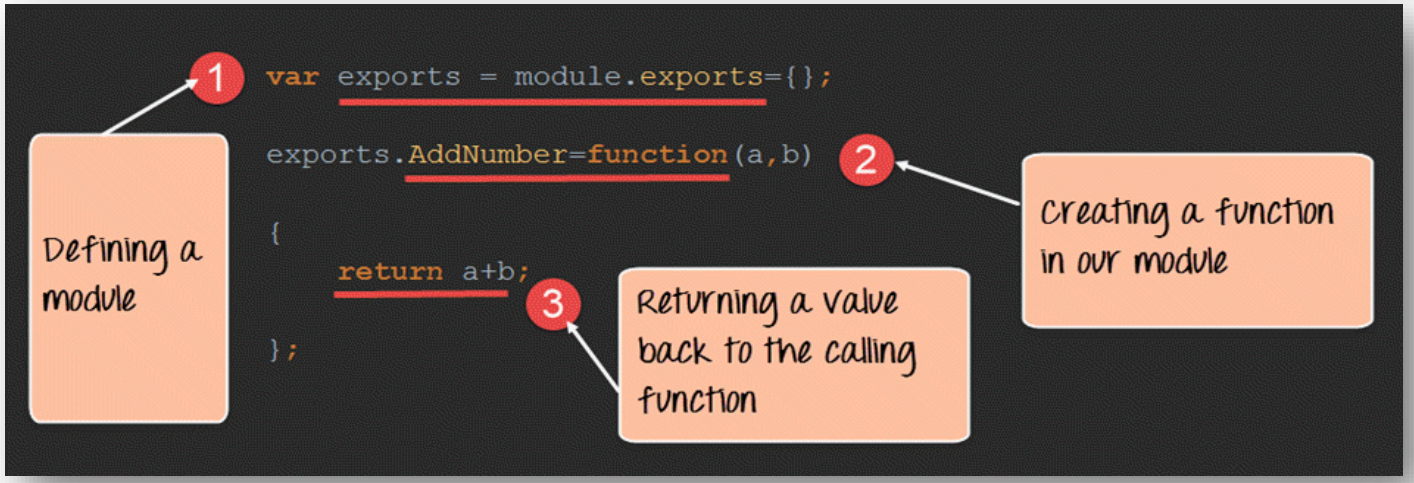


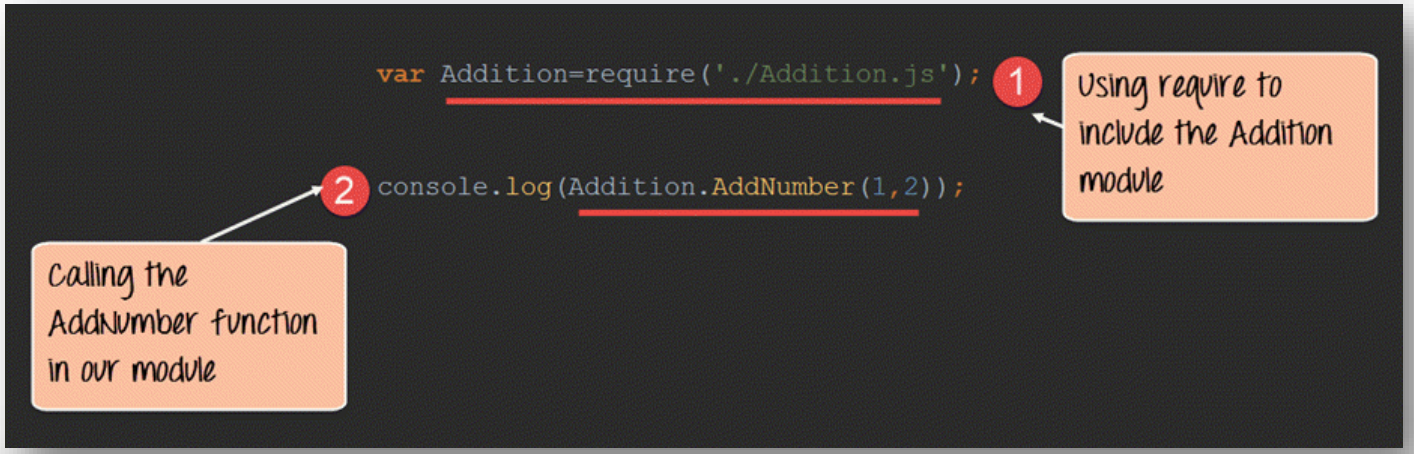
- In general : SoC Implementation via functions/classes/packages/sub systems etc.
- In Node.js : Logical (and Physical!) encapsulation of code in a single unit, managed as a separate unit of work.
- *Exports* a function or an object



# Export & Import Module





# Callback functions



- In general : An Implementation of IoC pattern
- Objects creation and the flow of the application delegated to a container or framework
- Increases the modularity of the program
- Leads to Loose Coupling between software parts (modules)
- Grate way to develop libraries or Infrastructure

```
var callback = function(data) {
  console.log('got data: '+data);
};

var doSomething = function(callback) {
  callback('simpler is better');
};

doSomething(callback);
```

המרכז
האוניברסיטאי
אריאל בשומרון

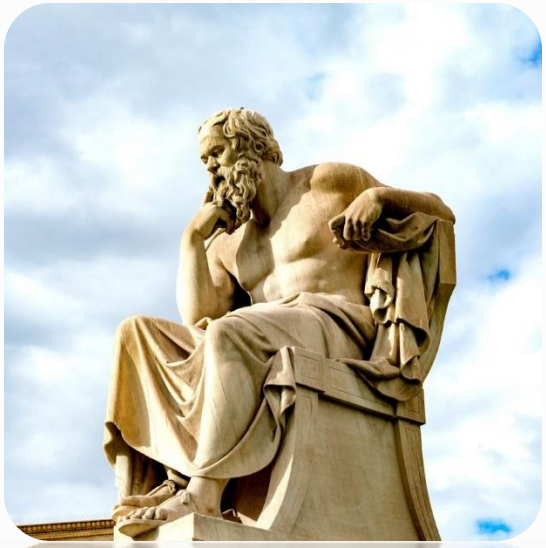
# Node.js Dev Spirit

# Node.js Philosophy



A set of principles and guidelines that are generally accepted by the community.

An ideology of doing things that influences the evolution of a platform, and how applications are developed and designed.



# Node.js Philosophy



- **Small core**
  - The rest is “User space” ->ecosystem of modules living outside the core
- **Small modules**
  - Code size & Scope
  - Physical *module* (file) is the fundamental means to structure the code
  - Reusable libraries called *packages* and typically have well-focused dependencies.
  - Solves “dependency hell” by making sure that each installed package will have its own separate set of dependencies, thus enabling a program to depend on a lot of packages without conflicts.
- **Consequences:**
  - Easier to understand and use
  - Simpler to test and maintain
  - Perfect to share with the browser
  - Boosting DRY (Don't Repeat Yourself)

Small is beautiful

Make each module do only one thing, yet do it well

# Node.js Philosophy



- **Focused Interfaces**
  - Node.js modules usually expose only a minimal set of functionality.
  - Usually expose only one piece of functionality, such as a function or a constructor
  - API becomes clearer to use and is less exposed to erroneous usage
  - Advanced aspects or secondary features become properties of the exported Entity, helping the user to identify what is important and what is secondary.
- **Modules usually are created to be "Aggregated" rather than extended**
- **Simplicity and pragmatism**
  - Takes less effort to implement
  - Allows faster shipping with less resources
  - Easier to adapt, understand and to maintain

"Favor Aggregation over Inheritance" (GOF)

"Simplicity is the ultimate sophistication." (Leonardo da Vinci)

# Managing Packages

# NPM



- Npm is the package manager for Node. Registry hosts over 1,000,000 open-source packages.
- CLI tool that aids installing packages and manage versions & dependencies.
- Install using npm : npm install <package>
- Example : **npm install colors**
- Npm is also used to manage metadata about a project in a file named package.json
- To create package.json : **npm init**
- To run Package: **npm run package** (later)

# Let's Try it

המרכז האוניברסיטאי אריאל בשומרון

```
var colors = require('colors/safe');

console.log(colors.green('hello')); // outputs green text
console.log(colors.red.underline('i like cake and pies')) //  red underlined text
console.log(colors.inverse('inverse the color')); // inverses the color
console.log(colors.rainbow('OMG Rainbows!')); // rainbow
console.log(colors.trap('Run the trap')); // Drops the bass
```

# HTTP & REST

# HTTP



- Client-server network protocol that made World-Wide Web what it is.
- Application-Level Protocol
- Peer-to-Peer communication
- Message Based (textual cross platform information units)
- Standard & Minimal Request types (called http verbs or method including :get , post, put, delete etc.)

# HTTP Request & Response



```
GET /httpgallery/introduction/ HTTP/1.1
Accept: */*
Accept-Language: en-gb
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: www.httpwatch.com
Connection: Keep-Alive
```

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/8.0
Date: Mon, 04 Jan 2015 12:04:43 GMT
X-Powered-By: ASP.NET
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache, no-store
Expires: -1
Content-Type: text/html; charset=utf-8
Content-Length: 14990

<!DOCTYPE html>  <html>...
```

# HTTP hello world

המרכז האוניברסיטאי אריאל בשומרון

```
const http = require('http')

const hostname = '127.0.0.1'
const port = 3000

const server = http.createServer((req, res) => {
  res.statusCode = 200
  res.setHeader('Content-Type', 'text/plain')
  res.end('Hello World\n')
})

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`)
})
```

Whenever a new request is received, the request event is called, providing two objects a request (an http.IncomingMessage object) and a response (an http.ServerResponse object).

# REST



- A set of architectural Principles/constraints to apply in network-based systems design
- Revolved around the concept of “Resource”
- Principles :
  - Give every “thing” an ID
  - Link things together
  - Use standard methods
  - Resources with multiple representations
  - Communicate statelessly
- RESTful API is one of the most popular Web Services Interface type

# Express.js

# What is Express.js



- Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
- With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy.
- Many popular frameworks are based on Express.

# Express “hello world”



```
md myapp
```

```
cd myapp
```

```
npm init
```

```
npm install express
```

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

# Live Demos



https://expressjs.com/

# MVC

# What is MVC



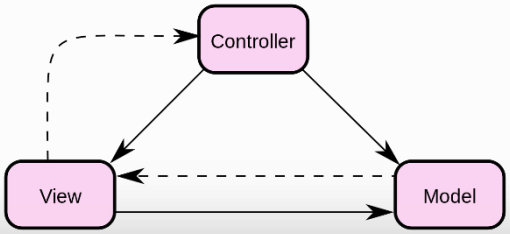
- Model-View-Controller (MVC) is one of the most popular architectural patterns (Is it an architecture indeed ?)

- We can describe the MVC architecture in simple terms:

  - *Model* : The **Module** that deals with the database or any data-related functionality.

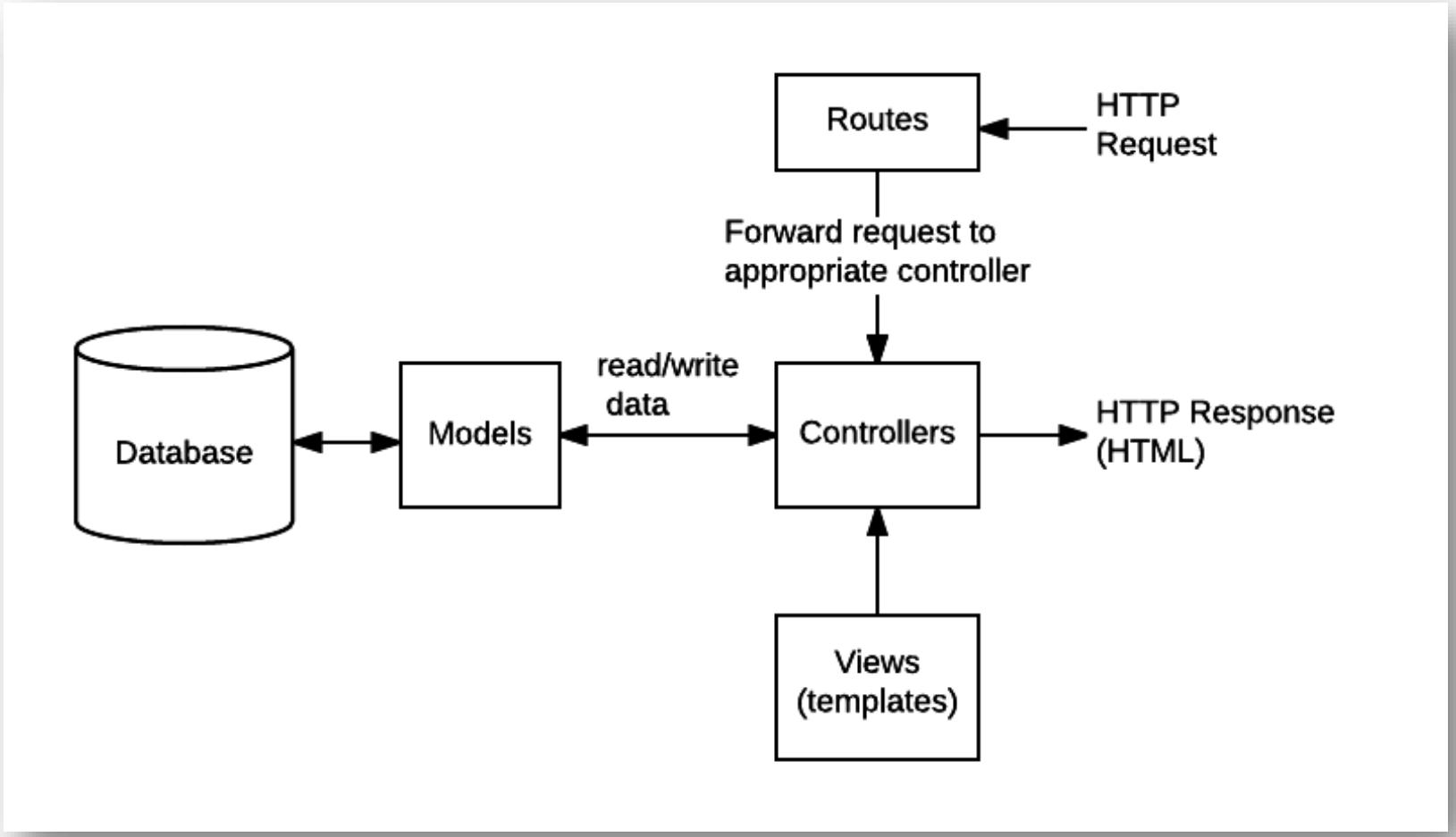
  - *View* : The **Module** responsible of the interaction/display with the user.
    In our case it will be a VIEW ENGINE.

  - Controller: The Module holding the main logic and managing execution flow . It will get a request, call models to get the data, then send the data to the view engine, getting html document to be sent to the user.



# Flow of processing





# Live Demos



https://expressjs.com/

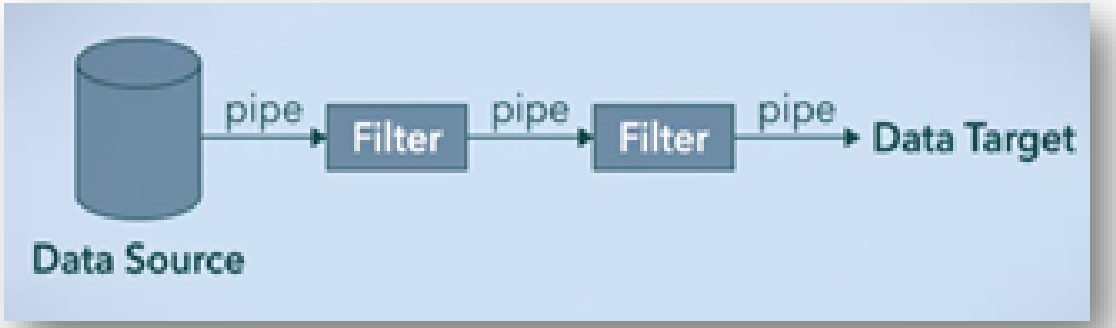
# Pipeline Pattern

# What are Pipes & Filters



- Architectural pattern having independent entities called *filters* (components) which perform transformations on data processing the input data, and *pipes*, which serve as connectors for the stream of data being transformed, each connected to the next component in the pipeline.

- Decompose a task that performs complex processing into a series of separate elements that can be reused. This can improve performance, scalability, and reusability by allowing task elements that perform the processing to be deployed and scaled independently.




https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

# Live Demos



https://expressjs.com/en/guide/writing-middleware.html

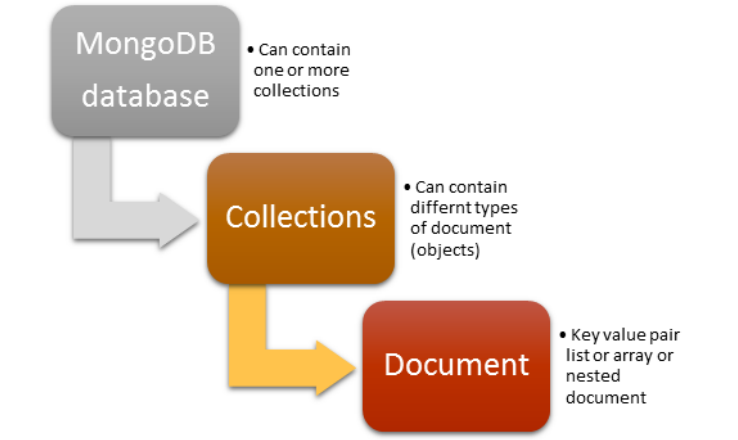
המרכז
האוניברסיטאי
אריאל בשומרון


# Implement Models using MongoDB

# What Is MongoDB



- (Very)Popular NoSQL DBMS.
- DB → Collection → Document
- Document is basically a json entity
- (Very) Highly Scalable




# Create DB



```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

To work using Schemas install mongoose instead of mongodb driver

# Create Collection



```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

Following w 3schools.com

# Create Collection



```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

Following w 3schools.com

# Insert Object



```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

Following w3schools.com

# Insert Array

המרכז האוניברסיטאי אריאל בשומרון

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = [
    { name: 'John', address: 'Herzelia'},
    { name: 'Peter', address: 'Ramat Gan'},
    { name: 'Amy', address: 'Arad'}
  ];
  dbo.collection("customers").insertMany(myobj, function(err, res) {
    if (err) throw err;
    console.log("Number of documents inserted: " + res.insertedCount);
    db.close();
  });
});
```

Following w3schools.com

# The Result Object



```
{
  result: { ok: 1, n: 3 },
  ops: [
    { name: 'John', address: ‘Herzelia'},
    { name: 'Peter', address: ‘Ramat Gan'},
    { name: 'Amy', address: ‘Arad’},,
],
  insertedCount: 3,
  insertedIds: [
    58fdbf5c0ef8a50b4cdd9a84,
    58fdbf5c0ef8a50b4cdd9a85,
    58fdbf5c0ef8a50b4cdd9a86
 ]
}
```

Following w3schools.com

# Data Query



```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var query = { address: “Arad" };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Following w3schools.com

# Data Projection

המרכז האוניברסיטאי אריאל בשומרון

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}, { projection: { _id: 0, name: 1, address: 1
} }).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Following w3schools.com

# Delete Data



```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: /^O/ };
  dbo.collection("customers").deleteMany(myquery, function(err, obj) {
    if (err) throw err;
    console.log(obj.result.n + " document(s) deleted");
    db.close();
  });
});
```

Following w3schools.com

# Update Data

המרכז האוניברסיטאי אריאל בשומרון

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myquery = { address: “Arad" };
  var newvalues = { $set: {address: “Negev" } };
  dbo.collection("customers").updateOne(myquery, newvalues, function(err,
res) {
    if (err) throw err;
    console.log("1 document updated");
    db.close();
  });
});
```

Follow ing w 3schools.com

# Live Demos



https://zellwk.com/blog/crud-express-mongodb/

המרכז
האוניברסיטאי
אריאל בשומרון

# JS – Intro & Programming Paradigms

# Built In Types - primitive types



- Boolean
  - false => 0, -0, null, false, NaN, undefined , empty string ("")
  - true => everything else , including “false” string or any object
- Number
  - 64-bit floating point
  - No integer type
  - Includes special values NaN & Infinity
- String
  - No character type
  - Literals quoted using the ' or "
  - Immutable
  - Escapement is done with the “\” character

# Built In Types - Primitive types



- Null
  - The Null type has exactly one value: null
- BigInt
  - Represent integers with arbitrary precision
  - Safely store and operate on large integers even beyond the safe integer limit for Numbers (Number.MAX_SAFE_INTEGER)
  - `const y = x + 1n => 9007199254740993n`
- Symbol
  - Unique and immutable primitive value
  - `let Sym1 = Symbol("Sym")`
  - Must cast in order to treat as string => `console.log(Sym.toString())`
  - `Symbol.keyFor(Symbol.for("tokenString")) === "tokenString" // true`

# Built In Types - Special Values



- Special values
  - null – An object
    - value represents a reference that points to a nonexistent or invalid object or address.
    - null is not as "primitive" as it first seems although its behavior is seemingly primitive: Every Object is derived from null value, and therefore typeof operator returns object : -> (typeof null) === 'object' // true
  - undefined - a privative , it's type is Undefined
  - undefined is the lack of a type and value, null is the lack of a value
  - Common conceptual Mistake
    - if (foo == null) { alert('foo really has no type. Object imposed'); }
    - if (foo === undefined) { alert('foo has undefined value.'); }
- Everything else is variations on object type:
  - Dates, Regular Expressions & More added by engine implementation These are really just objects

# Language Concepts - Global Object



- The global object is essential, and you can't avoid it.
- It contains things available on every module like require, exports, and __dirname
- It also contains timer functions like setTimeout and setInterval as well as your favorite, the console
- On the browser, you create a global variable(with var) which will be available everywhere (global scope)
- With NodeJs things are different*, the top level is the module itself*, any "global" variable you create is local to the module and you choose what the module exports.

# Language Concepts - Hashtable



- JS is loosely typed → don't use type names in declarations
- Define Hash Table:
  - var x = {};
- Subscript notation to add, replace, retrieve elements:
  - x["name"] = “Rina Levi";
- Dot notation is more convenient :
  - x.city = “Tel Aviv";
- Can be used when subscript is string constant in form of a legal identifier
- Because objects & hashtables are the same, this is Equal to above :
  var x = new Object()

**Object ↔ Hashtable ↔ Array**

# Language Concepts - Hashtable



- Member enumeration capability *built* into the **for..in statement**

```
for (var n in x) {
  if (x.hasOwnProperty(n))
        alert(x[n])
}
```

# Language Concepts - Object



- Object → Referenceable container of name/value pairs.
- Object literal notation: object description with a set of comma-separated name/value pairs inside curly braces
  - Names can be identifiers or strings followed by a colon
  - Values can be literals or expressions of any type

```
var Obj = {name: “Rina", ‘last': ‘Levi', city: ‘TA', level: 20};
```

- New members can be added to any object any time by assignment:
  Obj.nickname = ‘Rinale';

**Object ↔ Hash Table ↔ Array**

# Language Concepts - Arrays



- Two ways to make a new array:
  - var myArray = []
  - var myArray = new Array()
- Arrays implemented as objects
- Arrays are not typed.
  - Can contain numbers, strings, booleans, objects, functions & arrays
- You can mix strings, numbers & objects in the same array
- Arrays literal notation:
  - myList = ['oats', 'peas', 'beans', 'barley'];
  - slides = [ {url: 'slide1.html', title: 'First'}, {url: 'slide2.html', title:'Second'}];

**Object ↔ Hash Table ↔ Array**

# Language Concepts - Arrays



- New item can be added by assignment:
  - x[2]=30
  - a[i + j] = f(a[i], a[j])
  - var myarray=[x+y, 2, Math.round(z)]
- Undefined values allowed :
  - var myarray=[0,,,,5]
- Multi Dimensional Array:
  - var myarray=[["New York", "LA", "Seattle"], China, Japan]
  - myarray[0][1] -> returns "LA"

**Object ↔ Hash Table ↔ Array**

# Language Concepts – Function



- *Function* = constructor
  - var Average = new Function("x", "y", "return (x + y)/2");
  - Less efficient than declaring function
  - Actually Average is An object that has a value
  - Function object can be called just as if it were a function by specifying the variable name

**Function != function**

# Language Concepts - function



- *function* = operator
  - Declare block of code as in “other” languages
  - You CAN pass a fixed number of parameters
  - Excess parameters are ignored
  - function has access to “arguments” array
  - functions are First class objects
  - Functions have lexical scoping - inner level can access outer levels
  - Can hold there own Data & Methods , acting like a class
  - Inner functions (Closures) & anonymous functions EXIST
  - Every function returns a value. default “undefined”, constructors=> this

**Function != function**

# Language Concepts – prototype



- Objects contain a hidden link property pointing to the prototype member of the constructor of the object
- Prototype link chains:
  - When items accessed from object by dot notation or subscript notation, if item not found in the object then the link object is examined all the way up, returning “undefined” if not found
  - Provides sort of inheritance
- Members can be added to the prototype by assignment

```
function Demo() {}
Demo.prototype = new Ancestor();
Demo.prototype.foo = function () {};
```

# Language Concepts - this



- sayHello()
  - this => global object
- person.sayHello()
  - this=> person
- new person()
  - this=> the new object
- Arrow functions *Does not* have its own bindings to *this* (or super)
- Use call / apply / bind methods to adjust referencing

```
const materials = [  'Hydrogen',  'Helium',  'Lithium',  'Beryllium']

console.log(materials.map(material => material.length));
```

# Language Concepts – closure



- An expression (typically a function) that can have free variables together with an environment that binds those variables

- Using the *function* keyword inside another function creates a closure !

```
function sayHello(name) {
  var text = 'Hello ' + name;
  var sayAlert = function() { alert(text); }
  sayAlert();
}
SayHello("rina");
```

# Language Concepts – closure



```
function sayHello (name) {
  var text = 'Hello ' + name; // local variable
  var sayAlert = function() { alert(text); }
  return sayAlert;
}
var sayHelloAgain=sayHello('rina');
sayHelloAgain();
```

```
function sayHello(name) {
  var text = 'Hello ' + name;
  var sayAlert = function() { alert(text); }
  sayAlert();
}
```

# Language Concepts – closure



```
function setupSomeGlobals() {
    // Local variable ends up within closure
    var num = 100;
    // Store references to functions as global variables
    gAlertNumber = function () { alert(num); }
    gIncreaseNumber = function () { num++; }
    gChangeNumber = function (x) { num=x; }
}
```

```
setupSomeGlobals();
gChangeNumber(200);
gAlertNumber();
gIncreaseNumber();
gAlertNumber();

setupSomeGlobals();
gAlertNumber();
gIncreaseNumber();
gAlertNumber();
```

num is shared among all functions
Each call created new set of variables ( closure )

# Language Concepts – closure



```
function newClosure(someNum,  someRef) {
        var num = someNum;
        var anArray = [1, 2, 3];
        var ref = someRef;
        return function (x) {
            num += x;
            anArray.push(num);
            alert('num:' + num +',anArray:' + anArray.toString()
                  +',ref.someVar:' + ref.someVar);
        }
    }

    var firstClosure = newClosure(10, { someVar: 'first Closure' });
    var secondClosure = newClosure(20, { someVar: 'second Closure' });
    firstClosure(100);
    secondClosure(200);
```

No closure per function definition, rather per call
Each call creates new closure

# Language Concepts – closure



```
function buildList(list) {
        var result = [];
        for (var i = 0; i < list.length;  i++) {
            var item = 'item' + list[i];
            result.push(function  () { alert(item); });
        }
        return result;
    }
    function testList() {
        var fnlist = buildList([1,  2, 3]);

        for (var j = 0; j < fnlist.length;  j++) {
            fnlist[j]();
        }
    }
    testList();
```

```
function sayHello() {
        var sayAlert = function ()
                    { alert(Hello);  }
        var Hello = 'שלום';
        return sayAlert;
    }
sayHello()();
```

Loop Warning: I & item have only single copy
Variable declaration order not important

המרכז
האוניברסיטאי
אריאל בשומרון

# JS – Object Orientation

# Create Object – Literally



```
var person = {
    firstName:'Dani',
    lastName: 'Choen',

    //method
    getFunction : function(){
    return (`The name of the person is ${person.firstName} ${person.lastName}`)
    },
    //object within object
    phoneNumber : {
        mobile:'12345',
        landLine:'6789'
    }
}
console.log(person.getFunction());
console.log(person.phoneNumber.landLine);
```

# Create Object – Object Inheritance & Cloning



```
const coder = {
    isStudying : false,
    profession:'programmer',
    printIntroduction : function(){
        console.log(`My name is ${this.name}. Am I studying?
                    ${this.isStudying}.`)
    }
}

const me = Object.create(coder);

// "name" is a property set on "me", but not on "coder"
me.name = 'Dani';
me.isStudying = true;
console.log(me.profession);
```

```
let objClone = { ...obj }; // pass all key:value pairs from an object
```

Object.create() method creates a new object, using an existing object as the *prototype* of the newly created object.

# Create Object – Constructor



```
//using a constructor
function person(first_name,last_name){
   this.first_name = first_name;
   this.last_name = last_name;
}
//creating new instances of person object
let person1 = new person('Dani', 'Choen');
let person2 = new person('Rahul',Ganon');

console.log(person1.first_name);
console.log(`${person2.first_name} ${person2.last_name}`);
```

# Create Object – Class



```
class Vehicle {
  constructor(name, maker, engine) {
    this.name = name;
    this.maker =  maker;
    this.engine = engine;
  }
  getDetails(){
      return (`The name of the bike is ${this.name}.`)
  }
}
// Making object with the help of the constructor
let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');


console.log(bike1.name);    // Hayabusa
console.log(bike2.maker);   // Kawasaki
console.log(bike1.getDetails());
```

Unlike other Object Oriented Language there is no classes in JavaScript - only Object. JavaScript is a prototype based object oriented language, which means it doesn't have classes rather it define behaviors using constructor function and then reuse it using the prototype.
It is primarily syntactical sugar over JavaScript's existing prototype-based inheritance.

# Create Object – Class Inheritance



```
class person{
    constructor(name){
        this.name = name;
    }
    // method to return the string representing the object
    toString(){
        return (`Name of person: ${this.name}`);
    }
}
class student extends person{
    constructor(name,id){
        //super keyword to for calling above class constructor
        super(name);
        this.id = id;
    }
    // function override
    toString(){
        return (`${super.toString()},Student ID: ${this.id}`);
    }
}
let student1 = new student('Mukul',22);
console.log(student1.toString());
```

# Create Object – Static Constructor



```
class ClassWithStaticInitializationBlock {
static staticProperty1 = 'Property 1’;
static staticProperty2;
static { this.staticProperty2 = 'Property 2';  }
}


// output: "Property1“
console.log(ClassWithStaticInitializationBlock.staticProperty1);

// output: "Property 2"
console.log(ClassWithStaticInitializationBlock.staticProperty2);
```

# Module returning Object



```
// These variables will stay in the local scope of this module (person.js)
var firstName, lastName, age;

// Make sure your argument name doesn't conflict with variables set above
exports.setFirstName = function (fname) {
    firstName = fname;
};

exports.setLastName = function (lname) {
    lastName = lname;
};

exports.setAge = function (yrsold) {
    age = yrsold;
};

// You're returning an object with property values set above
exports.getPersonInfo = function () {
    return {
        firstName: firstName,
        lastName: lastName,
        age: age
    };
};
```

```
var person = require('./person.js');
person.setFirstName('Steve');
person.setLastName('Jobs');
person.setAge(56);

// Outputs first name, last name, and age as
// an object literal
 console.log(person.getPersonInfo());
```

# Getter & Setter



```
let obj = {
    get propName() {
      console.log("get")
    },

    set propName(value) {
        console.log("set")
    }
  };
```

```
let user =
{
    name: "John",
    surname: "Smith",
    get fullName()
    {
     return `${this.name} ${this.surname}`;
    },
    set fullName(value) {
     [this.name, this.surname]=value.split(" ");
    }
 }

user.fullName = "Alice Cooper"
console.log(user.name)
console.log(user.surname)
```

# Accessor descriptors



```
let user = {
    name: "John",
    surname: "Smith"
  };

 Object.defineProperty(user, 'fullName', {
    get() {
      return `${this.name} ${this.surname}`;
    },

    set(value) {
      [this.name, this.surname] = value.split(" ");
    }
  });

  console.log(user.fullName); // John Smith

  for(let key in user) console.log(key)
```

# Spread syntax (...)



```
function sum(x, y, z) {
    return x + y + z;
  }

  const numbers = [1, 2, 3];

  console.log(sum(...numbers));
  // expected output: 6
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

המרכז
האוניברסיטאי
אריאל בשומרון

# JS – Functional Programming

# Functional Programming

המרכז האוניברסיטאי אריאל בשומרון

- Functional Programming describes only operations
- No use of temporary variables to store intermediate results
- Emphasis is to capture "what and why" rather than the "how"
- Emphasizes definition of functions rather than the implementation of state machines, in contrast to procedural programming, which emphasizes execution of sequential commands.
- Knowledge Management systems benefit greatly as it simplifies development
- *Concepts*: Anonymous functions, different ways to call functions, pass functions as arguments to other functions.

# Functional Programming



*Classic*

```
function sum(x,y,z) {
return (x+y+z); }
```

*Anonymous*

```
function(x,y,z) {
return (x+y+z); }
```

- var sum = function(x,y,z) { return (x+y+z); }(1,2,3); console.log(sum);
- (console.log)((function (x, y, z) { return (x + y + z) })(1, 2, 3));
- function as an expression

- function calc(action,data) { console.log(action(data));} calc(function (x){ return x*2;},5);
  - function as first class object
- Hence:
  - Functions need not have names all the time.
  - Functions can be assigned to variables like other values.
  - A function expression can be written and enclosed in parentheses
  - Functions can be passed as arguments to other functions.

# Functional Programming

המרכז האוניברסיטאי אריאל בשומרון

*Too Static*

```
function printArray(array) {
for (var i = 0; i < array.length; i++)
 console.log(array[i]);
}
```

*More General*

```
function forEach(array, action)
 { for (var i = 0; i < array.length; i++)
   action(array[i]);
}
forEach(["rina", "dina],alert);
```

```
function sum(numbers) {
var total = 0;
forEach(numbers, function (number) {
      total += number; });
      return total;
}
console.log(sum([1, 10, 100]));
```

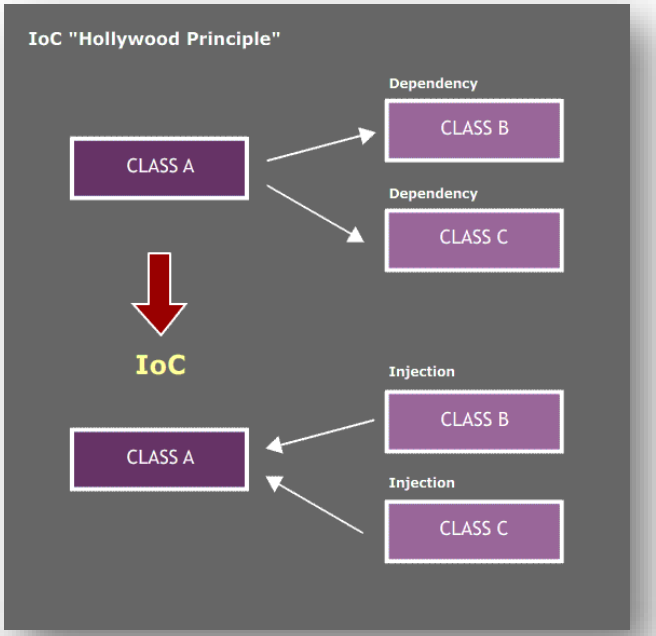
המרכז
האוניברסיטאי
אריאל בשומרון

# JS Async Programming

# IoC Programming Style



- Inversion of Control (IoC) is a design principle (some people refer to it as a pattern).
- Behavioral pattern that Reverses the “traditional” flow of control
- IoC helps in designing loosely coupled classes which make them testable, maintainable and extensible.
- Implementation examples :
  - Callbacks in general
  - Events
  - Observer/Mediator design patterns
  - DI Technique
  - Promises



# Using Event Emitter



- *emit* is used to trigger an event
- *on* is used to add a callback function that's going to be executed when the event is triggered

```
const EventEmitter = require('events')
const eventEmitter = new EventEmitter()
```

```
eventEmitter.on('start', number => {
  console.log(`started ${number}`)
})
```

```
eventEmitter.emit('start', 42)
```

# Using Event Emitter

המרכז האוניברסיטאי אריאל בשומרון

```
const express = require('express')
const other = require('./otherModule')
const app = express()
const port = 3000
other.setTheCb((seconds) => {
    console.log(`the session ended
after ${seconds} sec.`)
  });

app.get('/', (req, res) => {
  res.send('Hello World!')
  other.start(3000)
})
app.listen(port, () => {
  console.log(`listening at
http://localhost:${port}`)
})
```

otherModle.js

```
const EventEmitter = require('events')
const eventEmitter = new EventEmitter()


var setTheCb=function(cb)
{
    console.log("set cb")
    eventEmitter.on('endTimer',cb)
}

function start(seconds)
{
    console.log("counting")
    setTimeout(() =>
{eventEmitter.emit('endTimer',seconds)},
seconds);
}

module.exports={start:start,
                setTheCb:setTheCb
               };
```

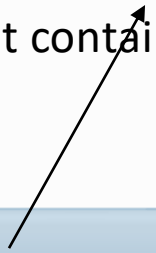
# Promises



- *A promise* is an object which represents the result of an asynchronous operation which is either resolved or rejected

- Promises are used to handle asynchronous operations. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create ***callback hell*** leading to unmanageable code

- The function passed to new Promise is called the executor. When new Promise is created, the executor runs automatically. It contains the producing code which should eventually produce the result.



```
let promise = new Promise(function(resolve, reject){
//do something });
```

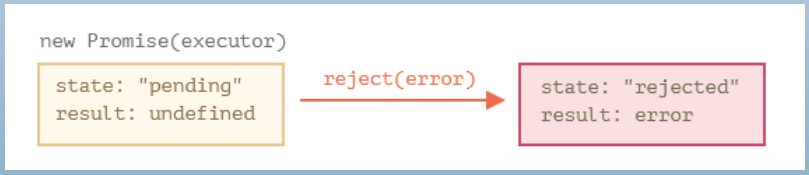
# Promises



- states:
  - Fulfilled: resolve() was called)
  - Rejected: reject() was called)
  - Pending: not yet fulfilled or rejected
  - settled: Promise has fulfilled or rejected

```
// the function is executed automatically when the promise is constructed
// after 1 second signal that the job is done with the result "done"
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done"), 1000);
});
```


```
// after 1 second signal that the job is finished with an error
let promise = new Promise(function(resolve, reject) {
setTimeout(() => reject(new Error("Whoops!")), 1000); });
```



# Promises

המרכז האוניברסיטאי אריאל בשומרון

```
let promise = new
Promise(function(resolve, reject) {
setTimeout(() => resolve("done!"), 1000); });


promise.then(
result => console.log(result),
error => console.log(error)
);
```

```
const myPromise = new Promise((resolve,
reject) => {
    if (Math.random() > 0) {
        resolve('Hello, I am positive number!');
    }
    reject(new Error('I failed some times'));
})
```

https://javascript.info/promise-basics

# async functions



- a function declared with the async keyword, and the await keyword is permitted within them.
- The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
function resolveAfter2Seconds() {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve('resolved');
      }, 2000);
    });
  }

  async function asyncCall() {
    console.log('calling');
    const result = await resolveAfter2Seconds();
    console.log(result);
    // expected output: "resolved"
  }

  asyncCall();
```

# async functions



- a function declared with the async keyword, and the await keyword is permitted within them.
- The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
function resolveAfter2Seconds() {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve('resolved');
      }, 2000);
    });
  }

  async function asyncCall() {
    console.log('calling');
    const result = await resolveAfter2Seconds();
    console.log(result);
    // expected output: "resolved"
  }

  asyncCall();
```

# Namespace & IIFE Patterns

# IIFE Pattern



- IIFE (Immediately Invoked Function Expression)
- Limit the number of global variables

```
(function ()
{ statements})();
```

```
(function ()
{ let firstVariable;
  let secondVariable;
})();
```

# Namespacing



- Namespaces can be found in almost any serious JavaScript application Unless we're working with a simple code.

- Common Patterns:
  - Single global variables
  - Prefix namespacing
  - Object literal notation
  - Nested namespacing
  - Immediately-invoked Function
  - Expressions
  - Namespace injection

# Revealing Module pattern

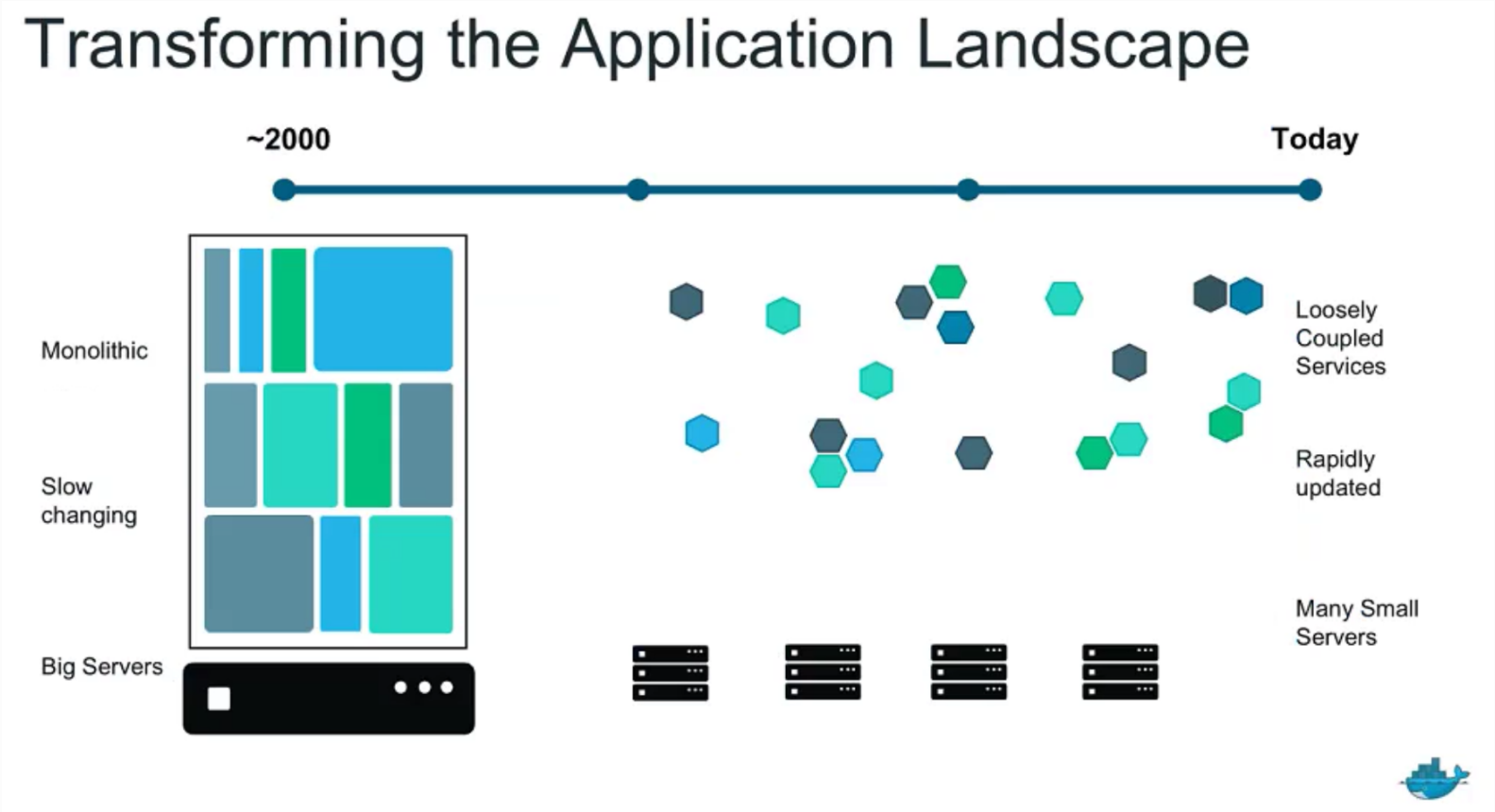
המרכז האוניברסיטאי אריאל בשומרון

```
var namespace = (function () {

    // defined within the local scope
    var privateMethod1  = function () { console.log("function 1") },
        privateMethod2  = function () { console.log("function 2") }
        privateProperty1 = "foobar";

    return {

        // the object literal returned here can have as many nested depths as we wish, however
        // this way of doing things works best for smaller limited-scope

        publicMethod1:  privateMethod1,

        // nested namespace with public properties
        properties:{
            publicProperty1: privateProperty1
        },

        // another tested namespace
        utils:{
            publicMethod2:  privateMethod2
        }

    }
})();

console.log(namespace.properties.publicProperty1)
```

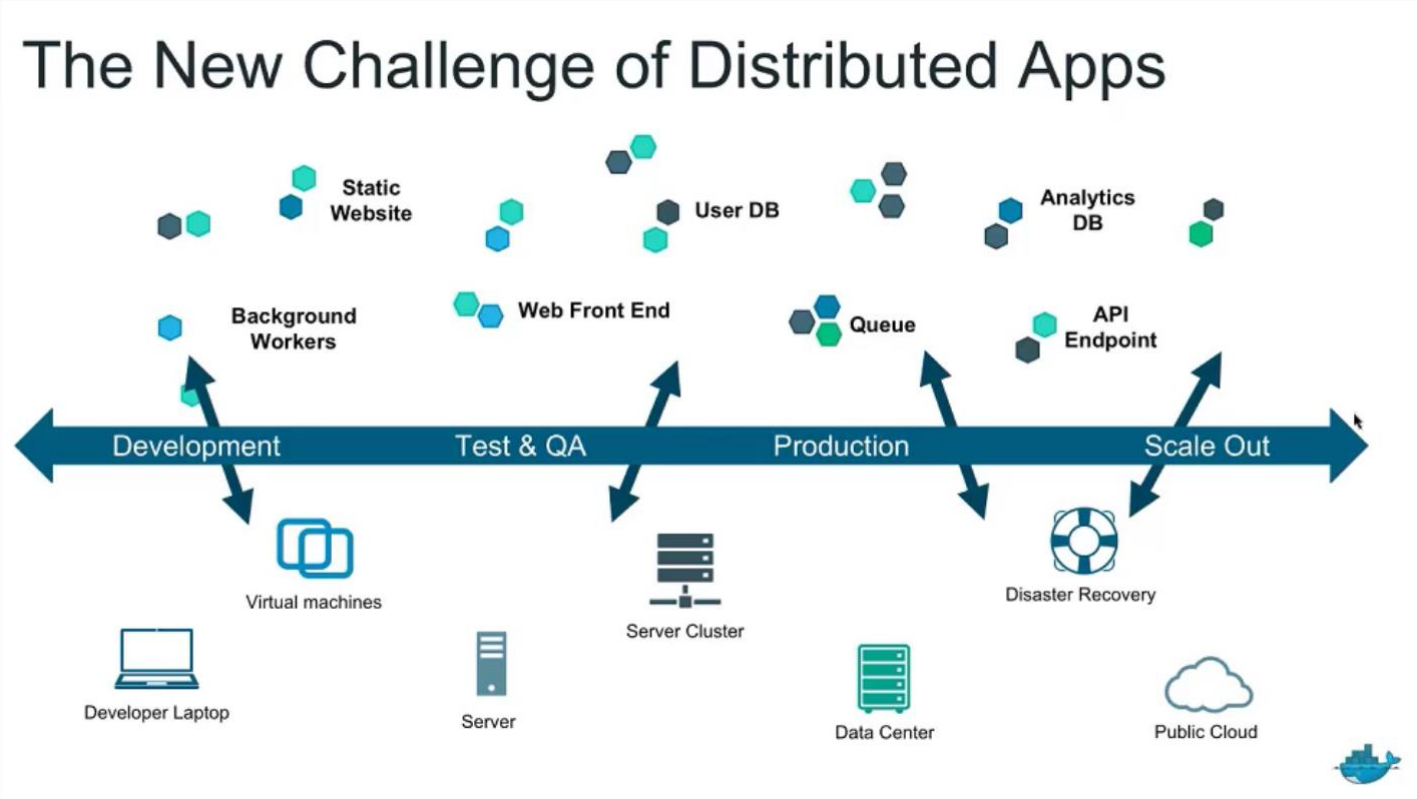
המרכז
האוניברסיטאי
אריאל בשומרון

# Service Orientation & Microservices Pattern

# Development Trends

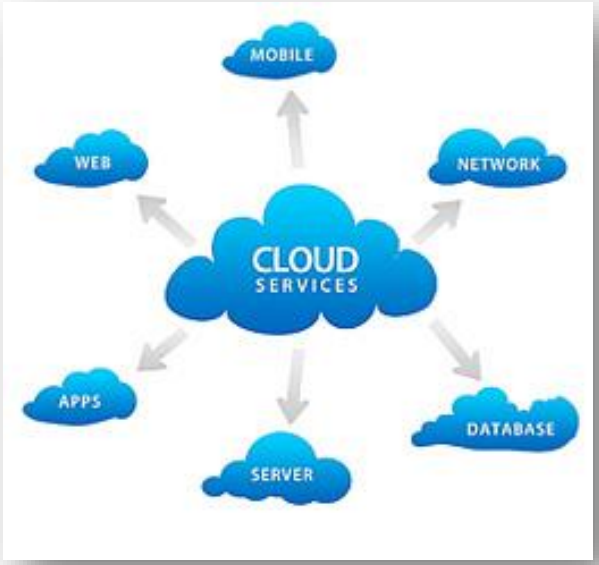


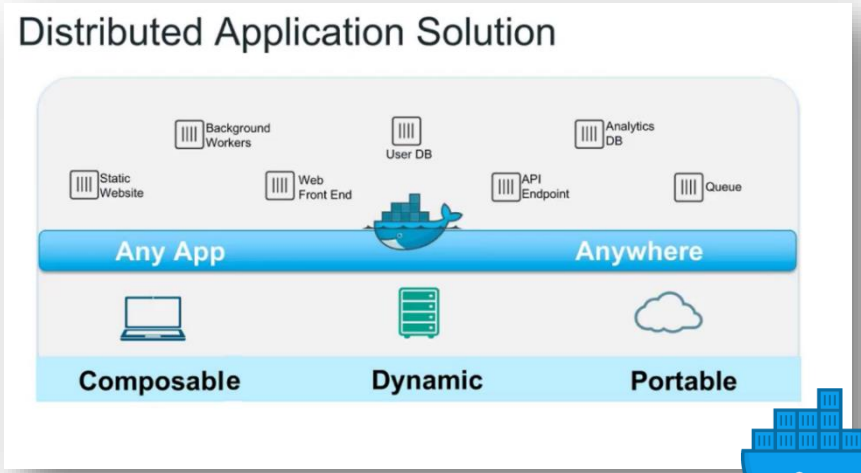
# Development Trends



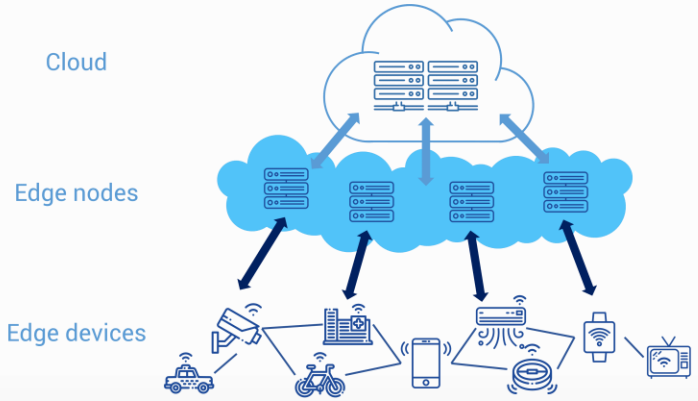
# Where Services Live ?






docker



# Software Services



- Several Granularity References (SaaS, PaaS, IaaS etc.)
- Our Focus: Functionality Units suppling Two types of Services
  - Compute
  - Storage
- The Successor of Software Component
- Loose Coupling between services
- Cross Platform
- Implementation Alternatives:
  - RESTFul API
  - gRPG
  - GraphQL
- Services can be implemented as Microservices

https://www.imaginarycloud.com/blog/grpc-vs-rest/

# Microservices Defined



- There is no formal definition of the term microservices

- Yet most microservice systems share a few notable characteristics:
  1) Software broken down into multiple component services
  2) Organized around business capabilities and priorities
  3) Simple Routing - microservices have smart endpoints that process info and apply logic, and dumb pipes through which the info flows.
  4) Decentralized & involve a variety of technologies and platforms
  5) Failure Resistant - microservices are designed to cope with failure
  6) Evolutionary by Design

# Microservices - Pros



- Microservice architecture gives developers the freedom to independently develop and deploy services
- A microservice can be developed by a fairly small team
- Code for different services can be written in different languages (though many practitioners discourage it).
- Easy integration and automatic deployment (using open-source continuous integration tools such as Jenkins, Hudson, etc.)
- Easy to understand & modify for developers, can help a new team member become productive quickly.
- The code is organized around business capabilities.
- When change is required in a certain part of the application, only the related service can be modified and redeployed—no need to modify and redeploy the entire application.
- Better fault isolation: if one microservice fails, the other will continue to work (although one problematic area of a monolith application can jeopardize the entire system).
- Easy to scale and integrate with third-party services.
- No long-term commitment to technology stack, Can adopt new one's easily.

# Microservices - Cons



- Due to distributed deployment, testing can become complicated and tedious.
- Increasing number of services can result in information barriers.
- The architecture brings additional complexity as the developers have to mitigate fault tolerance, network latency, and deal with a variety of message formats as well as load balancing.
- Being a distributed system, it can result in duplication of effort.
- When number of services increases, integration and managing whole products can become complicated
- In addition to several complexities of monolithic architecture, the developers have to deal with the additional complexity of a distributed system.
- Developers have to put additional effort implementing mechanism of communication between the services
- Handling use cases that span more than one service without using distributed transactions is not only tough but also requires communication and cooperation between different teams.
- Partitioning the application into microservices is very much an art.

# API's

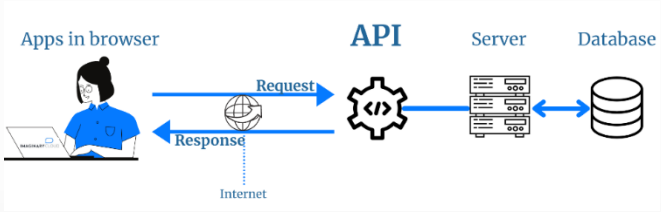


- **Monolithic application**: all the project's functionalities are included in a single unit, more precisely, in a single codebase.

- **Microservice architecture:** comprised of several smaller services that communicate with each other using protocols like HTTP.

- The component services that are part of the microservices architecture communicate and interact with each other through APIs.

- **APIs** allow all the services that are integrated into a microservice application to connect and communicate.
- Most used architectural style is the **REST API**. Yet there are three main models when building an API:
  - **RPC** (Remote Procedure Call)
  - **REST** (Representational State Transfer)
  - **GraphQL**

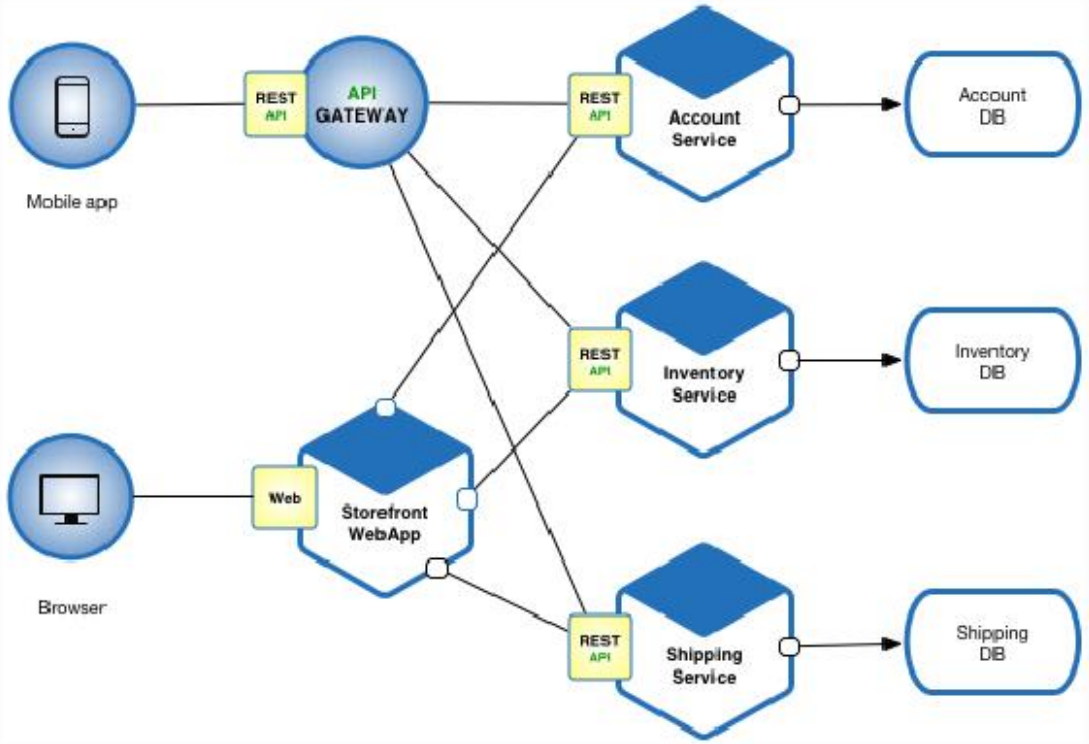


# API Gateway



API gateway is an API management tool that sits between a client and a collection of backend services.

An API gateway acts as a reverse proxy to accept all application programming interface (API) calls, aggregate the various services required to fulfill them, and return the appropriate result.




http://microservices.io/patterns/microservices.html

# Swagger & OpenAPI

המרכז האוניברסיטאי אריאל בשומרון

- Swagger is a software tool used for designing, building, documenting, and using RESTful APIs. It follows the OpenAPI specification

- OpenAPI specification is a specification used for creating interfaces used in describing, producing, consuming, and visualizing RESTful APIs.

```
module.exports = {
  // method of operation
  get: {
    tags: ["Todo CRUD operations"], // operation's tag.
    description: "Get todos", // operation's desc.
    operationId: "getTodos", // unique operation id.
    parameters: [], // expected params.
    // expected responses
    responses: {
      // response code
      200: {
        description: "Todos were obtained", // response desc.
        content: {
          // content-type
          "application/json": {
            schema: {
              $ref: "#/components/schemas/Todo", // Todo model
            },
          },
        },
      },
    },
  },
};
```

https://www.section.io/engineering-education/documenting-node-js-rest-api-using-swagger/


# Containers & Docker

# Containers



## *Who is Malcom McLean ?*


https://en.wikipedia.org/wiki/Malcom_McLean

# Docker Terminology



## Let's start with some basic concepts

**Docker Image**

The basis of a Docker container

**Docker Container**

The standard unit in which the application service resides

**Docker Engine**

Creates, ships and runs Docker containers deployable on physical or virtual host locally, in a datacenter or cloud service provider

**Docker Trusted Registry**

Dedicated image store and distribution service deployed in your firewall

# Docker



We can build our software *and the environment it runs on* as code and deploy easily.

```
docker run -it ubuntu
```

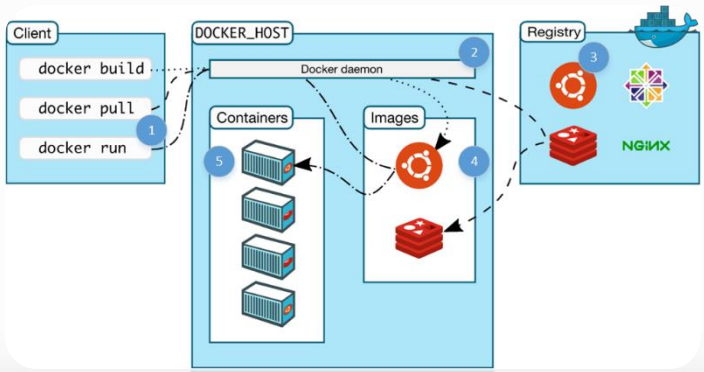
After a bit of spinning, you'll see a prompt like this:

```
root@719059da250d:/#
```

Try out a few commands and then exit the container:

```
root@719059da250d:/# lsb_release -a
No LSB modules are available.
Distributor ID:    Ubuntu
Description:    Ubuntu 14.04.4 LTS
Release:    14.04
Codename:    trusty
root@719059da250d:/# exit
```

1. We issue a docker command:
   - `docker`: run the docker client
   - `run`: the command to run a new container
   - `-it`: option to give the container an interactive terminal
   - `ubuntu`: the image to base the container on
2. The docker service running on the host (our machine) checks to see if we have a copy of the requested image locally- which there isn't.
3. The docker service checks the public registry (the docker hub) to see if there's an image named `ubuntu` available- which there is.
4. The docker service downloads the image and stores it in its local cache of images (ready for next time).
5. The docker service creates a new container, based on the `ubuntu` image.



# Docker Images

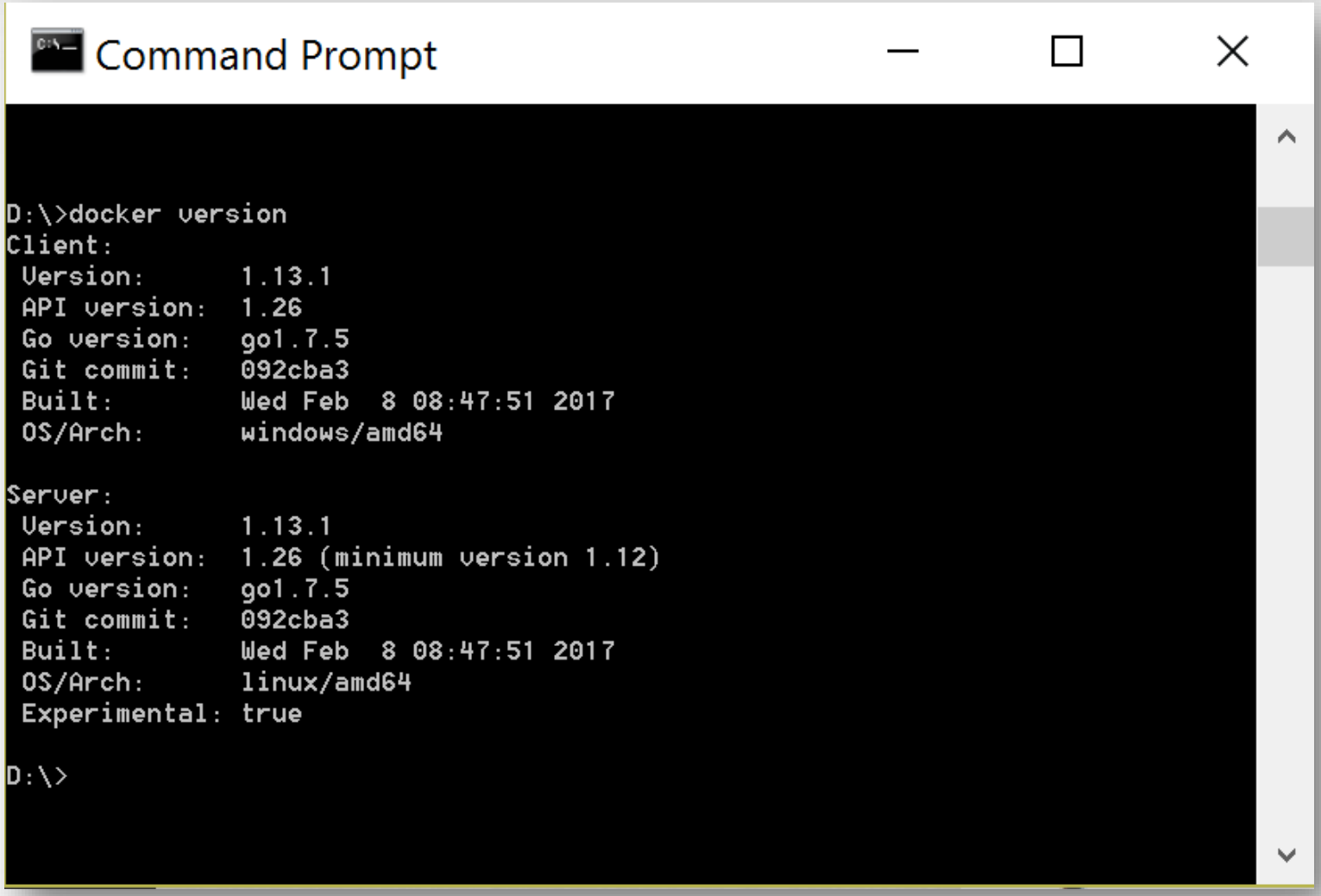


| | | | |
|---|---|---|---|
| nginx official | 3.5K STARS | 10M+ PULLS | DETAILS |
| busybox official | 737 STARS | 10M+ PULLS | DETAILS |
| ubuntu official | 4.3K STARS | 10M+ PULLS | DETAILS |
| redis official | 2.4K STARS | 10M+ PULLS | DETAILS |
| registry official | 950 STARS | 10M+ PULLS | DETAILS |
| swarm official | 410 STARS | 10M+ PULLS | DETAILS |
| mongo official | 2.1K STARS | 10M+ PULLS | DETAILS |

https://hub.docker.com/explore/

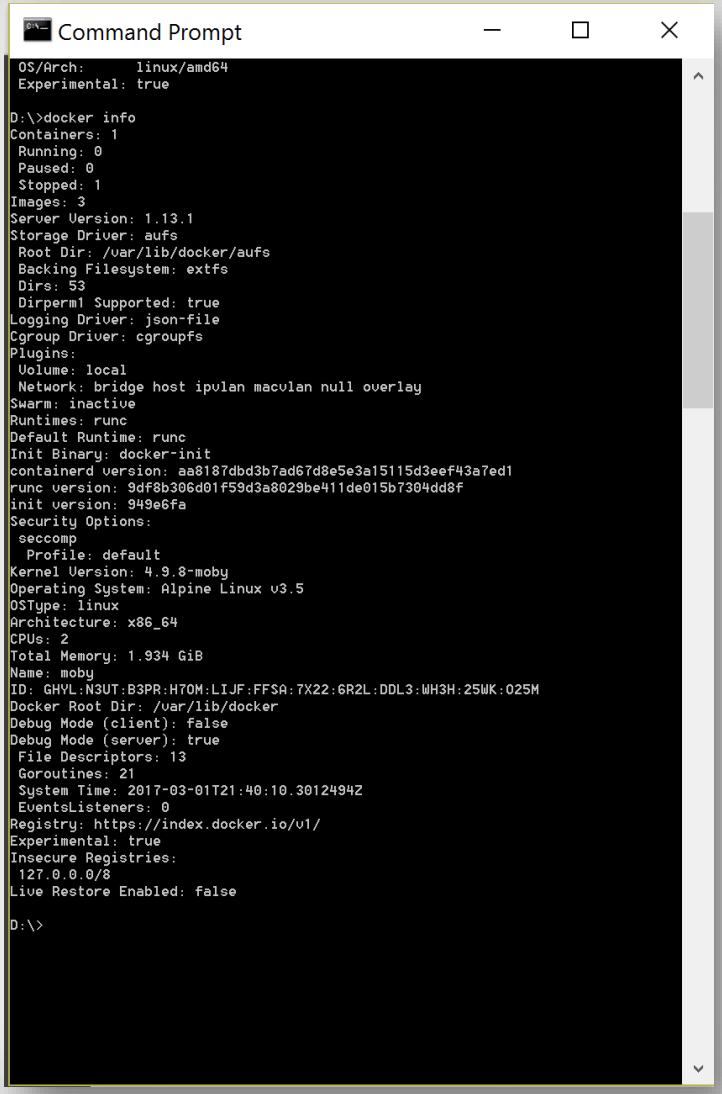
# Docker Version





# Docker Info

המרכז האוניברסיטאי אריאל בשומרון

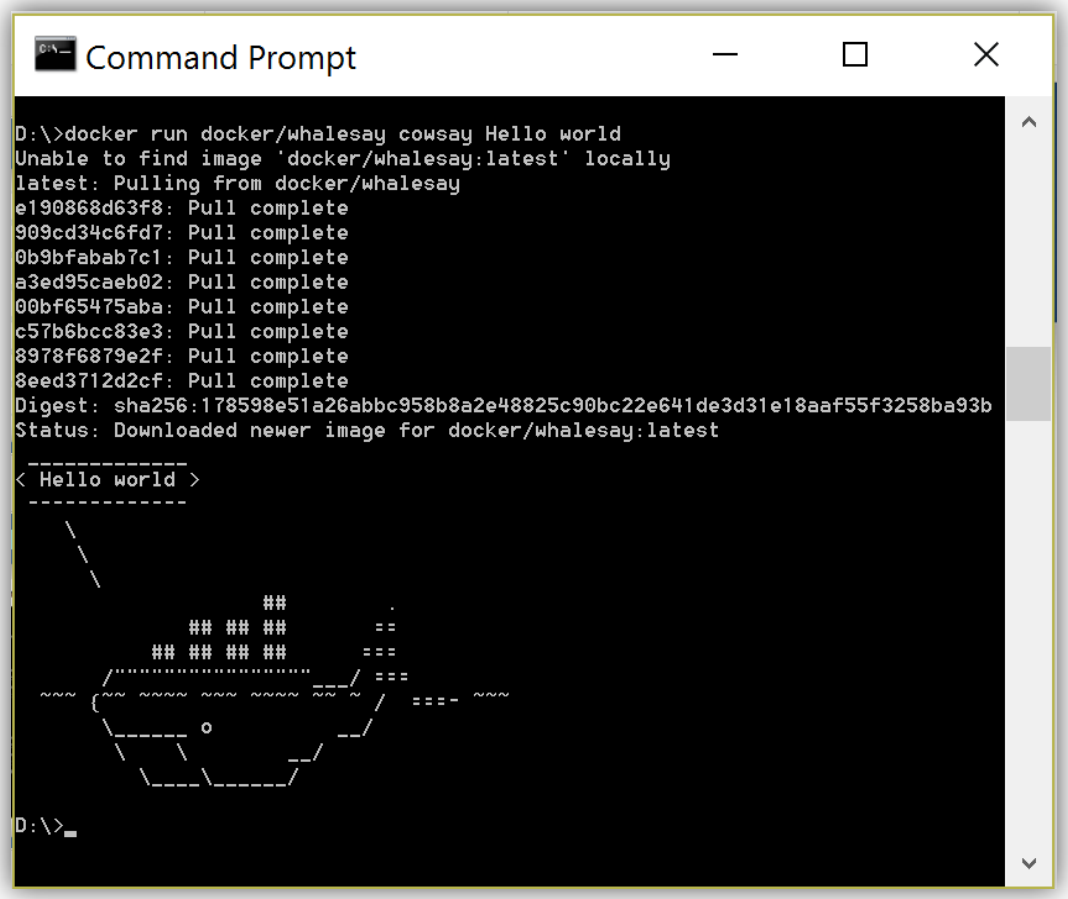



machine
system
a network

# Docker Hello World



```
docker run docker/whalesay cowsay Hello world
```



המרכז
האוניברסיטאי
אריאל בשומרון

# Serverless Development

# What is Serverless

המרכז האוניברסיטאי אריאל בשומרון

- Serverless is a cloud development model that allows developers to build and run applications without having to manage servers
- Servers are abstracted away from application developers
- Cloud provider deals with non-functional aspects like scaling or provisioning
- Developers can simply package their code in Containers for deployment.
- Serverless apps respond to demand and Automaticlly scale up and down as needed.
- Usually using event driven execution model
- Two Types:
  - Backend-as-a-Service (BaaS)
  - Function-as-a-Service (FaaS)

# Serverless



Simulates a physical machine
Provides a local file system
Can be accessed over a network


https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started-hello-world.html

https://www.serverless.com/blog/serverless-express-rest-api/