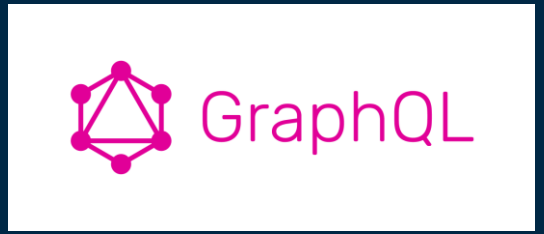
Application frameworks



Introduction
Kushira Godellawatta

Overview

- What is this course?
- Our objectives?
- What are we going to cover?
- How are we going to evaluate you?

What is this course?

- This course will focus on application development using industry standards and industry leading frameworks.
- Mainly this course will build around Java and JavaScript languages.
- Course will also focus on industry practices and principles in software engineering.
- Popular JavaScript and Java frameworks as well as an introduction to popular NoSQL database will be delivered as well.

Objectives

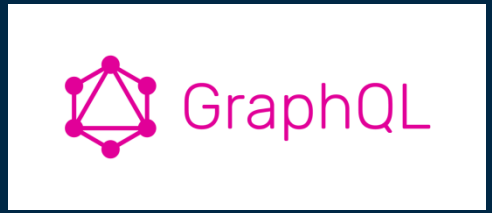
- Deliver industry practices and principles in software engineering and encourage student to use them in their development.
- Let student to discover new trends two industry leading software development languages.
- Deliver sufficient knowledge on JavaScript and Java full stack development.
- Deliver an introduction NoSQL databases, MongoDB and how to use MongoDB in full stack development.
- Introduction to REST style web services.
- Introduce student to industry leading frameworks in web application and web service development along with leading architecture and authentication mechanisms being followed in industry.

A student should be able to develop a full stack web application using JavaScripts and MongoDB while applying engineering principles and practices and should be able to replace the backend service code with a Java web service.

Objective

What are we going to cover?

- JavaScript
- React JS
- NodeJS
- KoaJS
- Graph QL
- Java
- Spring Boot
- MongoDB
- Docker



Evaluation

- Evaluation is based on applying the concepts and learnings in practical applications.
- Technical blog.
- 1 lab examinations.
- Midterm examination.
- Hackathon
- In class marks
- Group project.
- Final examination.


KEEP
CALM
AND
NEVER
GIVE UP

Principles

- S.O.L.I.D
- Guidelines - Approaching the solution
- Guidelines - Implementing the solution.
- Practices
  - Unit testing
  - Code Quality
  - Code review
  - Version controlling
  - Continuous integration

S.O.L.I.D

S.O.L.I.D are 5 object oriented principles that should be followed when designing software.

- Single responsibility
- Open-close
- Liskov substitution
- Interface segregation
- Dependency inversion

Single responsibility principle

“A class should have one and only one reason to change, meaning that a class should have only one job”

This is not only about a class it what we consider as a unit ex: function, module, API etc.

- Think about a class that calculate area or a circle and output the area as a HTML.
- What if in case a JSON output is required?

Open/close principle

“Objects or entities should be open for extension, but closed for modification”

- Think about a class that calculate area or a circle and a square and output the area as a console value.
- What if in case this class need to calculate area of a triangle?
- Use abstraction to keep classes open for extension.
- When there is high possibility to change always depend on abstractions than concrete implementations.

Liskov substitution principle

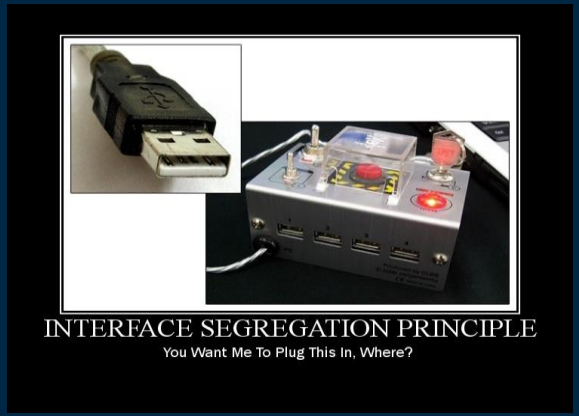
“Every subclass/derived class should be able to substitute their parent/base class”

- When extending a class it should perform basic functionalities of the base class.
- Child class should not have unimplemented methods.
- Child class should not give different meaning to the methods exist in the base class after overriding them.

Interface segregation principle

“Clients should not be forced to implement methods they do not use”

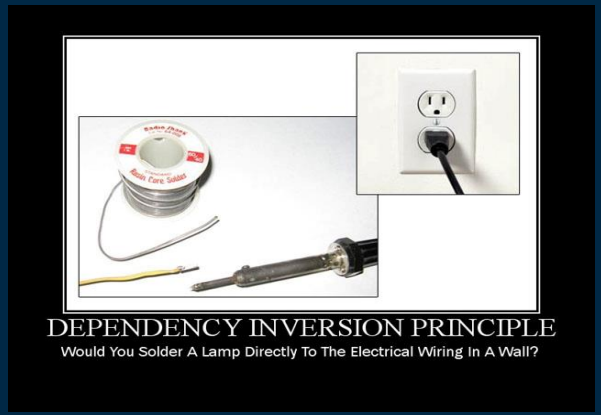
- Think of Shape interface with draw() and calculateArea() methods and a client who wants only the shape to be drawn.
- These are called fat interfaces.
- Group methods into different interfaces each serving different set of clients.



Dependency inversion principle

“Higher level modules should not depend on lower level modules, but they should depend on abstractions”

- Think of the classic 3 tier architecture. Business logic layer depends on Data access layer.
- What if we need to change the data access layer (Different database?).



Approaching the solution

In software engineering we try to find a technical solution for a business problem.

How we can achieve this solution in the best way possible?

- Think throughout the problem
- Divide and conquer
- KISS
- Learn, especially from mistakes
- Always remember why software exists
- Remember that you are not the user

Think throughout the problem

- Before approaching the solution or even before starting to think about the problem think through the problem.
- Make sure you understand the problem that you are going to resolve, completely.
- Make sure to clear any unclear part before designing the solution.
- Don’t be afraid to question there are no stupid questions.

Divide and conquer

- Now divide the problem into smaller problems.
- Make it manageable and easily understandable.
- Try to find the perfect balance between priority and clarity (less complex, easily understandable).

KISS

- Keep it simple and stupid.
- Do not deliberately make your solution complex.
- Do not overthink or over engineer.

Learn from the mistakes

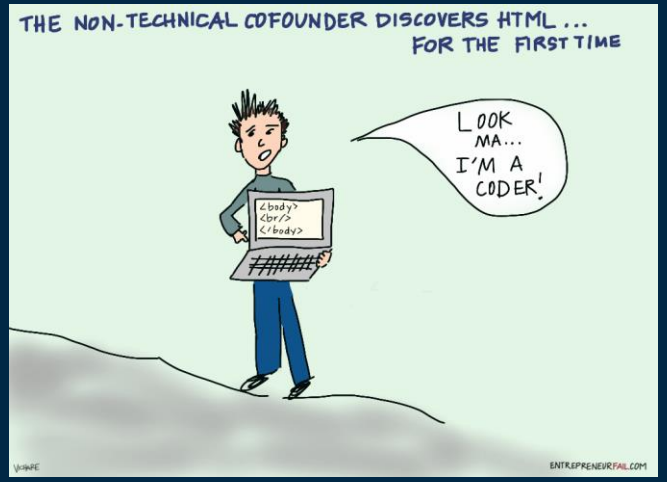
- Embrace the change.
- Always anticipate the changes as much as possible.
- Do not over engineer it, but keep the provisions to extend

Reason software exists

- Keep in mind the bigger picture why this software exists.
- Loss of the bigger picture might cause following a wrong path.

You won’t be using the software

- Enduser is not technically capable as same as you are.
- Do not assume that user will understand.
- User friendliness and user experience matters.



Implementing the solution

When we are implementing the designed solution there are some guidelines to keep in mind.

- YAGNI
- DRY
- Embrace abstraction
- DRITW
- Write code that does one thing well
- Debugging is harder than writing code
- Kaizen

YAGNI - You ain’t gonna need it

- Do write code that is no use in present but you are guessing will come in handy in the future.
- Future always change. This is a waste of time.
- Write the code you need for the moment only that.

DRY - Don’t repeat yourself

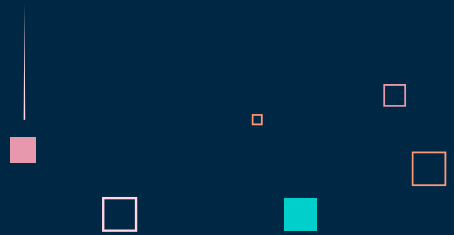
- Always reuse code you wrote.
- Code as best as possible and keep it generalize and reusable.

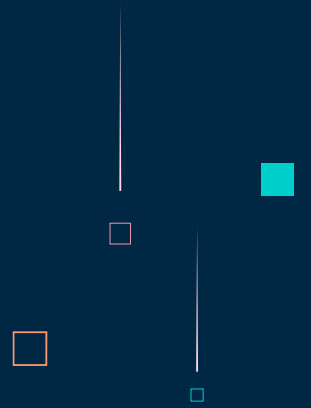
Embrace abstraction

- Make sure your system functions properly without knowing the implementation details of every component part.
- User class should be able to authenticate user without knowing where to get username and password.

DRITW - Don’t reinvent the wheel

- Someone else might have already solved the same problem.Make use of that.
- So, do you need to write a code that communicate with the database?





Write code that does one thing well

- SIngle piece of code that do one thing and that one thing really well.
- Do not try to write the magic code that does it all….

Debugging is harder than writing the code

- Make it readable as possible.
- Readable code is better than compact code.

Kaizen - Leave it better than when you found it

- Fix not just the bug but the code around it.
- Band-aid bugfix won’t help if the real problem is a design problem.

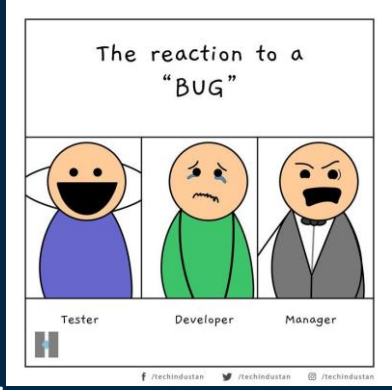
Practices

- Unit testing
- Code Quality
- Code review
- Version controlling
- Continuous integration

Practices

- Practices are norms or set of guidelines that we should follow when we are developing code.
- Introduced by coding guru after studying years of years experience.
- These practices are being considered industry wide as best practices for software engineering.
- Unit testing
- Code quality
- Code review
- Version controlling
- Continuous integration

Unit testing



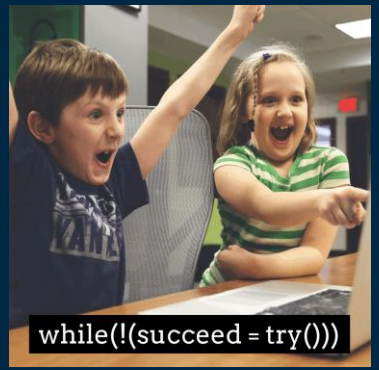
- Small code that verifies parts of your main code.
- Unit could be a class, function, module, API etc.
- Unit test will verify the class, function.. Is working as expected and delivers the expected output.
- Allows developer to freely change or improve the code, make sure it didn’t break anything by running the unit test.
- Unit testing will eventually make code testable which basically results an extensible code base.
- Verification mechanism for developers.
- Early identification of integration issues.

Code quality

- Code to be maintainable code quality is vital.
- Code should readable and easily understandable.
- Code should adhere to engineering best practices as well as language and domain best practices.
- Frequently analyze quality of the code using tools.
- Identify potential erroneous scenarios.
- Improve the performance of the code.
- Code complexity, large methods and classes, meaningless identifiers, code duplication, large number of method parameters.

Code review

- Best way to improve code quality.
- Objective is to improve the code not to criticize the developer.
- Improve the performance, find out the best way of resolving the problem; 4+ eyes on the code.
- Review less than 400 LOC and rate should be 500 LOC per hour, do not review continuously more than hour.
- Peer reviews, lead reviews and pair programming are some methods of doing code reviews.

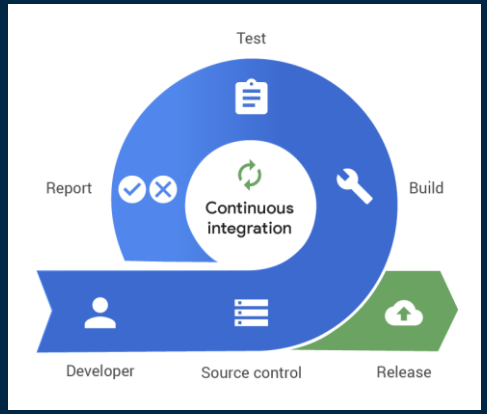


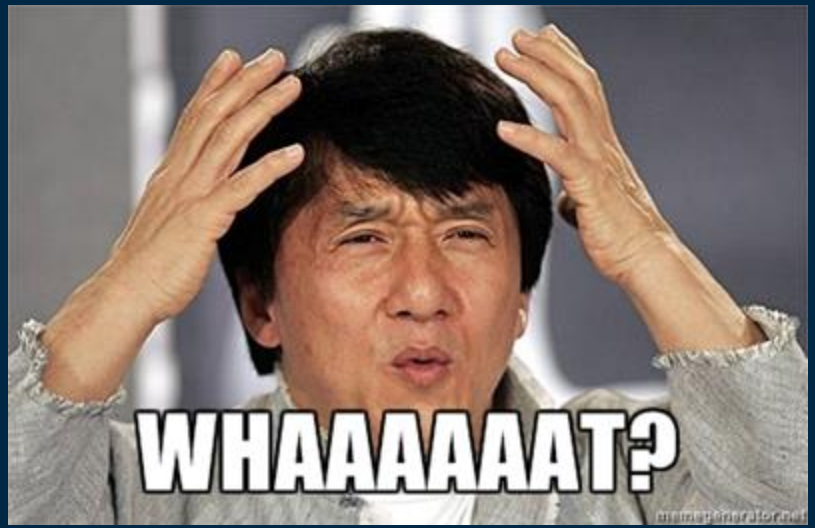
Version controlling

- Code should always be version controlled.
- Allow developers to change and improve the code freely without being afraid of breaking the code.
- Let multiple developers to collaborate on the same code base.
- Remove the single point of failure in code base.
- Use multiple branches tags for maintaining the code base.

Continuous Integration

- Continuous integration is a development practice.
- Developers need to check-in the code to a shared repository several times a day.
- Each checking is verified by an automated build.
- This allows developers to detect issues early and fix them without a delay.




WHAAAAAAT?

Any Questions?

Application frameworks

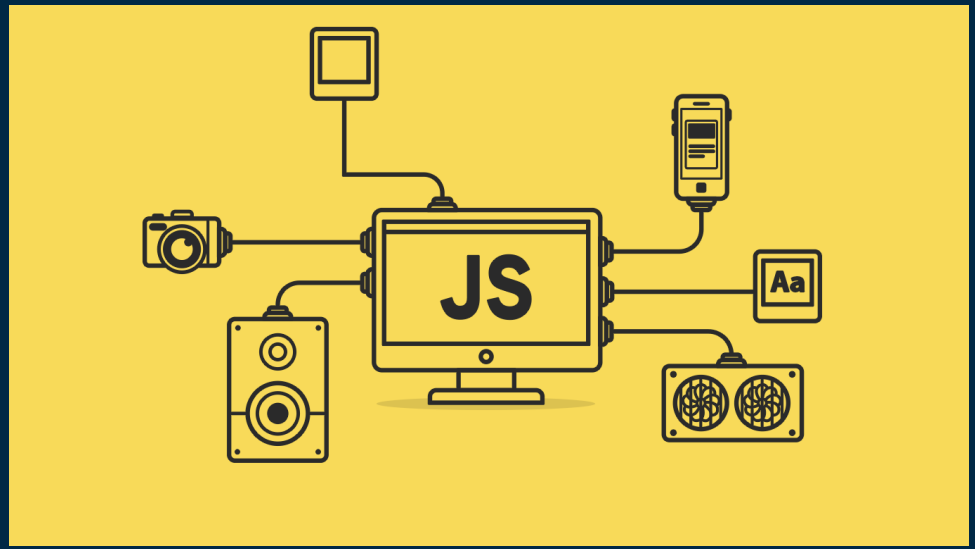
JavaScript, Version controlling and NoSQL

Overview

- JavaScript
- Version controlling
- NoSQL

JavaScript

- Introduction
- Classes, objects and prototype
- How ‘this’ acts
- Strict notation
- Function closure
- Callbacks and promises



JavaScript

- By default, JavaScript programs run using a single thread. Though there are ways to create new threads JavaScript is considered as Single threaded language.
- JavaScript does not wait for I/O operations to get completed, instead it continues the execution of the program. This is called as non-blocking I/O.
- JavaScript is asynchronous because of the NIO nature.
- JavaScript is dynamically typed. It determines variable types, ordering etc. in runtime.
- JavaScript support OOP as well as functional programming (Multi-paradigm).
- JavaScript has an eventing system which manages it’s asynchronous operations.

Classes and objects

- In JavaScript, a constructor function is used with the ‘new’ key keyword when creating a Object.
- Constructor function is a just another function.
- When function is used with ‘new’ keyword that function acts as a Class.
- Recently JavaScript introduced ‘class’ keyword, but it is not yet adopted by all JavaScript engines.
- Another way of creating a object is using object literals (‘{}’). These objects are considered to be singleton.
- JavaScript supports static methods and variables.
- When ‘new’ keyword is used, new object is created and it assigned as ‘this’ for the duration of call to the constructor function.

Prototypes

- JavaScript functions has a reference to another object called prototype. It is somewhat similar to class definition in other languages.
- It is really another object instance.
- In JavaScript prototype object is used when creating objects, for inheritance and adding methods to a JavaScript class.
- Because of the flexibility in JavaScript there are multiple ways to create classes as well as to extend classes. Prototypes are the recommended way of doing so.
- Function that is being used to create objects is called a constructor function.
- Object instance also has a prototype it is basically the object instance from which object is being created. Object '__proto__' is where object get its properties inherited from.
- Functions prototype is used to inherit properties to object instances.

‘this’ in JavaScript

- Unlikely other languages in JavaScript ‘this’ keyword acts differently.
- Inside a object ‘this’ refers to object itself.
- In global context ‘this’ refers to global object (in browser it is the window object). This behaviour will get changed in strict mode.
- If a function which is using ‘this’ keyword is being passed to another object then ‘this’ will refers to that object, but not to the original object where function was declared at first place.
- This behaviour is very noticeable in callback and closures.

Strict notation

- Restricted mode of JavaScript.
- Purpose it make it easier to write secure JavaScript.
- Strict mode make bad practices in JavaScript to errors.
- Keep developer away from using syntaxes that will get invalidate with future JavaScript developments.
- For example it does not allows to create variables without the var keyword (Variable have to be declared).
- Another example would be it will stop referring to the window object as ‘this’ from outside object instances.

Closure

- JavaScript closure is a function which returns another function.
- In JavaScript closure is used to encapsulate variables into a function and restrict access to it from the outside.
- JavaScript creates an environment with all the local variables from the outer function when the inner function is created. Closure is the combination of this environment and the inner function.

Callback and promises

- JavaScript is asynchronous. All I/O operations in JavaScript is implemented to be asynchronous by nature.
- Reason for this is JavaScript being a single threaded language if an I/O operations holds the thread till it get completed JavaScript won’t perform well as a programming language.
- But asynchronous operation introduce difficulty when we need to do synchronous processing using data.
- This is solved by using callbacks and promises.
- Callback is a function that is being passed to an async task and on completion the function will be executed.
- Promise is an object that is being returned from async tasks. Promise have properties to deal with async operations synchronously.

Callback and promises…

- Nested callbacks passed into sequence of async tasks is referred to a ‘callback hell’.
- Promise object was introduced to solve this problem.
- Promise object has a set of properties, methods and mechanism of chaining to handle complex async tasks nicely.

Version controlling

- What and why?
- Terminology
- Best practices
- GIT

What?

- Managing changes to a source.
- Changes are identified using a revision number.
- Each revision has its timestamp as well as the person who done the change.
- Revisions can be restored, compared and merged.
- “Management of multiple revisions of the same unit of information”

Why?

- Easier backups and centralized source code repository.
- Easy collaborative development.
- Overview of changes performed to a file.
- Access control.
- Conflict resolvement.

Terminology

- Repository
  - Central location where all the files are being kept. Usually a directory with set of files.
- Trunk
  - Also referred to as master branch. This is where the most stable code is being placed which is referred as the production code.
- Stage
  - Mark files for tracking changes.
- Commit
  - Create a snapshot of the changes being made to the files.

Terminology…

- Branch
  - Copy of the master branch taken at a given point. All the feature developments and bug fixes will be done in a branch. Usually it is allowed to have multiple branches at the same time.
- Checkout
  - Mark/unlock file for changing.
- Merge
  - Combining branches together to update the master branch.
- Merge conflict
  - Merge conflicts occur when merging a file which has been changed in two separate branches or places. Changes that interfere other changes.

Best practices

- Use a source control system.
- Always make sure to have the latest version of the file.
  - In distributed source control system advice is to get the latest source code at least start of the day.
- Checkout only what you need.
- Merge code with the development branch at least once per day.
- Always make sure code is working as expected and it is not causing any other code to break.
- Follow a formal review process when merging.

Git

- Most popular version control system.
- Distributed version control system.
  - Client get a complete clone of the source code. In a disaster situation full source along with all history can be restored from a client.
- Free and open source.
- Multiple branches and tags.
  - Feature branches, role branches (production).
- Faster comparing to other systems (works on a linux kernel and written in C).
- Support multiple protocols
  - HTTP, SSH
- Staging area, local commits and stashing.
  - Staging area - Mark files to be committed.
  - Local commit - Commit code locally without pushing into the remote branch.
  - Stashing - Keep file changes in Stash and apply them in a later.

Git commands

- Git init
- Git clone
- Git add
- Git stage
- Git commit
- Git push
- https://confluence.atlassian.com/bitbucketserver/basic-git-commands-776639767.html

Git: Interactive Learning

Following are two good interactive demos for learning git.

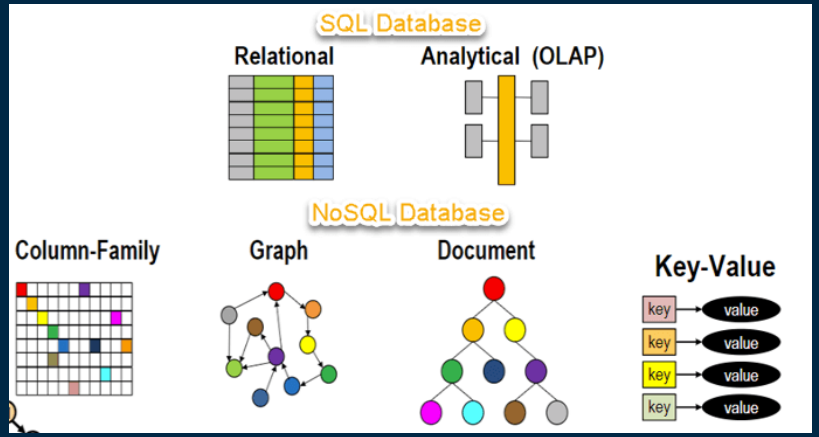
The fundamentals are found in [1] and advanced branching demo is in [2].

[1] https://try.github.io

[2] http://pcottle.github.io/learnGitBranching/

NoSQL

- What?
- Why not SQL?
- Strengths and weaknesses
- MongoDB



What?

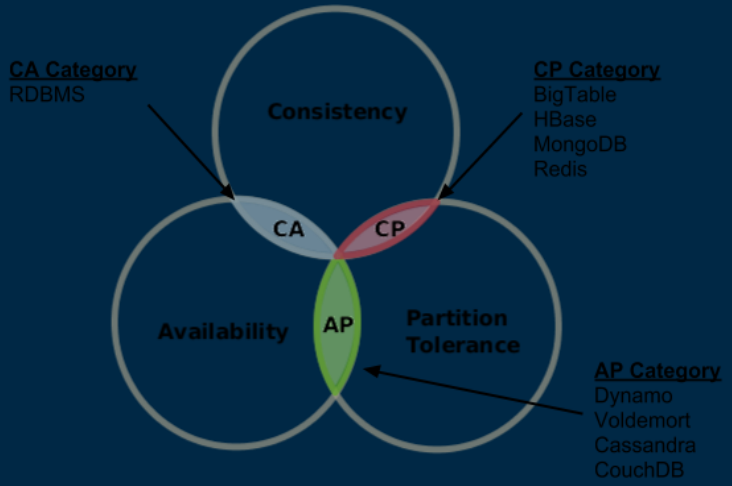
- Non relational (mostly), Schema free.
- Distributed.
- Horizontally scalable.
- Easy replication.
- Eventually consistent.
- Open source (mostly).
- No transaction support.

Why?

- Remove the burden of data structures mismatch between application in-memory and relational databases.
- Integrate databases using services. (ElasticSearch).
- Relational databases not designed to run efficiently on clusters.
- Aggregate oriented databases, are easier to manage inside clusters and based on the domain driven design. (Order details inside the order).
  - But inter aggregate relationships are harder to manage.

CAP theorem

- Consistency - Every read receives the most recent write.
- Availability - Every request receives a response (no guarantee that it is the most recent write).
- Partition tolerance - System will continue to operate despite number of messages lost among network nodes.



Types

- Key-Value - Stores data as key value pairs.
  - Ex: Redis, Riak, Memcached.
- Document - Stores data as documents (JSON, BSON, XML) in maps or collections.
  - Ex: MongoDB
- Column Family - Store data in column families as rows that have many columns associated with.
  - Ex: Cassandra
- Graph - Store entities(nodes) and relationships(edges) between them and represent it in a graph.
  - Ex: Neo4j

MongoDB

- NoSQL document database.
- Strong query capabilities with aggregations using JavaScript.
- Use SpiderMonkey JavaScript engine.
- High availability with replica sets.
- Reads and writes on primary by default.
- Eventually consistent on secondary instances.
- In built file storage called Grid File System.

MongoDB queries

- Insert
- Find
- Update
- Remove

Any Questions?

Application frameworks

NodeJS

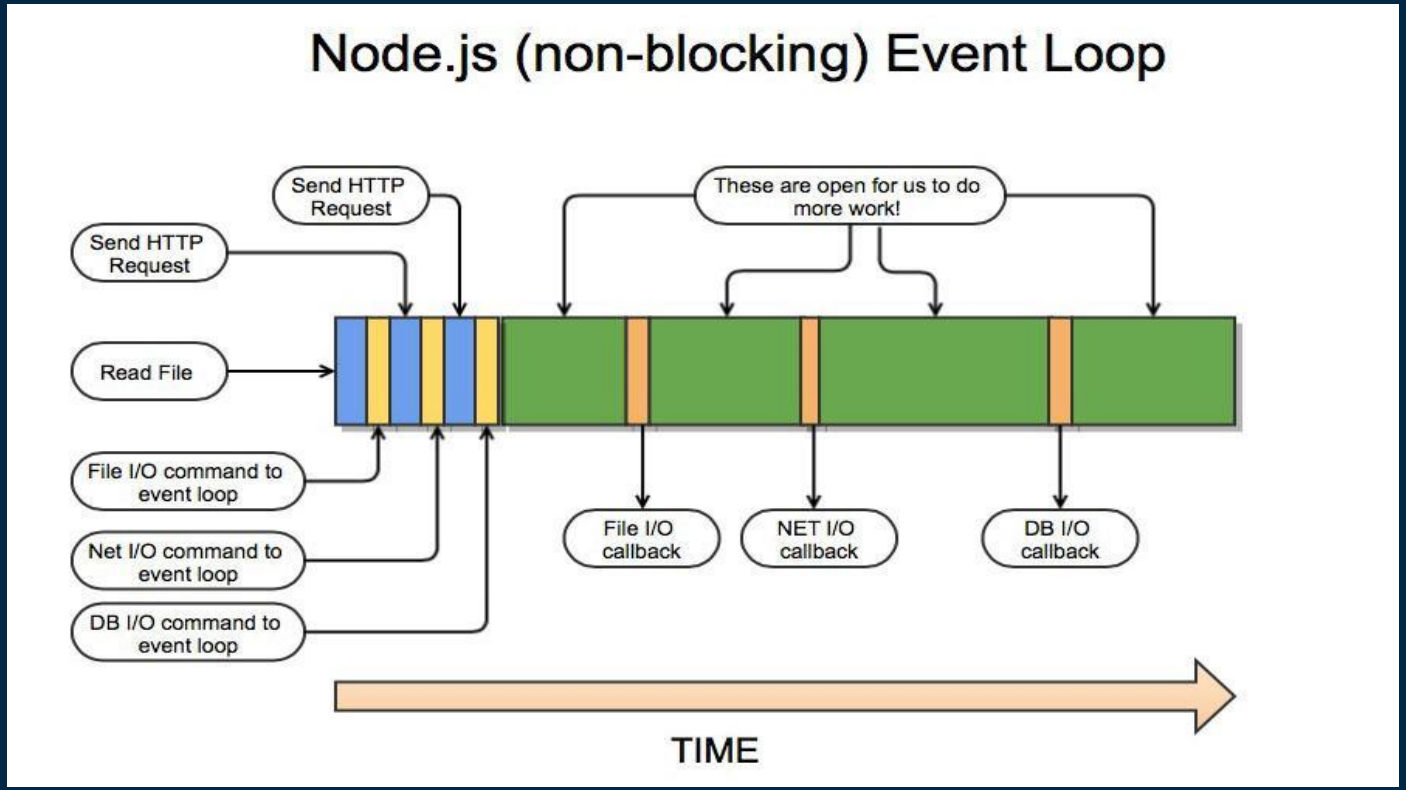
Overview

- NodeJS
- Event loop
- Use cases
- Advantages and disadvantages
- Package manager
- Require
- Let’s do coding

NodeJS

- Developed by Ryan Dahl.
- Created with the aim of creating real-time websites with push capabilities (websockets).
- NodeJS is an open source, cross platform runtime environment for server-side and networking applications.
- Build on V8 engine, Chrome's JavaScript engine.
- Uses event-driven, non-blocking I/O model which makes NodeJS lightweight and efficient.
- Ideal for data-intensive real-time applications that run across distributed devices.
- NodeJS comes with several JavaScript libraries that help basic programming.
- NodeJS eco-system 'npm' is the largest in the world for open source libraries.

Event loop


Node.js (non-blocking) Event Loop
Send HTTP Request
Send HTTP Request
These are open for us to do more work!
Read File
File I/O command to event loop
Net I/O command to event loop
DB I/O command to event loop
File I/O callback
NET I/O callback
DB I/O callback
TIME

Use cases

- Not the best platform for CPU intensive heavy computational applications.
- Ideal for building fast and scalable network applications.
- NodeJS is capable of handling a huge number of simultaneous connections with high throughput.
- For each connection NodeJS does not spawn new Thread causing max out of memory instead handle all in single thread using non-blocking I/O model.
- NodeJS has achieved over 1 Million concurrent connections.
- Bubbling errors up to NodeJS core event loop will cause crashing the entire program.

Use cases

- I/O bound applications.
- Data streaming applications.
- Data intensive real-time applications.
- JSON APIs based applications.
- Single page applications.

Advantages

- Ability to use single programming language from one end of the application to the other end.
- NodeJS applications are easy to scale both horizontally and vertically.
- Delivers improved performance since V8 engine compile the JS code into machine code directly.
- Performance increased via caching modules into memory after the first use.
- Easily extensible.
- Support for common tools like unit testing.
- Well build ‘npm’ package manager and it’s large number of reusable modules.

Disadvantages

- Even though there are number of libraries available, the actual number of robust libraries is comparatively low.
- Not suitable for computationally intensive tasks.
- Asynchronous programming model is complex than synchronous model.

Node package manager

- Reusable NodeJS components easily available through online repository.
- Build in version and dependency management.
- Build in scripting mechanism.
- Global installations will be available throughout the system while local installations will only be available for that particular application.
- By default all the dependencies will get installed to ‘node_modules’ directory.
- ‘package.json’ contains all information related to the NodeJS application. The file be placed in the root directory of the application.
- ‘package.json’ will contain name, version, author, repository, required node version, scripts and dependencies etc.

Node package manager…

- To denote the compatible version numbers npm has mechanism for defining them;
  - Less than or equal ‘<=’, greater than or equal ‘>=’.
  - Approximately equivalent to ‘~’.
  - Compatible with ‘^’.
  - Any ‘*’.
  - Any ‘1.2.x’.
  - Latest ‘latest’
- There are two types of dependencies ‘devDependencies’ (development time dependencies) and ‘dependencies’ (application runtime dependencies).

Node require

- NodeJS follows commonJS pattern when loading modules.
- Require modules get loaded synchronously and will be cached after the first use.
- If the file does not start with ./, ../, or / module is not a core module NodeJS will look the dependency on the node_modules directory.

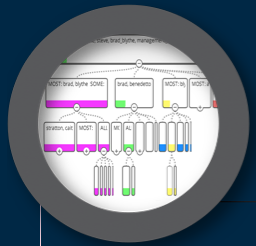
Let’s do some coding

- Hello world.
- Read file.
- Write file.
- Stream the content of a file.
- Create event and subscribe to it.
- Create a child process and message passing between two processes.
- Create a web server and listen to a port via http protocol.
- Add a separate module and use it.
- Use a promise.

Any Questions?

Application frameworks

REST services & KoaJS



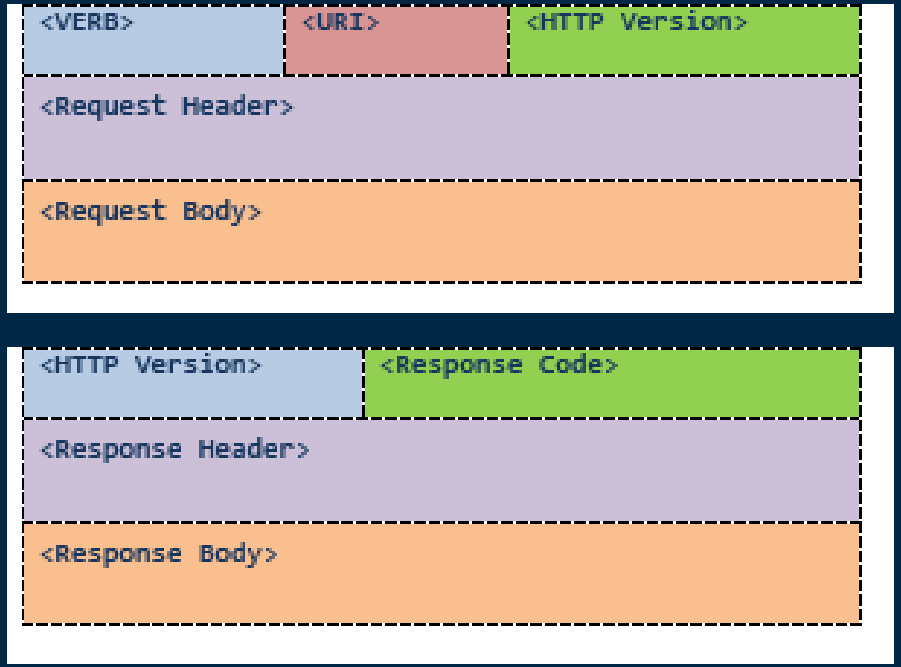
verview

- REST
- HTTP messages
- HTTP verbs
- Six Constraints
- Compliance
- Best practices
- HTTP codes
- Koa JS
- Let’s do coding

RESTful services

- REpresentational State Transfer protocol.
- Built around things or resources - Resource based.
  - Nouns vs actions. REST services are not action based though HTTP verbs being used to perform different operations on the same resource.
- Resource are identified by a URI.
- Multiple URIs may refer the same resource.
- Client and server communication.
- JSON or XML is used to pass data.
- Lightweight, scalable and maintainable.

HTTP messages



HTTP Verbs

Verb	Operation	Safe
GET	Fetch a resource	Yes
POST	Insert new resource	-
PUT	Replace existing resource	Idempotent
PATCH	Update Resource	Idempotent
DELETE	Remove a resource	No
OPTIONS	Get all allowed options	Yes
HEAD	Get only the response header	Yes

6 Constraints

- Uniform interface
- Stateless
- Cacheable
- Client-Server
- Layered System

Uniform interface

- Based on the HTTP specification.
- URIs refers to resources and HTTP verbs are actions performed on resources.
- HTTP verbs (GET, PUT, POST, DELETE).
- URIs (resource).
- HTTP request (header, body, query parameters and URI) and response (header/status and body).
- Use hypermedia to better navigation through resources (HATEOAS - Hypermedia As The Engine Of Application State). Use links for retrieval of related resources.

Stateless

- No client state on the server.
- Self containing messages. Each message contains sufficient information for that particular operation.
- Session state will be kept in client side.

Client-Server

- Use URI to make the connection between two.
- Clearly separates user interface and services. Client is portable, client does not have any connection to data server. Server stand independent from different user interfaces.
- HTTP stack is the communication platform.

Cacheable

- REST services should be cashable.
- Resources returned from the server should be cashable.
- Client can cache resources (implicit caching).
- Server determines what and how should be cached (explicit caching).
- Server-Client negotiate the caching scheme.

Layered System

- Client does not sees the underlying layers or complexities of the service.
- Client only knows the URI.
- Server is decoupled enough to have multiple layers and intermediate services sitting in between client and server.
- Client only deals with the abstraction of resource URI and verb.
- Highly scalable.

Code on demand

- Optional constraint.
- Server has the luxury to transfer some part of the logic to the client.
- JS is being passed to client to execute.

Compliance with REST constraints

- Performance
- Scalability
- Simplicity - Uniform interface
- Modifiability - Change components while running
- Visibility - Communication between agents
- Portability - Components, by moving program code with data
- Reliability

Best practices

- Nouns over verbs.
- Do not use GET method to alter the state.
- Use plural nouns.
- Represent subresources.
- Use HTTP status codes for errors.
- Use links to other related resource where needed.
- Provide additional functionalities like filtering, pagination, sorting and field selection via query parameters.
- Version the api.
- Always expose via HTTPS.

HTTP codes

- 200 - OK *
- 201 - Created *
- 202 - Accepted
- 204 - No content *
- 301 - Moved permanently
- 302 - Found
- 304 - Not modified *
- 400 - Bad request *
- 401 - Unauthorized *
- 403 - Forbidden *
- 404 - Not found *

HTTP codes

- 405 - Method not allowed
- 409 - Conflict *
- 412 - Precondition failed
- 500 - Internal server error *
- 502 - Bad gateway
- 503 - Service unavailable

Koa JS

- Node JS web application framework.
- Designed for developing web applications and APIs.
- Minimal features but many features available via plugins.
- Robust routing.
- High performance.
- Test coverage.
- Extensible.
- Reusable.

Let’s do some coding

- Serve html file using Koa JS.
- Create REST API using Koa JS.
- Persist resources in Mongodb.

References

- http://www.restapitutorial.com/lessons/whatisrest.html#
- https://restfulapi.net/
- https://martinfowler.com/articles/richardsonMaturityModel.html
- https://en.wikipedia.org/wiki/Representational_state_transfer
- https://developer.paypal.com/docs/api/
- https://koajs.com/
- https://github.com/koajs/bodyparser
- https://github.com/koajs/router

Any Questions?