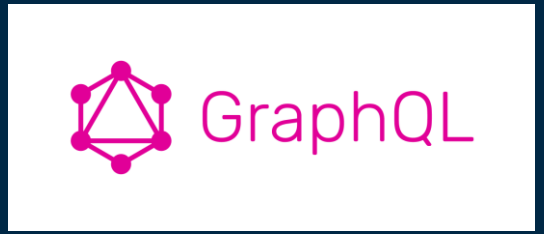
Application frameworks



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

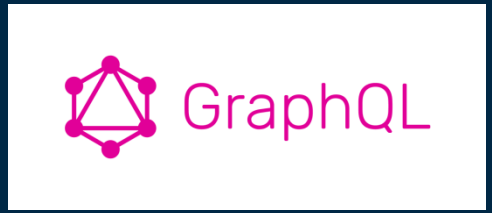
- Deliver industry practices and principles in software engineering and encourage student to use them in their development.
- Let student to discover new trends two industry leading software development languages.
- Deliver sufficient knowledge on JavaScript and Java full stack development.
- Deliver 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.

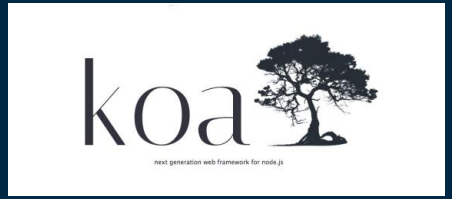
A student should be able to develop a full stack web application using JavaScripts and MongoDB while applying engineering principles and practices and should be able to replace the backend service code with a Java web service.

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

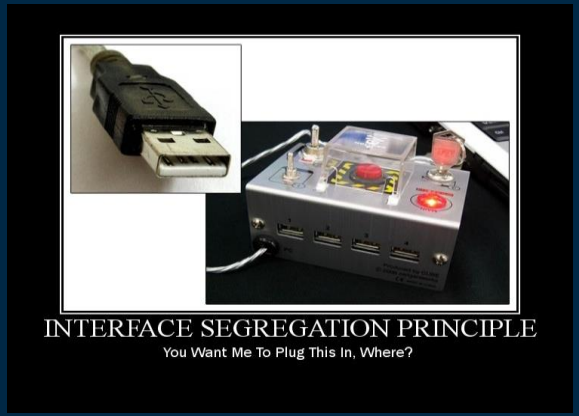
“Every subclass/derived class should be able to substitute their parent/base class”

- 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”

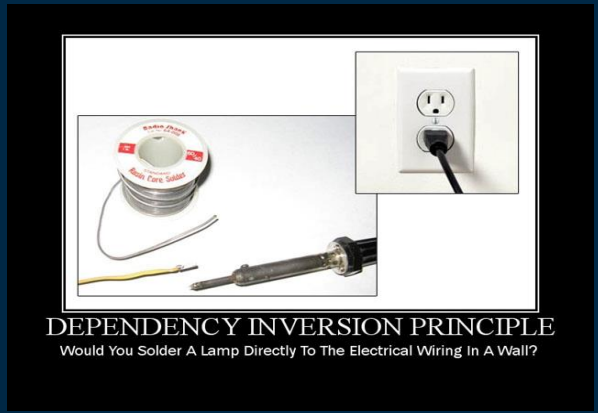
- Think of Shape interface with draw() and calculateArea() methods and a client who wants only the shape to be drawn.
- These are called fat interfaces.
- Group methods into different interfaces each serving different set of clients.



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

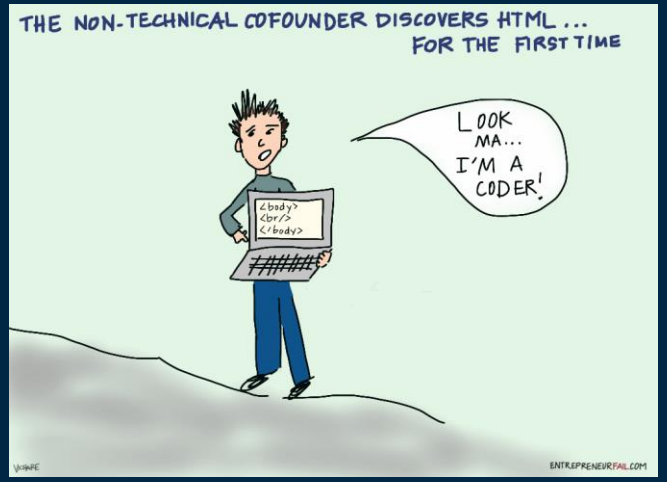
- Embrace the change.
- Always anticipate the changes as much as possible.
- Do not over engineer it, but keep the provisions to extend

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

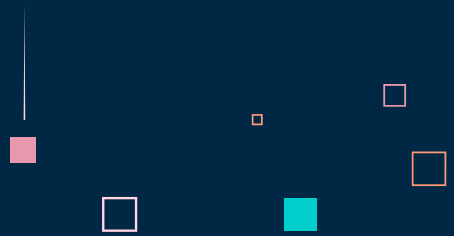
- Always reuse code you wrote.
- Code as best as possible and keep it generalize and reusable.

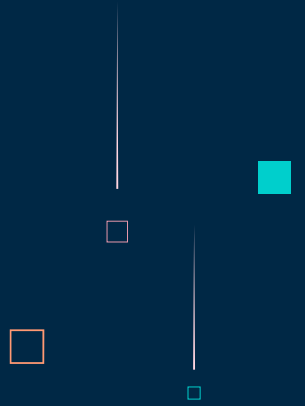
Embrace abstraction

- 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.

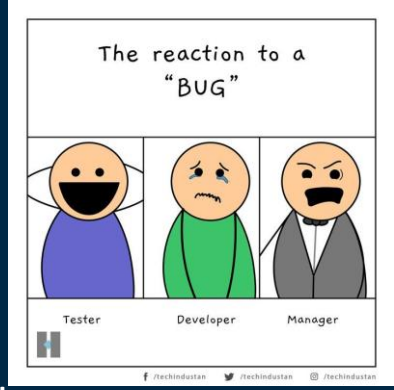
Practices

- 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



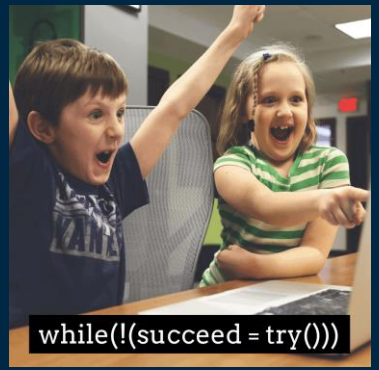
- Small code that verifies parts of your main code.
- Unit could be a class, function, module, API etc.
- Unit test will verify the class, function.. Is working as expected and delivers the expected output.
- Allows developer to freely change or improve the code, make sure it didn’t break anything by 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.

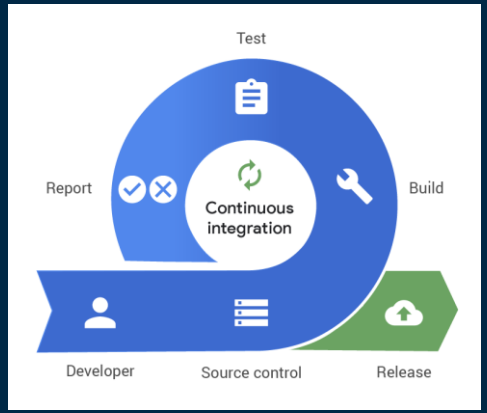


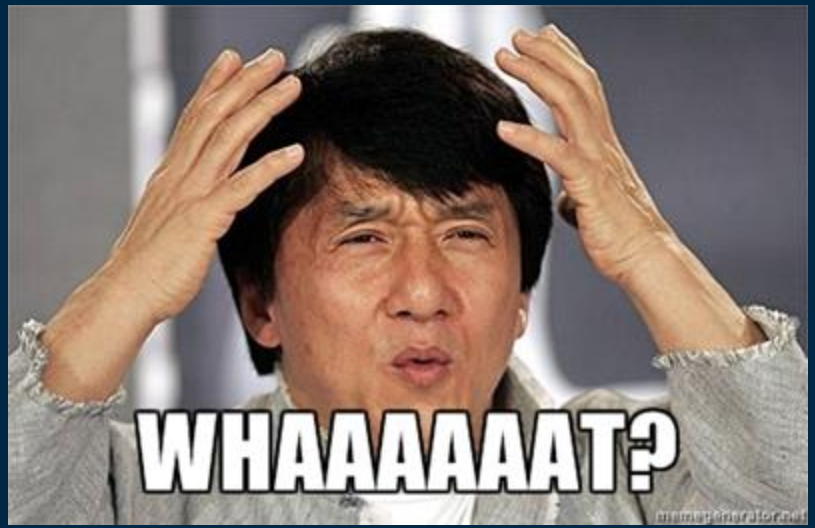
Version controlling

- 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?