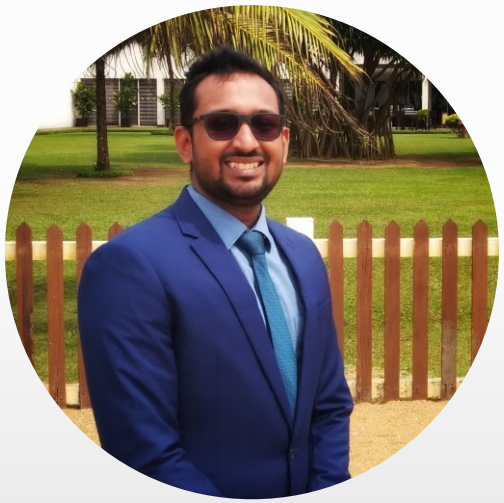
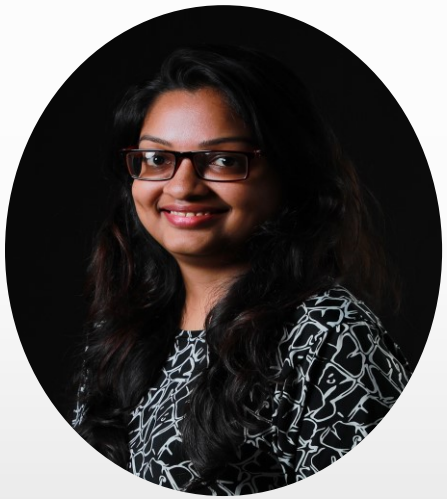
Application Frameworks

Introduction

Meet the Crew





Overview

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

What is Application Frameworks

- 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 a popular NoSQL database will be delivered as well.

Objectives

- Deliver industry best practices and principles in software engineering and encourage students to use them in their development.
- Let students to discover new trends in two industry leading software development languages.
- Deliver an introduction to NoSQL databases and how to use them in full stack development.
- Introduce students to leading frameworks in web applications and web services development along with leading architectures and authentication mechanisms being followed in the industry.

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

Things to be Covered

Evaluation

- Technical Blog – 4%
  - Due on 14th week
- React Lab Assignment – 10%
  - 7th week
- Mid-Term Evaluation – 10%
  - 8th week
- Hackathon – 10%
  - 9th week
- Angular Lab Assignment – 6%
  - 12th week
- Group Project – 20%
  - Due on 14th week
- Final Examination – 40%

Evaluation

General Assignment and viva policies

- All students are expected to submit assignments within the given deadline.
- There will be penalties for late submissions
  - Upto one week delay – 25% of marks will be deducted
  - Upto two weeks delay – 50% of marks will be deducted (This is only for assignment submissions before the 12th week)
  - Beyond this zero marks will be awarded
- Note : Assignments which have deadlines on the 12th and 13th weeks of the semester will entertain only upto a one week delay
- Vivas/Presentations/Online Exams are treated as formal exams. No additional dates will be provided. You will get the dates of vivas and presentations at least one month before.

Principles

Principles

- S.O.L.I.D
- Guidelines - Approaching the solution
- Guidelines - Implementing the solution

S.O.L.I.D

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

Single Responsibility Principle

- A class should have one, and only one, reason to change.
- It makes your software easier to implement and change.
- A class should be modified as soon its responsibilities changes.
- Fewer testing, lower coupling and microservices organization.

Open/Close Principle

- Software entities should be open for extension, but closed for modification.
- Ability of adding new functionality without changing the existing code.
- Restrict to modify existing code and causing potential new bugs.
- Enables loose coupling.

Liskov Substitution Principle

- Objects of a superclass shall be replaceable with objects of its subclasses.
- Requires the objects of subclass to behave in the same way as the objects of its superclass.
- An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass.
- The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass.

Interface Segregation Principle

- Clients should not be forced to depend upon interfaces that they do not use.
- Easier to violate the concept, especially when the software grows eventually when the new requirements arise and have to add more and more features.
- Larger interfaces should be split into independent components to ensure that implementing classes only need to be concerned about the methods that are of interest to them.
- Reduces the side effects and frequency of required changes.

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- Dependency Inversion Principle can be achieved by following both Open/Closed Principle and the Liskov Substitution Principle.
- Reduces coupling between different pieces of code.

Approaching the solution

Approaching the solution

- 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 such 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).
- The solution of all smaller sub problems is finally merged in order to obtain the solution of the original problem.

KISS

- Keep It Simple and Stupid.
- A simple solution is better than a complex one, even if the solution looks stupid.
- States that designs and/or systems should be as simple as possible.
- Can be used in a variety of disciplines, such as interface design, product design, and software development.

Learn from mistakes

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

Reasons for software exists

- Keep in mind the bigger picture, why this software exists.
- Loss of the bigger picture might cause following a wrong path.
- Try to separate into small, independent and individual components, don't enhance the complexity.
- Always consider from the user perspective than the domain perspective.

You won’t be using the software

- End-user is not that technically capable as same as you are.
- Do not assume that user will understand the logic.
- Do not assume the user is a domain expert.
- User friendliness and user experience matters.

Implementing the solution

Implementing the solution

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

YAGNI - You aren't gonna need it

- A phrase extracted from Extreme Programming that's often used generally in agile software teams.
- Do write code that is no use in present while guessing that will come in handy in the future.
- Future requirements always change. This is a waste of time.
- Write the code you need for the moment only that.

DRY - Don’t repeat yourself

- Duplication in logic should be eliminated via abstraction.
- Duplication in process should be eliminated via automation.
- Always reuse code you write.
- Code as best as possible. Keep it generalize and reusable.

Embrace abstraction

- Most powerful concept in object oriented programming.
- Promotes code quality, reusability, modularity and maintainability.
- 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

- Don’t need to design a solution from nothing when a solution already exists.
- Someone else might have already solved the same problem. Make use of that.
- You could just take the concept of a “wheel” and tweak it to fit your needs; save time, save effort, and save yourself from a headache.
- Consider if any areas of your current project can use a solution that has already been created by others.

Write code that does one thing well

- A method with just few lines of code is quite typical of well designed app/library.
- Single piece of code that do one thing and that one thing really well.
- Spend less time trying to figure out what certain code segments actually do and more time on fixing, revising, extending, etc.
- Reduces the likelihood of misunderstandings between the team members, which also means fewer bugs in the long run.

Debugging is harder than writing the code

- People are not smart enough to debug their own code.
- A lot of beginners write code that is too complex for them to debug themselves over the years.
- This happens because they try to write too much code at once before testing any of it.
- Make it readable as much as possible. Readable code is better than compact code.

Kaizen - Leave it better than when you found it

- Create a culture of continuous improvement where all employees are actively engaged in improving the company.
- Fix not just the bug but the code around it.
- Constantly asks team members to evaluate their own work, and to help review the work of their peers.
- By reviewing work every now and then, teams tend to detect errors and shortcomings at the right time.

Practices

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.
- They are called "best practices" not because we can precisely quantify their value but rather they are observed to be commonly used in industry by successful organizations.

Unit Testing

- 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 won'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 and improve the performance.
- 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.

Questions

Until Next Week

- Start the technical blog.
- Create your GIT account.
- Form your 4 member group for the Hackathon and Group Project.