1 GIT (1 day)

- Solid knowledge of how source control works, why it's important, using either CLI or GUI tools
- Know the basic commands: clone, push, fetch, pull, commit
- Understand what remotes are
- Understands how to hide, add, remove, and stash files
- Understands how to branch and merge
- Understands how to resolve merge conflicts Can reliably commit based on project needs, with communication to the rest of the team
- Uses relevant and important information in commit notes
- Understands differences between HTTPS SSH validation

2 FRONT END DEVELOPMENT

HTML & CSS

- Solid knowledge of common tag types and how to implement them
- Understands how the browser handles conflicting styles
- Understands how the DOM is laid out and is structured
- Understands the difference between content & styling, and how each technology is used i those regards
- Understands that different browsers render and interpret things slightly differently and the need to test on commonly used platform
- Understand difference between database and library

JavaScript

- Solid understanding of HTML & CSS
- Understanding of browser runtime & where to inject scripts
- Understanding of dynamic typing & how it makes using the language quick and easy
- Understanding of Document Object Model
- Knowledge of JSON and working with JSON objects
- Knowledge on how to traverse the DOM for element handles
- Know how to structure native javascript classes (OOP, Pub/Sub) using prototype or object literal statements
- Understanding asynchronous concepts & implications, such as callbacks

AJAX

- Ability to implement AJAX calls in a project
- Understanding the difference between HTTP methods (get, post, load, etc) and their uses

- Understanding of error handling and Promises

HTTP Methods

- Understands difference between HTTP Methods POST, GET, and PUT
- Understands difference between AJAX and HTTP

Framework vs. Library

- Angular/React/Vue/ (build app in two different ways)

3 DATABASE DEVELOPMENT

- Understands the difference between a relational database and document based database.

CRUD operations

- Able to interact with Firebase documents, & database collections

SQL

- Can create tables, columns & stored procedures in a timely manner
- Can take business logic and turn it into a database design
- Knowledge of how to make a primary key or foreign key & how all the relationship's work
- Can create reliable, tested queries that efficiently return data
- Comfortable with CLI and/or GUI tools
- Can write repository methods that perform basic CRUD operations

NoSQL- MONGO DB

4 BACK END DEVELOPMENT

C#

- Solid understanding of object-oriented programming in C#
- Knowledge of Visual Studio, can identify the Solution Explorer, the debugging menu, the build run and clean functions & unit testing
- Understanding all common modifiers (static, abstract, etc) & can build and implement classes correctly.
- Understanding namespaces & how to import dependencies

- Knowledge on how to use model binding
- Knowledge of serialization and deserialization with JSON
- Understanding the difference between pass by reference and pass by value

Node

HTML elements

- Before writing a function or class, be able to utilize existing functions or classes

APIs

- Create modern, REST APIs from existing information assets silos
- Secure and authorize information assets exposed via APIs
- Understanding C.R.U.D

5 Best Practices

In a collaborative setting, a code review should observe the following practices:

Code formatting

- While going through the code, check the code formatting to improve readability and ensure that there are no blockers.
- Understand how to use code linter

Architecture

The code should follow the defined architecture.

- Separation of Concerns followed
- Split into multiple layers and tiers as per requirements (Presentation, Business and Data layers).
- Split into respective files (HTML, JavaScript and CSS).
- Code is in sync with existing code patterns/technologies.
- Design patterns: Use appropriate design pattern (if it helps), after completely understanding the problem and context.

Coding best practices

- No hard coding, use constants/configuration values.
- Group similar values under an enumeration (enum).

- Comments – Do not write comments for what you are doing, instead write comments on why you are doing. Specify about any hacks, workaround and temporary fixes. Additionally, mention pending tasks I your to-do comments, which can be tracked easily.
- Avoid multiple if/else blocks.
- Use framework features, wherever possible instead of writing custom code.

Non Functional requirements

a) Maintainability (Supportability) – The application should require the least amount of effort to support in near future. It should be easy to identify and fix a defect.

- Readability: Code should be self-explanatory. Get a feel of story reading, while going through the code. Use appropriate name for variables, functions and classes. If you are taking more time to understand the

code, then either code needs refactoring or at least comments have to be written to make it clear.

- Testability: The code should be easy to test. Refactor into a separate function (if required). Use interfaces while talking to other layers, as interfaces can be mocked easily. Try to avoid static functions, singleton classes as these are not easily testable by mocks.
- Debuggability: Provide support to log the flow of control, parameter data and exception details to find

the root cause easily. If you are using Log4Net like component then add support for database logging also, as querying the log table is easy.

- Configurability: Keep the configurable values in place (XML file, database table) so that no code changes are required, if the data is changed frequently.

b) Reusability

- DRY (Do not Repeat Yourself) principle: The same code should not be repeated more than twice.
- Consider reusable services, functions and components.
- Consider generic functions and classes.

c) Reliability – Exception handling and cleanup (dispose) resources.

d) Extensibility – Easy to add enhancements with minimal changes to the existing code. One component should be easily replaceable by a better component.

e) Security – Authentication, authorization, input data validation against security threats such as SQL injections and Cross Site Scripting (XSS), encrypting the sensitive data
(passwords, credit card information etc.)

f) Performance

- Use a data type that best suits the needs such as StringBuilder, generic collection classes.
- Caching and session/application data.

g) Scalability – Consider if it supports a large user base/data? Can this be deployed into web farms?

h) Usability – Put yourself in the shoes of a end-user and ascertain, if the user interface/API is easy to understand and use. If you are not convinced with the user interface design, then start discussing your ideas with the business analyst.

6. Object-Oriented Analysis and Design (OOAD) Principles/SOLID

- Single Responsibility Principle (SRS): Do not place more than one responsibility into a single class or function, refactor into separate classes and functions.
- Open Closed Principle: While adding new functionality, existing code should not be modified. New functionality should be written in new classes and functions.
- Liskov substitutability principle : The child class should not change the behavior (meaning) of the parent class. The child class can be used as a substitute for a base class.
- Interface segregation: Do not create lengthy interfaces, instead split them into smaller interfaces based on the functionality. The interface should not contain any dependencies (parameters), which are not required for the expected functionality.
- Dependency Injection: Do not hardcode the dependencies, instead inject them.