

# Software Quality Report: Week-1

## 1. Introduction

Software quality refers to the ability of a software product to meet user requirements, perform efficiently, and maintain reliability over time. Qualityful software does not merely work — it is **dependable, maintainable, and easy to understand**.

This report discusses the core principles and practices behind creating qualityful software, focusing on key ideas such as **managing complexity**, the **KISS principle**, the role of **design and process**, and the team main concept of **no surprises**.

## 2. The Key to Software Quality: Managing Complexity

As software systems grow in scale and functionality, complexity becomes one of the greatest threats to quality. Managing complexity effectively ensures that systems remain **reliable, scalable, and maintainable**.

### 2.1 Understanding Complexity

Complexity arises when a system contains too many interconnected components, unclear logic, or overlapping responsibilities. Such systems are difficult to modify, test, or debug.

## 2.2 Managing Complexity

Effective methods to manage complexity include:

- **Decomposition:** Divide the system into smaller, manageable modules.
- **Abstraction:** Hide unnecessary implementation details behind interfaces.
- **Consistency:** Maintain uniform coding and architectural standards.
- **Documentation:** Ensure every component is well explained for future maintenance.

By controlling complexity, developers make software more predictable, testable, and sustainable.

### 3. The Core Principle of Qualityful Software: KISS

#### *Keep It Simple, Stupid (KISS)*

The KISS principle emphasizes **simplicity** as the foundation of good software engineering. Overly complicated solutions may appear clever but often lead to confusion and defects.

### 3.1 Importance of Simplicity

Simplicity contributes to:

- **Ease of maintenance:** Less complex code is easier to modify and debug.
- **Reduced risk:** Simple designs have fewer potential points of failure.
- **Better performance:** Fewer layers and abstractions reduce overhead.
- **Improved collaboration:** Clear and straightforward logic helps all team members understand the system.

## 3.2 Applying KISS in Practice

Developers should:

- Avoid overengineering.
- Build only what is necessary.
- Choose clarity over cleverness.
- Keep architectures clean and well-defined.

## 4. Tools of Qualityful Software: Design and Process

Achieving software quality requires more than principles — it requires practical tools and structured practices. Two key tools are **Design** and **Process**.

### 4.1 Design

Design is the architectural backbone of a system. A well-designed system aligns with user needs and technical feasibility.

### 4.1.1 Characteristics of Good Design

- **Clarity:** Each component has a clear role.
- **Modularity:** The system is divided into logical, reusable units.
- **Scalability:** The design supports future growth.
- **Consistency:** Common design patterns are used throughout the project.
- **Readability:** The system is understandable even to new team members.

## 4.2 Process

Process defines how the team plans, develops, and delivers software.

A disciplined process ensures **stability, transparency, and continuous improvement**.

### 4.2.1 Importance of Process

A well-structured process:

- Prevents chaos in team coordination.
- Promotes accountability and progress tracking.
- Reduces confusion and unplanned risks.
- Ensures that the project remains aligned with goals.

#### **4.2.2 Stages of a Good Software Process**

- 1. Planning:** Identify objectives and allocate resources.
- 2. Design and Development:** Translate requirements into structured software.
- 3. Testing:** Detect and resolve issues before deployment.
- 4. Review and Feedback:** Encourage communication and iteration.
- 5. Deployment and Maintenance:** Keep the system updated and secure.

## 5. Team Dynamics: The “No Surprises” Principle

Team collaboration is the human side of software quality. A team guided by the “No Surprises” principle maintains transparency, trust, and predictability.

### 5.1 Meaning of “No Surprises”

The idea is simple — every team member should know what’s happening at all times. There should be no hidden changes, assumptions, or unexpected outcomes.

### 5.2 Achieving No Surprises

- Communicate regularly and clearly.
- Review code and design changes collaboratively.
- Document decisions and changes immediately.

## 6. Conclusion

Building **qualityful software** is not about writing perfect code — it's about creating systems that are **manageable, simple, and transparent**.

By managing complexity, following the KISS principle, establishing strong design and process structures, and maintaining open teamwork with no surprises, developers can build software that stands the test of time.

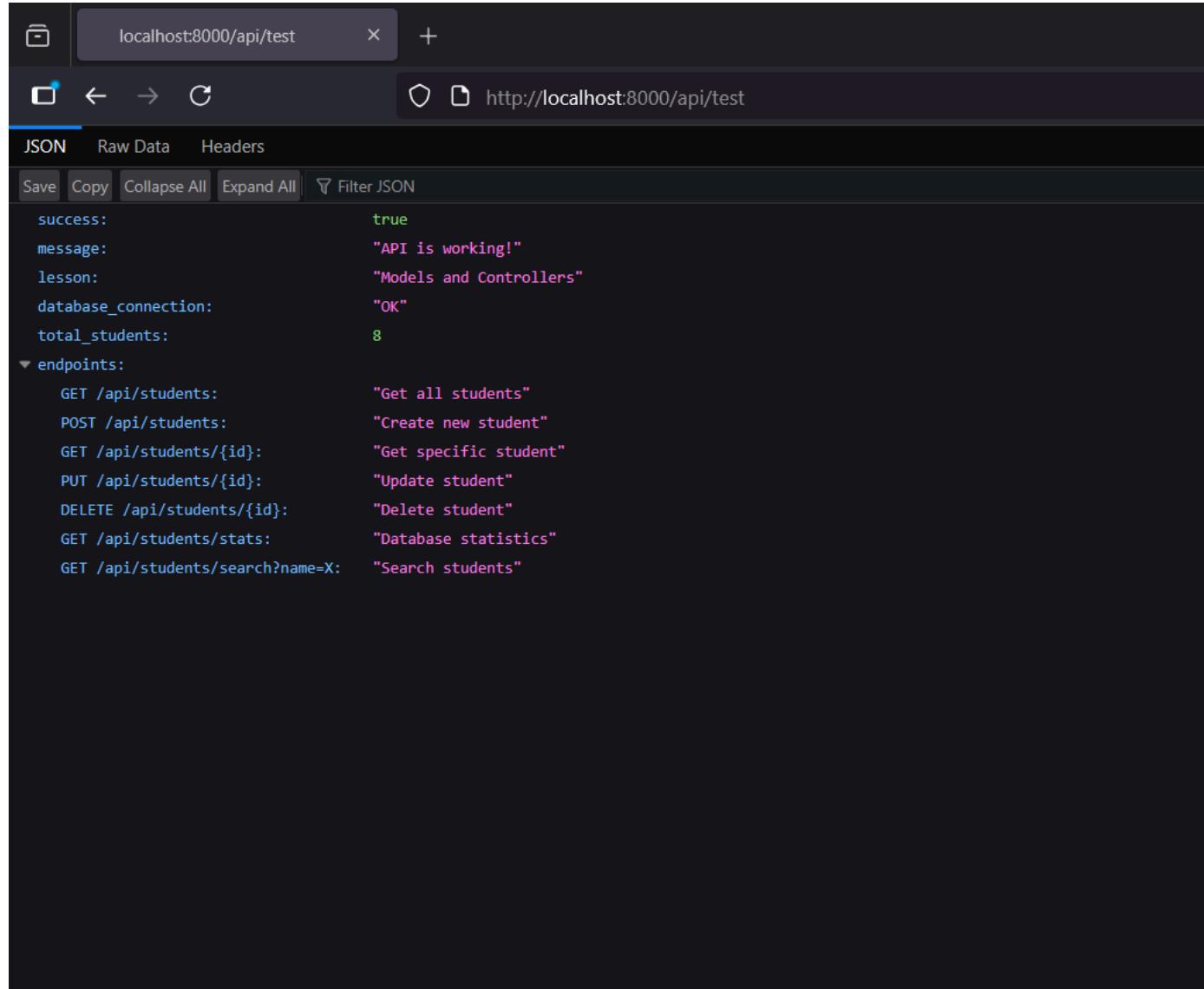


# Software Engineering Concepts and Web Integration Report: Week-2

## ❖ PHP and REST API with Bearer Token and NGINX

My project Student Management System integrates **PHP** with **11 RESTful APIs** that use **Bearer Token authentication** for secure communication between clients and servers. This is deployed using **NGINX**, a high-performance web server known for its efficiency, scalability, and reverse proxy capabilities.

Here is a example of working api which is deployed by NGINX:



A screenshot of a browser developer tools Network tab, specifically the 'JSON' tab, showing a successful API response. The URL is `localhost:8000/api/test`. The response body is a JSON object with the following structure:

```
success: true
message: "API is working!"
lesson: "Models and Controllers"
database_connection: "OK"
total_students: 8
endpoints:
  GET /api/students: "Get all students"
  POST /api/students: "Create new student"
  GET /api/students/{id}: "Get specific student"
  PUT /api/students/{id}: "Update student"
  DELETE /api/students/{id}: "Delete student"
  GET /api/students/stats: "Database statistics"
  GET /api/students/search?name=X: "Search students"
```



## Software Process

The **software process** defines a structured set of activities required to develop software systems. It includes planning, design, implementation, testing, and maintenance.



## Agile Model

The **Agile model** is an iterative and incremental software development approach emphasizing flexibility, collaboration, and continuous feedback. Agile breaks projects into small, manageable iterations called **sprints**, enabling teams to adapt quickly to changing requirements and deliver functional software frequently.



## Spiral Model

The **Spiral model** combines iterative development with systematic risk analysis. It consists of four main phases: planning, risk analysis, engineering, and evaluation repeated in a spiral loop. This model is ideal for large and complex projects requiring frequent refinement.

## 💧 Waterfall Model

The **Waterfall model** follows a **linear sequential** approach where each phase must be completed before moving to the next. It is one of the earliest software models and works best for projects with well-defined requirements.



## Scrum

**Scrum** is a framework under the Agile methodology that focuses on **team collaboration and iterative progress**.

Work is divided into **sprints**, during which teams deliver small, functional increments of the product.

### Scrum Roles:

- **Product Owner:** Defines goals and priorities
- **Scrum Master:** Facilitates the process
- **Development Team:** Builds the product



## Agile vs Scrum

While **Agile** is a philosophy that guides software development with principles of flexibility and collaboration, **Scrum** is a practical framework that applies these principles.

Agile promotes adaptability and continuous delivery, whereas Scrum structures the workflow into **fixed-length sprints** with defined roles and ceremonies.

In essence, Scrum is a subset of Agile that provides a structured way to implement Agile values through team-based iteration cycles.



## Software Engineering Rules Applied - Process

Software engineering applies systematic rules and guidelines to ensure high-quality outcomes. Rules such as proper documentation, testing, version control, and modularization maintain consistency and scalability. Following these principles ensures that the system can evolve and integrate new features without compromising performance.



## Divide and Conquer

The **Divide and Conquer** strategy involves breaking down complex problems into smaller, manageable parts. Each subproblem is solved individually, and the results are combined to form the final solution. This principle is widely used in both **algorithm design** and **software architecture**, improving maintainability and performance.

## SRP (Single Responsibility Principle)

The **Single Responsibility Principle (SRP)** states that every module or class should have one, and only one, reason to change. In practice, this means each component in the system should perform one specific function. SRP enhances code readability, testability, and modularity, reducing maintenance costs over time.

## Conclusion

Through the combination of structured software processes and modern web development practices like **PHP REST APIs** deployed with **NGINX**, we achieve a balance between agility, maintainability, and performance. Models such as Agile, Scrum, and Spiral provide flexibility and risk management, while principles like **SRP** and **Divide and Conquer** ensure sustainable, high-quality codebases.



# Laravel, Python, OOP, and UML: Week-3

# 1. Laravel Installation

Laravel is a modern PHP web framework used for building scalable, maintainable web applications. It follows the **Model-View-Controller (MVC)** design pattern and includes tools for routing, authentication, caching, and database management.

## Steps to Install Laravel

### 1. Install PHP and Composer

- Laravel requires PHP  $\geq$  8.1 and [Composer](#) for dependency management.
- Check installation:

```
php -v  
composer -v
```

# Example of Laravel installed

```
faiza@DESKTOP-T1L4RHD:~/ + ▾
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

  INFO  Discovering packages.

laravel/pail ..... DONE
laravel/sail ..... DONE
laravel/sanctum ..... DONE
laravel/tinker ..... DONE
nesbot/carbon ..... DONE
nunomaduro/collision ..... DONE
nunomaduro/termwind ..... DONE

81 packages you are using are looking for funding.
Use the 'composer fund' command to find out more!
> @php artisan vendor:publish --tag=laravel-assets --ansi --force

  INFO  No publishable resources for tag [laravel-assets]. 

No security vulnerability advisories found.

  INFO  Published API routes file.

  INFO  Preparing database.

Creating migration table ..... 33.28ms DONE

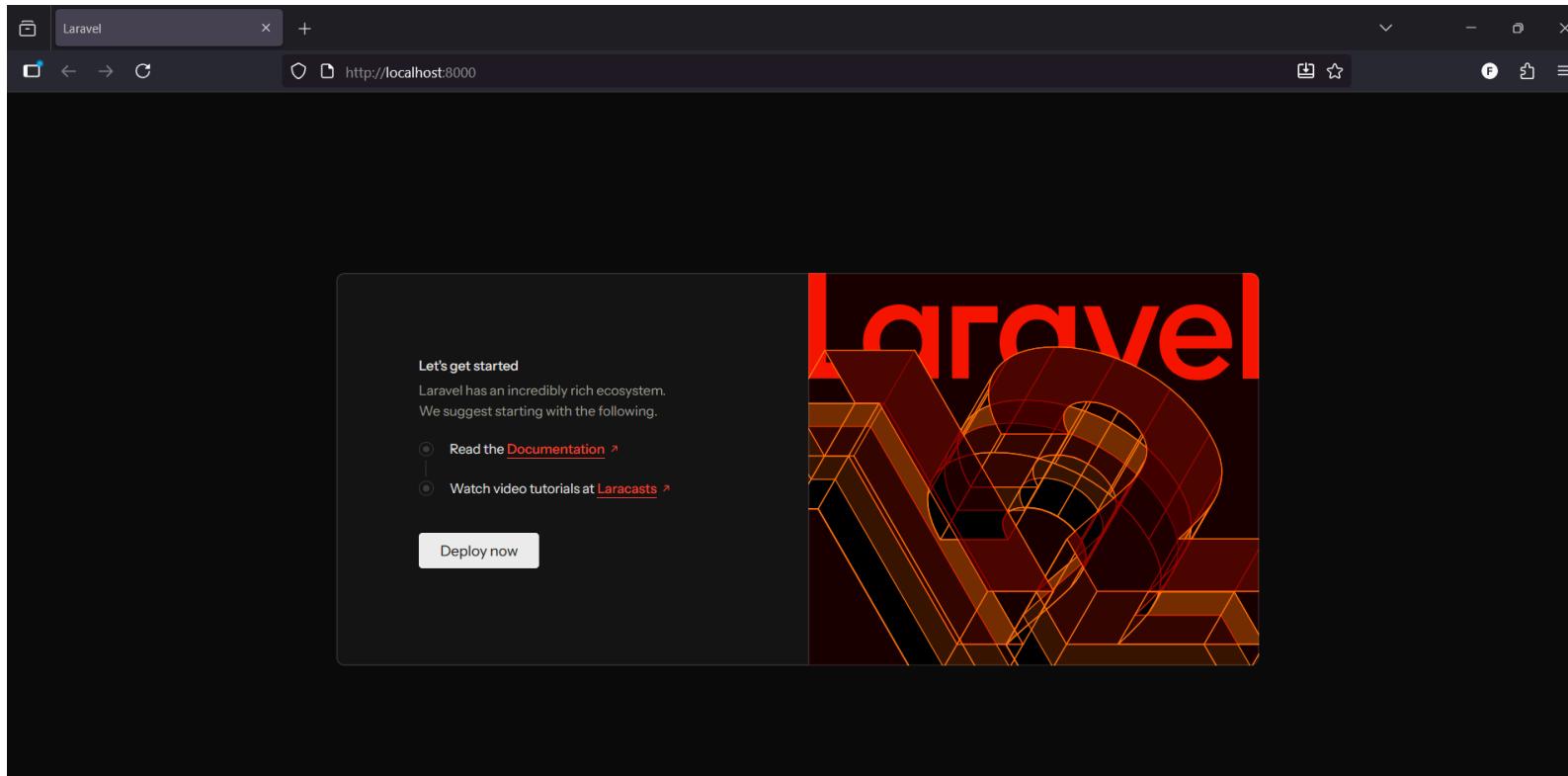
  INFO  Running migrations.

0001_01_01_000000_create_users_table ..... 154.19ms DONE
0001_01_01_000001_create_cache_table ..... 75.83ms DONE
0001_01_01_000002_create_jobs_table ..... 226.81ms DONE
2025_10_25_235517_create_personal_access_tokens_table ..... 102.97ms DONE

  INFO  API scaffolding installed. Please add the [Laravel\Sanctum\HasApiTokens] trait to your User model.

Copying Controllers...
Copying Models...
Copying routes/api.php with test routes...
Copying DB migrations/seeders...
```

## Example of Laravel page



## 2. Create a New Laravel Project

```
composer create-project laravel/laravel student-management
```

## 3. Navigate to the Project Folder

```
cd student-management
```

## 4. Start the Local Development Server

```
php artisan serve
```

Visit: <http://127.0.0.1:8000>

## 2. ORM Installation and Usage (Eloquent ORM)

Laravel uses **Eloquent ORM (Object-Relational Mapping)** to interact with databases using models instead of writing raw SQL queries.

### ⚙️ Installing ORM (Already included in Laravel)

Eloquent ORM comes pre-installed with Laravel. You just need to configure your **Model** and **Database** connection.

### ✳️ Example: Defining and Using a Model

#### 1. Create a Model

```
php artisan make:model Student -m
```

The `-m` flag also creates a migration file.

## 2. Migration Example

```
Schema::create('students', function (Blueprint $table) {  
    $table->id();  
    $table->string('name');  
    $table->string('email')->unique();  
    $table->timestamps();  
});
```

## 3. Run Migration

```
php artisan migrate
```

## 4. Using the Model

```
$student = new Student();
$student->name = "Faiza";
$student->email = "faiza@example.com";
$student->save();
```

### 3. Database Setup in Laravel

Laravel supports multiple databases like MySQL, PostgreSQL, and SQLite.

#### Steps to Configure Database

##### 1. Edit the `.env` File

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=student_db
DB_USERNAME=root
DB_PASSWORD=
```

## 2. Check Database Connection

Run:

```
php artisan migrate
```

If successful, your Laravel app is connected to the database.

## 3. Use Artisan Tinker for Database Testing

```
php artisan tinker  
>>> Student::all();
```

## Example of Database in laravel

```
faiza@DESKTOP-T1L4RHD:~/m + | - X
faiza@DESKTOP-T1L4RHD:~$ bash mysql_root_password.sh
bash: mysql_root_password.sh: No such file or directory
faiza@DESKTOP-T1L4RHD:~$ cd "/mnt/d/MS CSE/Software Quality/Course Materials/ase230-main/module2/code/1_Laravel/2. Laravel Installation and Project Structure"
faiza@DESKTOP-T1L4RHD:/mnt/d/MS CSE/Software Quality/Course Materials/ase230-main/module2/code/1_Laravel/2. Laravel Installation and Project Structure$ bash mysql_root_password.sh
[sudo] password for faiza:
Enter password:
faiza@DESKTOP-T1L4RHD:/mnt/d/MS CSE/Software Quality/Course Materials/ase230-main/module2/code/1_Laravel/2. Laravel Installation and Project Structure$ bash mysql_user.sh
Creating MySQL user for Laravel...
Enter password:
Done! Add this to your Laravel .env file:
DB_DATABASE=laravel_app
DB_USERNAME=laravel_user
DB_PASSWORD=password123
faiza@DESKTOP-T1L4RHD:/mnt/d/MS CSE/Software Quality/Course Materials/ase230-main/module2/code/1_Laravel/2. Laravel Installation and Project Structure$
```

# Example of Database setup

```
faiza@DESKTOP-T1L4RHD: ~/ + ^

Unpacking dos2unix (7.5.1-1) ...
Setting up unzip (6.0-28ubuntu4.1) ...
Setting up dos2unix (7.5.1-1) ...
Processing triggers for man-db (2.12.0-4build2) ...
== Verification ==

Installation Summary:
=====
✓ PHP: PHP 8.3.6 (cli) (built: Jul 14 2025 18:30:55) (NTS)
PHP version 8.3.6 (/usr/bin/php8.3)
Run the "diagnose" command to get more detailed diagnostics output.
✓ Composer: Composer version 2.8.12 2025-09-19 13:41:59
✓ MySQL: mysql Ver 8.0.43-0ubuntu0.24.04.2 for Linux on x86_64 ((Ubuntu))

== Checking MySQL Service ==
Checking MySQL port (WSL2):
[sudo] password for faiza:
LISTEN 0      151          127.0.0.1:3306      0.0.0.0:*      users:(("mysqld",pid=20717,fd=25))

LISTEN 0      70           127.0.0.1:33060     0.0.0.0:*      users:(("mysqld",pid=20717,fd=21))

MySQL service status:
active

== File Conversion Setup ==

== MySQL Configuration Instructions ==
🔧 Next Steps for MySQL Setup:
1. Secure MySQL installation (recommended):
   sudo mysql_secure_installation

2. Set root password manually:
   sudo mysql -u root
   Then run these SQL commands:
   ALTER USER 'root'@'localhost' IDENTIFIED BY '123456';
   FLUSH PRIVILEGES;
   EXIT;
```

## 4. What is Python?

Python is a **high-level, interpreted programming language** known for its readability and flexibility. It supports multiple paradigms, including **procedural, object-oriented, and functional programming**.

### ✨ Key Features

- Easy syntax similar to English
- Dynamic typing
- Rich standard library
- Widely used in AI, ML, Web Dev, and Automation



## Example:

```
name = "Faiza"  
print("Hello,", name)
```

## Output:

```
Hello, Faiza
```

## 5. Python OOP (Object-Oriented Programming)

OOP in Python helps structure code into **classes** and **objects** to promote reusability and modularity.

### Key OOP Concepts

In Object-Oriented Programming (OOP), a class serves as a blueprint for creating objects, defining their attributes and behaviors.

An object is an instance of a class.

Encapsulation involves hiding internal details of an object.

Inheritance allows a class to reuse and extend code from another class.

Lastly, polymorphism enables the same method to behave differently across classes.



## Example:

```
class Student:  
    def __init__(self, name):  
        self.name = name  
  
    def greet(self):  
        return f"Hello, I am {self.name}"  
  
s1 = Student("Faiza")  
print(s1.greet())
```

## Output:

```
Hello, I am Faiza
```

## 6. UML (Unified Modeling Language)

UML is a **visual modeling language** for designing and documenting software systems, especially object-oriented ones.

### Purpose of UML

- Visualize system structure and behavior
- Plan software architecture
- Communicate design between teams

## Types of UML Diagrams

UML diagrams are generally divided into two main categories: structural and behavioral.

Structural diagrams, such as Class, Object, and Component diagrams, illustrate the static aspects of a system.

On the other hand, behavioral diagrams, including Use Case, Sequence, and Activity diagrams, focus on the dynamic aspects of the system.

### Example: Class Diagram Components

- **Class Name:** Student
- **Attributes:** name, email
- **Methods:** enroll(), attend\_class()

# **Software Design and Development Concepts: Week-4**

# 1. Object-Oriented Programming (OOP) for Software Design

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which can contain data and methods. OOP helps in organizing software design around **data and behavior**, rather than just functions and logic. It encourages modularity, code reuse, and scalability, making it ideal for complex software projects. Key principles of OOP include **Abstraction, Encapsulation, Inheritance, Polymorphism, and Composition**.

## 2. Requirements in Software Development

Requirements define what a system should do and the constraints under which it must operate. They serve as a blueprint for software design, implementation, and testing.

- **Functional requirements** describe system behavior or functionalities.
- **Non-functional requirements** describe system qualities like performance, usability, and security.

### 3. Abstraction and Inheritance

- **Abstraction:** Abstraction is the process of exposing only the necessary details of an object while hiding the internal complexity. It allows developers to focus on **what an object does** rather than **how it does it**.

*Example:* A `Vehicle` class may have methods like `start()` and `stop()`, without exposing the internal engine mechanics.

- **Inheritance:** Inheritance allows a class to **derive properties and behaviors from another class**, promoting code reuse and hierarchy.

*Example:* A `Car` class can inherit from the `Vehicle` class and reuse its methods while adding car-specific features like `airConditioning()`.

## 4. Encapsulation and Dependency Injection

- **Encapsulation:** Encapsulation is the practice of **restricting direct access** to an object's **data** and exposing it only through controlled methods. It protects the internal state and prevents unintended interference.

*Example:* Using private variables `_balance` and public getter/setter methods in a `BankAccount` class.

- **Dependency Injection (DI):** DI is a design pattern where a class receives its dependencies from an external source rather than creating them internally. This reduces **tight coupling** and enhances testability.

*Example:* Passing a `DatabaseService` object to a `UserService` constructor instead of instantiating it inside the class.

## 5. Polymorphism and Composition

- **Polymorphism:** Polymorphism allows objects of different classes to respond to the same method call in a class-specific way. It enables **flexible and extensible code**.

*Example:* `Dog` and `Cat` classes both implement `makeSound()`. Calling `makeSound()` on a `Dog` object outputs "Woof", while on a `Cat` object outputs "Meow".

- **Composition:** Composition is a design principle where a class is composed of one or more objects of other classes to achieve complex behavior. It is often preferred over inheritance for flexibility.

*Example:* A `Car` object contains `Engine`, `Wheel`, and `Seat` objects instead of inheriting from them.

## 6. Software Testing

Software testing ensures that the system behaves as expected. It includes:

- **Unit Testing:** Testing individual components for correctness.
- **Integration Testing:** Ensuring that combined modules interact correctly.
- **System Testing:** Testing the complete system in an environment simulating production.
- **Acceptance Testing:** Validating the system against user requirements.

## 7. SOLID Principles

SOLID is a set of principles to write **maintainable** and **scalable** object-oriented software.

- **S - Single Responsibility Principle (SRP)**: A class should have **only one reason to change**, meaning it should have a single responsibility.
- **O - Open/Closed Principle (OCP)**: Software entities should be **open for extension but closed for modification**, allowing behavior to be extended without altering existing code.
- **L - Liskov Substitution Principle (LSP)**: Subclasses must be **substitutable for their base classes** without breaking functionality.

- **I - Interface Segregation Principle (ISP):** Many specific interfaces are better than a single general-purpose interface.
- **D - Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules; both should depend on abstractions.

### Example:

- **SO** (Single Responsibility + Open/Closed) ensures that each class is focused on one task but can be extended without modification.
- **LID** (Liskov + Interface Segregation + Dependency Inversion) encourages safe inheritance, minimal interface exposure, and decoupled design.

## Conclusion

This report covers key software engineering concepts including OOP design, requirements analysis, fundamental OOP principles (Abstraction, Inheritance, Encapsulation, Polymorphism, Composition), software testing, and SOLID principles. Applying these practices leads to modular, maintainable, and scalable software solutions.

# High-Level Languages, JavaScript, and TypeScript: Week-5

# What Is a High-Level Programming Language?

High-level programming languages are designed to be easy for humans to understand and write. Unlike low-level languages that directly interact with hardware, high-level languages provide abstraction so developers think in terms of logic instead of machine operations.

## Key Features

- Human-readable syntax
- Portable across systems
- Automatic memory management
- Built-in libraries
- Error-handling mechanisms

# What Is JavaScript?

JavaScript (JS) is a **high-level, dynamic scripting language** used to make web pages interactive. It runs inside all browsers and is essential for web development.

## Where JavaScript Is Used

- Frontend web development
- Backend (Node.js)
- Mobile apps (React Native)
- Desktop apps (Electron)
- Game development

# Pros & Cons of JavaScript

## ✓ Pros

- Easy to learn
- Runs on all browsers
- Large community
- Fast development

## ✗ Cons

- No type safety
- Runtime errors
- Difficult for large projects

# JavaScript Basics: Examples

## Print a Message

```
console.log("Hello World!");
```

## Add Numbers

```
let x = 5;  
let y = 8;  
console.log(x + y);
```

## Modify HTML

```
document.getElementById("title").innerText = "Welcome Home";
```

# Build a To-Do App in JavaScript

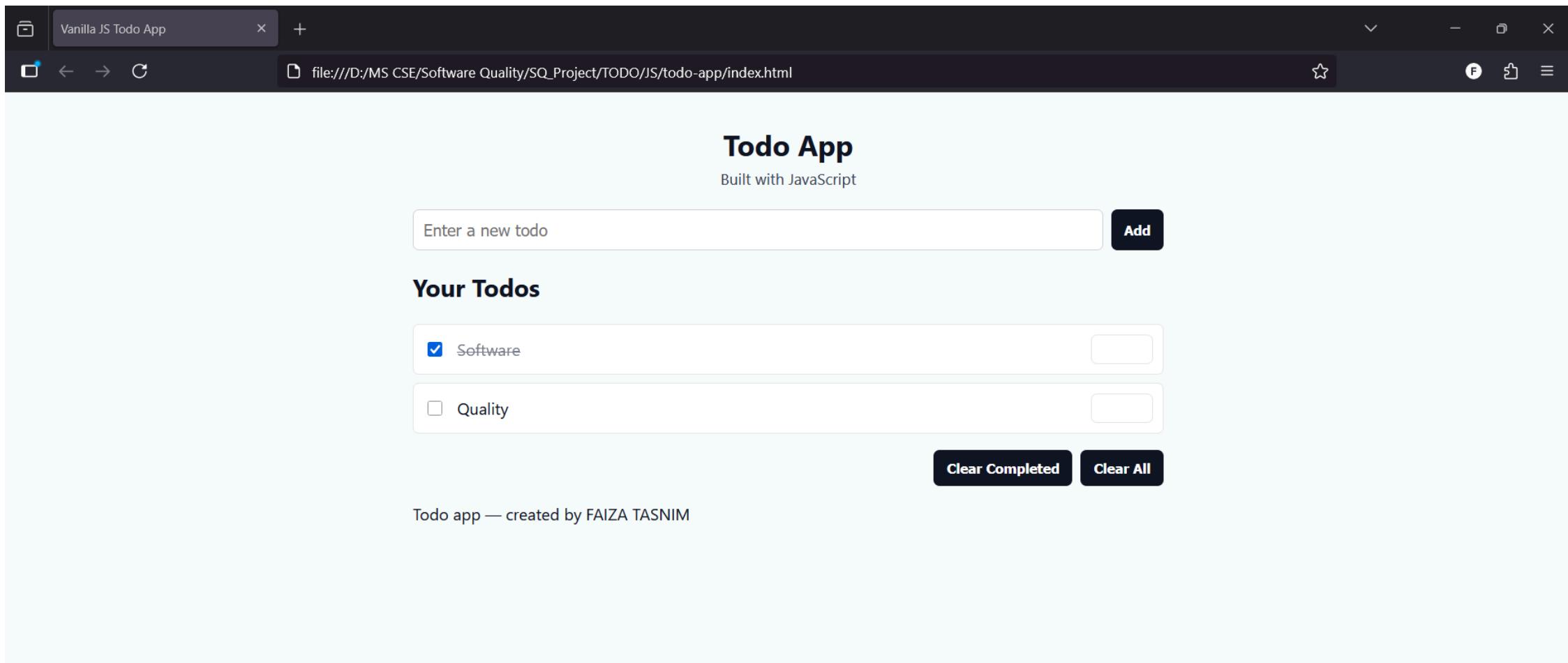
## HTML

```
<input id="taskInput" placeholder="Add a task" />
<button onclick="addTask()">Add</button>
<ul id="taskList"></ul>
```

## JavaScript

```
function addTask() {
  const input = document.getElementById("taskInput");
  const task = input.value.trim();
  if (!task) return;
  const li = document.createElement("li");
  li.textContent = task;
  document.getElementById("taskList").appendChild(li);
  input.value = "";
}
```

# Example of Javascript ToDo



# What Is TypeScript?

TypeScript (TS) is a **superset of JavaScript** created by Microsoft. It introduces **static typing**, making code safer and more maintainable.

## Key Characteristics

- Uses `.ts` files
- Compiles to JavaScript
- Has interfaces, enums, generics
- Offers excellent IDE support

# Pros & Cons of TypeScript

## ✓ Pros

- Type-safe
- Better for large apps
- Enhanced developer experience
- More maintainable

## ✗ Cons

- Requires compilation
- Harder to learn
- Extra setup needed

# TypeScript Basics: Examples

## Typed Function

```
function multiply(a: number, b: number): number {  
    return a * b;  
}
```

## Interface Example

```
interface Student {  
    name: string;  
    age: number;  
}  
const s1: Student = {  
    name: "Faiza",  
    age: 24  
};
```

# Build a To-Do App in TypeScript

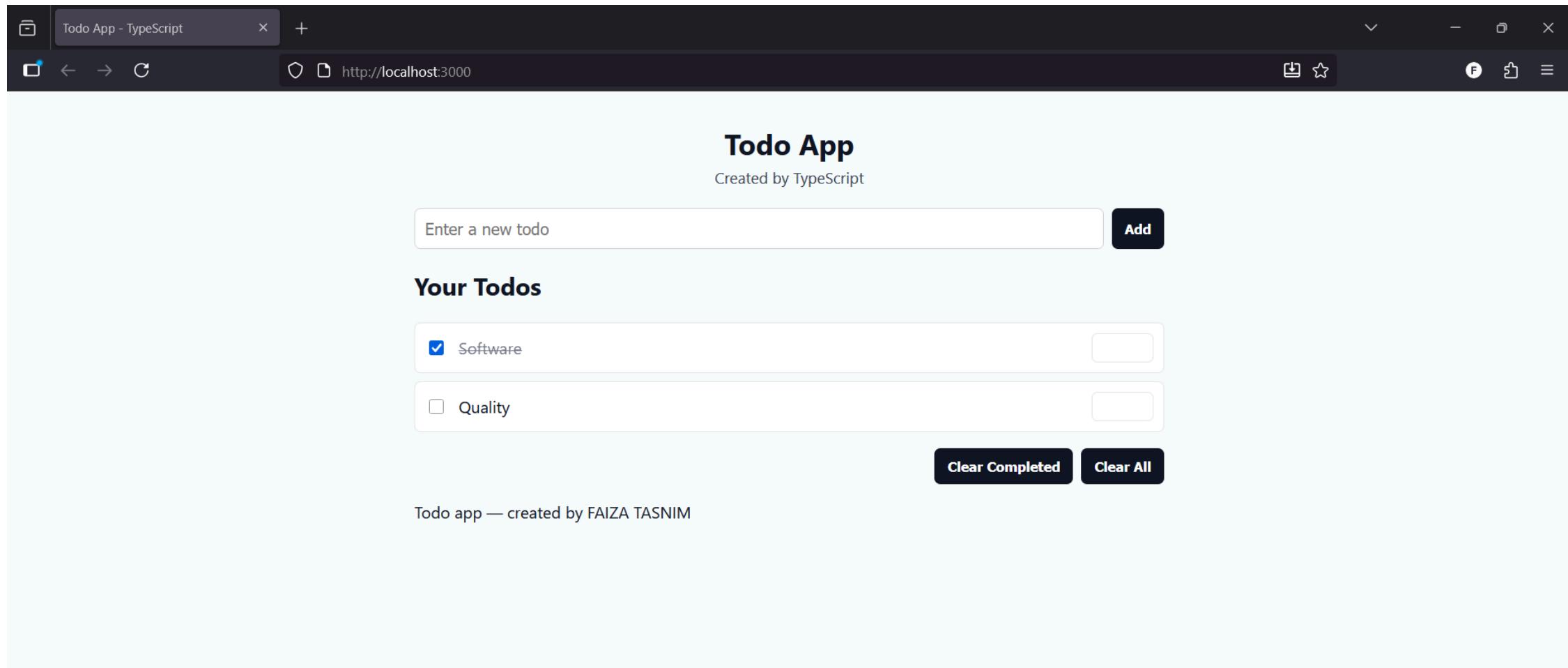
## HTML

```
<input id="taskInput" placeholder="Add a task" />
<button id="addBtn">Add</button>
<ul id="taskList"></ul>
```

# TypeScript

```
const input = document.getElementById("taskInput") as HTMLInputElement;
const btn = document.getElementById("addBtn") as HTMLButtonElement;
const list = document.getElementById("taskList") as HTMLULListElement;
function addTask(): void {
    const task: string = input.value.trim();
    if (!task) return;
    const li = document.createElement("li");
    li.textContent = task;
    list.appendChild(li);
    input.value = "";
}
btn.addEventListener("click", addTask);
```

# Example of Typescript ToDo



# JavaScript vs TypeScript — Detailed Comparison

## Typing System

- JavaScript → Dynamic typing
- TypeScript → Static typing

## Error Handling

- JavaScript → Errors at runtime
- TypeScript → Errors during development

## Learning Curve

- JavaScript → Easy for beginners
- TypeScript → Moderate difficulty

## Tooling

- JavaScript → Basic tools
- TypeScript → Advanced IDE features

## Compilation

- JavaScript → No compilation
- TypeScript → Must compile to JS

# Conclusion

- High-level languages simplify coding.
- JavaScript is ideal for fast and flexible development.
- TypeScript improves reliability with static types.
- Both can build apps like To-Do lists.

# React: Week-6

## What Is React?

- JavaScript library for building user interfaces
- Developed by Facebook (Meta)
- Based on components
- Uses Virtual DOM for fast rendering
- Enables interactive, dynamic web apps

# Why Use React?

## Advantages

- Component reusability
- Fast UI rendering (Virtual DOM)
- Large community & ecosystem
- Easy integration with APIs
- Suitable for scalable apps

## Disadvantages

- JSX learning curve
- Frequent library updates
- Not a full framework (needs external libraries)
- Initial setup can be complex

## Simple React ToDo App Example:

```
import { useState } from "react";

function App() {
  const [task, setTask] = useState("");
  const [list, setList] = useState([]);

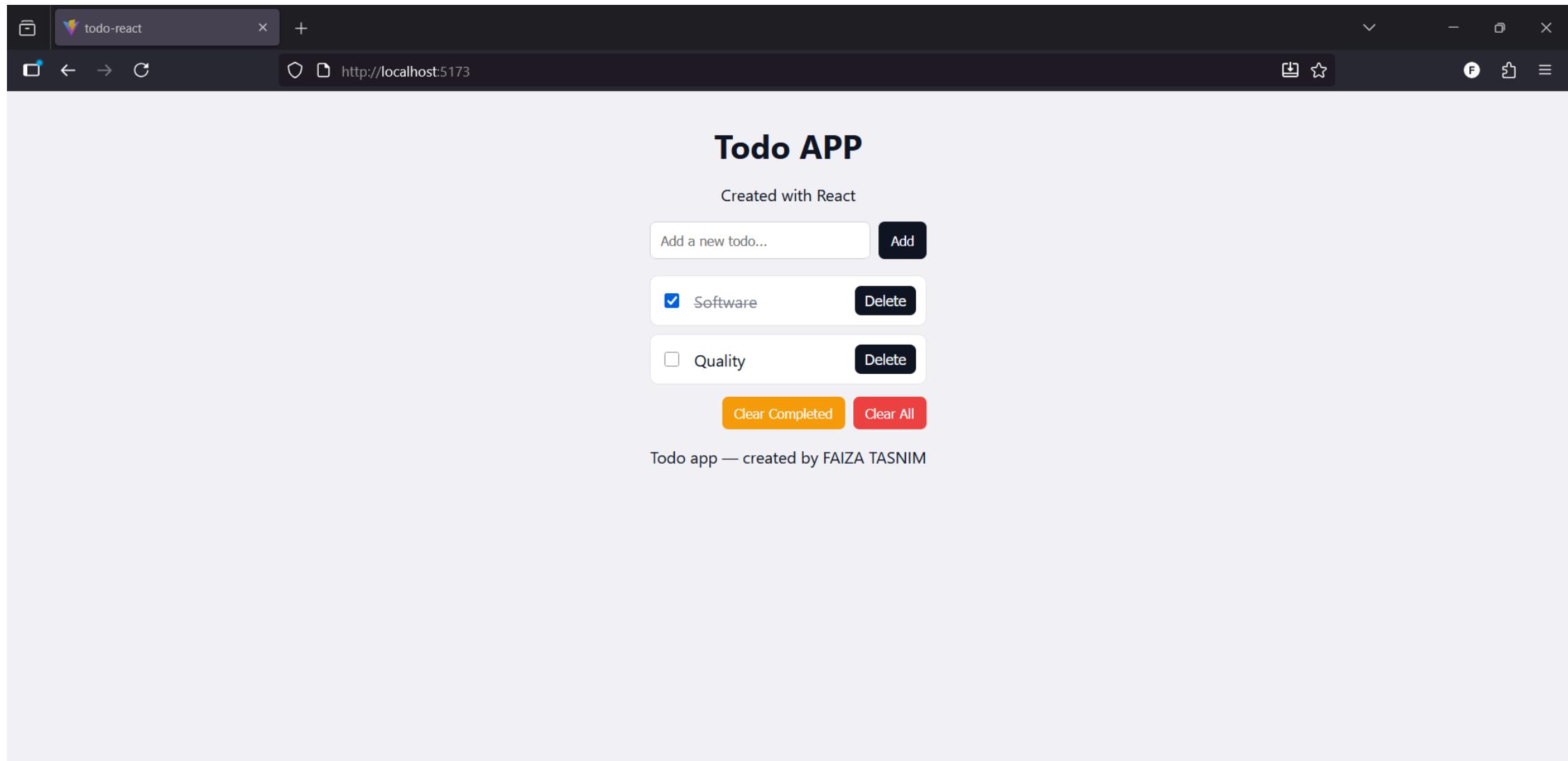
  const addTask = () => {
    if (task.trim() === "") return;
    setList([...list, task]);
    setTask("");
  };
}
```

```
return (
  <div>
    <h2>React ToDo App</h2>
    <input
      value={task}
      onChange={(e) => setTask(e.target.value)}
      placeholder="Enter task"
    />
    <button onClick={addTask}>Add</button>

    <ul>
      {list.map((item, i) => (
        <li key={i}>{item}</li>
      ))}
    </ul>
  </div>
);
}

export default App;
```

# Example of React ToDo



# Conclusion

- JavaScript → versatile programming language
- TypeScript → JS + static types, safer for large apps
- React → library for building dynamic web interfaces
- React + JS/TS → modern, interactive, scalable web apps

# GitHub, GitHub.io, Docker, and Hugo: Week-7

# 1. Introduction

This report provides an in-depth overview of four major tools in modern software development: **GitHub**, **GitHub.io**, **Docker**, and **Hugo**. The goal is to explain their functionality, use cases, and how they integrate into a developer's workflow.

## 2. GitHub

### Definition:

GitHub is a cloud-based platform that provides **version control** using the Git system. It allows developers to manage, track, and collaborate on code efficiently.

### Core Features:

- **Repositories:** Central storage for project files.
- **Branches:** Isolate development efforts for features, bug fixes, or experiments.
- **Pull Requests & Code Reviews:** Facilitate collaborative development and ensure code quality.
- **Issue Tracking:** Manage tasks, bugs, and feature requests.

## Use Cases:

- Collaborative software development.
- Open-source project hosting.
- Version control for both code and documentation.

## Working Mechanism:

Developers can **clone** a repository locally, make changes, and **push** them back. Others can **pull** these changes and merge them into their local copy or main branch.

## Example Diagram:

Developer A ---> GitHub Repo <--- Developer B  
(push, pull, merge)

### **3. GitHub.io**

#### **Definition:**

GitHub Pages (GitHub.io) is a free static website hosting service provided by GitHub. It allows developers to deploy websites directly from their repositories.

#### **Key Features:**

- Free hosting for public repositories.
- Customizable with themes and templates.
- Supports static content including HTML, CSS, and JavaScript.

## **Common Uses:**

- Personal portfolio websites.
- Project documentation.
- Blogs and static content platforms.

## **Workflow:**

```
Code on GitHub Repo -> GitHub Pages -> Live Website (username.github.io)
```

## 4. Docker

### Definition:

Docker is a platform for **containerization** that enables developers to package an application with its dependencies into a **container**. Containers are lightweight, portable, and ensure consistent execution across different environments.

### Core Concepts:

- **Container:** Encapsulates the application + dependencies.
- **Image:** Read-only template to create containers.
- **Docker Engine:** Runs and manages containers.

## Benefits:

- Environment consistency: runs the same on development, staging, and production.
- Lightweight and efficient compared to virtual machines.
- Simplifies deployment, scaling, and testing.

## Diagram:

```
[App + Dependencies] --> Docker Container --> Runs on Any OS
```

## Use Cases:

- Local development environment setup.
- Testing applications in isolated containers.
- Microservices architecture deployment.

## 5. Hugo

### Definition:

Hugo is a **static site generator** written in Go that transforms Markdown content into fully functional HTML websites.

### Core Features:

- **Fast Build Times:** Can generate thousands of pages in seconds.
- **Theming and Templates:** Supports reusable templates and layouts.
- **Markdown Support:** Easy content creation without HTML knowledge.

## **Benefits:**

- Ideal for blogs, portfolios, and documentation sites.
- Works seamlessly with GitHub Pages for hosting.
- Reduces server-side processing since the site is static.

## **Workflow:**

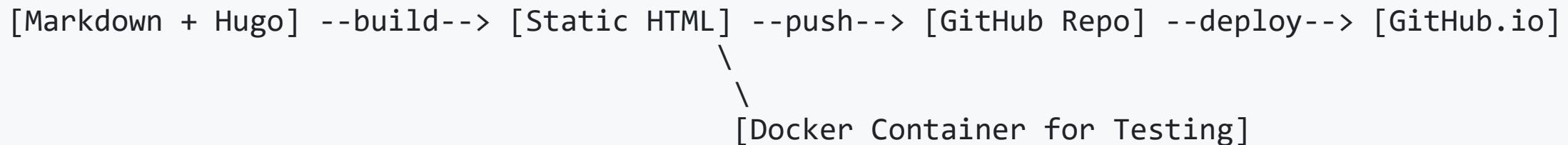
```
Markdown + Hugo Templates -> Hugo Build -> Static HTML Website -> Deploy (GitHub.io)
```

## 6. Integration: How They Work Together

### Step-by-Step Workflow:

- 1. Development:** Write content in Markdown and structure it with Hugo templates.
- 2. Version Control:** Store project and content on GitHub.
- 3. Build and Test:** Optionally use Docker to test Hugo site locally.
- 4. Deployment:** Push static HTML files to GitHub repository.
- 5. Hosting:** Enable GitHub Pages to make the site live at `username.github.io`.

### Diagram:

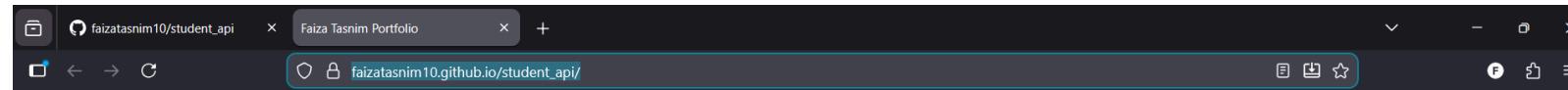


## 7. Advantages of Using These Tools Together

- **Fast Deployment:** Hugo + GitHub Pages allows almost instant publishing.
- **Collaboration:** GitHub supports team workflows, pull requests, and reviews.
- **Portability:** Docker ensures consistent environments for local testing.
- **Automation:** Integrate CI/CD pipelines for automatic builds and deployments.

# 8. Uses in my laravel project

## Example of Github.io Page



Faiza Tasnim Portfolio

[Laravel Student API & Docker](#)

**Laravel Student API & Docker**

Faiza Tasnim

Course: Software Quality Project: Student Management System

---

### What is a REST API?

A REST API allows different computer systems to communicate using HTTP methods such as **GET**, **POST**, **PUT**, and **DELETE**.

---

### Key Concepts

#### Client and Server

- Client sends request (browser, frontend app).
- Server processes request and returns JSON response.

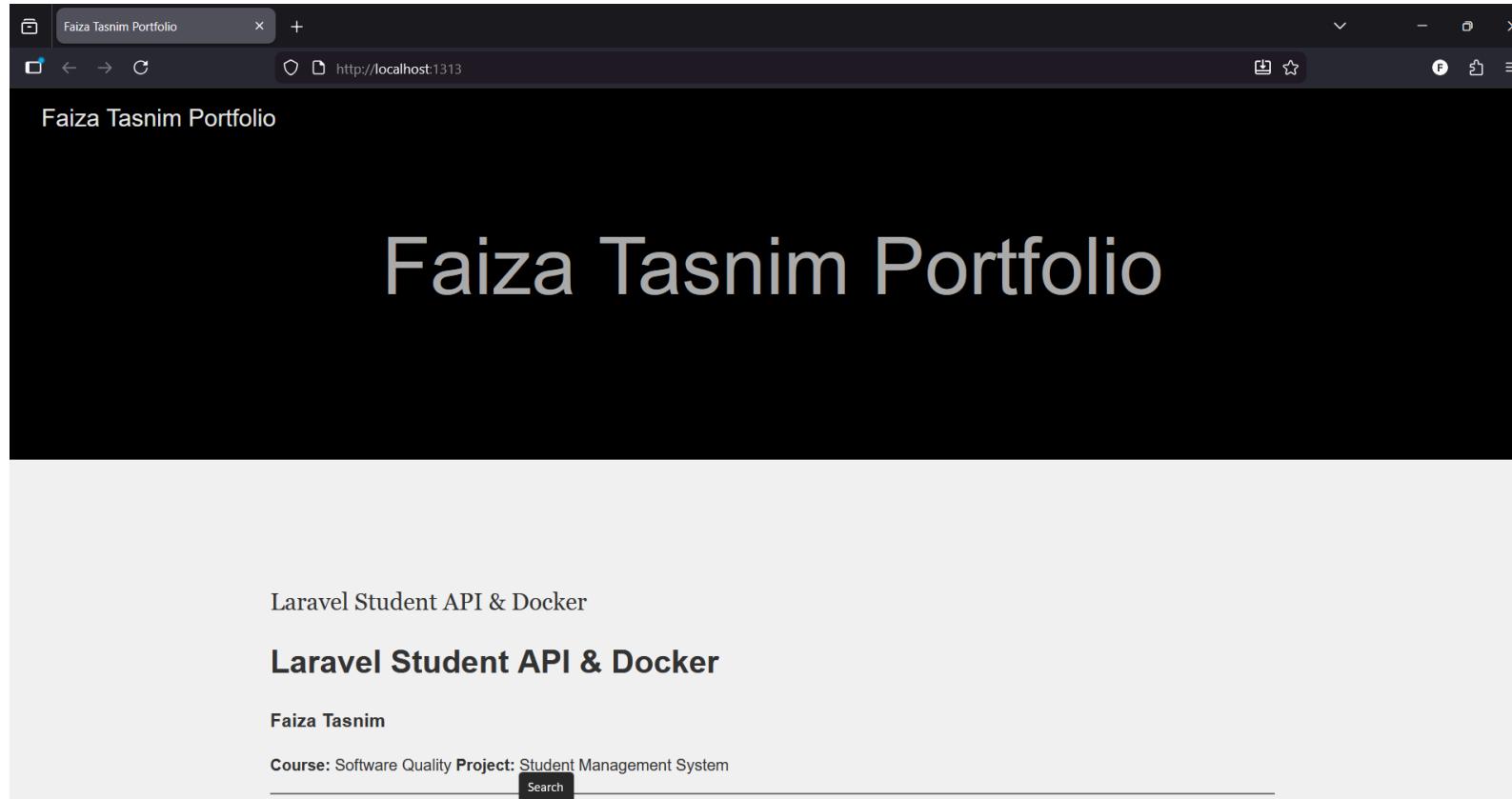
#### Stateless Communication

- Each request contains all necessary info.
- Server does not store session info.

---

### Resources

## Example of Hugo Page



# Example of working Docker

```
/usr/bin/bash --login -i D:\MS CSE\Software Quality\Upload HW\HW4_3\project-stage2-submission\code\student-api\setup.sh
=====
Starting Docker & Laravel (student-api)

[+] Building 11.0s (13/15)
=> [internal] load local bake definitions          0.0s
=> => reading from stdin 634B                      0.0s
=> [internal] load build definition from Dockerfile 0.1s
=> => transferring dockerfile: 739B                0.0s
=> [internal] load metadata for docker.io/library/composer:2 2.2s
=> [internal] load metadata for docker.io/library/php:8.2-apache 2.2s
=> [internal] load .dockerignore                   0.0s
=> => transferring context: 2B                  0.0s
=> [stage-0 1/8] FROM docker.io/library/php:8.2-apache@sha256:b8c8cd2be5690fcc2a6d0a44fee6c319052ee961148236f6a62a03f2b65e439f 0.1s
=> => resolve docker.io/library/php:8.2-apache@sha256:b8c8cd2be5690fcc2a6d0a44fee6c319052ee961148236f6a62a03f2b65e439f 0.1s
=> FROM docker.io/library/composer:2@sha256:8f3b7d97cd7436dae41d6af0dd6a53853779040ecac26e65544fc31746035705 0.1s
=> => resolve docker.io/library/composer:2@sha256:8f3b7d97cd7436dae41d6af0dd6a53853779040ecac26e65544fc31746035705 0.1s
=> [internal] load build context                 1.7s
=> => transferring context: 3.10MB            1.4s
=> CACHED [stage-0 2/8] WORKDIR /var/www/html      0.0s
=> CACHED [stage-0 3/8] RUN apt-get update && apt-get install -y libzip-dev zip unzip git curl    && docker-php-ext-install pdo_mysql zip 0.0s
=> CACHED [stage-0 4/8] RUN a2enmod rewrite       0.0s
=> CACHED [stage-0 5/8] COPY --from=composer:2 /usr/bin/composer /usr/bin/composer      0.0s
=> [stage-0 6/8] COPY .                          4.4s
=> [stage-0 7/8] RUN composer install --no-interaction --optimize-autoloader           2.1s
=> => # laravel/tinker ..... DONE
=> => # nesbot/carbon ..... DONE
=> => # nunomaduro/collision ..... DONE
=> => # nunomaduro/termwind ..... DONE
=> => # 81 packages you are using are looking for funding.
=> => # Use the 'composer fund' command to find out more!
```

## 9. Conclusion

- **GitHub:** Manages code and facilitates collaboration.
- **GitHub.io:** Provides free static website hosting.
- **Docker:** Enables consistent, containerized environments.
- **Hugo:** Generates static websites quickly from Markdown.