Surge Internship - Practical Test

1. OOP concepts

   There are 4 OOP concepts in java, and they are,
   Inheritance
   Abstraction
   Polymorphism
   Encapsulation

   Inheritance
   Inheritance is a mechanism in which one object acquire all the properties and behaviors of the parent object.
   As an example in Java :

```
class Animal{
 void eat(){
    System.out.println(“eating”);
}
}
class Dog extends Animal{
  void bark(){
     System.out.println(“barking”);
}
}
Class Single_inheritance{
  Public static void main(String[] args){
  Dog dog = new Dog();
  dog.Bark();
 dog.eat();
}
}
```

   The class Dog inherits its properties from the class Animal.

   Types of inheritance
   - Single inheritance
   - Multilevel inheritance
   - Hierarchical inheritance
   - Multiple inheritance
   - Hybrid inheritance

   Abstraction

The mechanism which represents the essential features without including implementation details. It is also called the process of generalization.

As an example,

```
 Abstract class MobilePhone{
     Public void calling();
     Public void sendSMS();
}
Public class Blackberry: MobilePhone
{
   Public void FMRadio();
   Public void Camera();
  Public void Recording();
}
```

In a mobile phone, there are different types of functionalities such as Camera, Recording, multimedia. But it is abstraction, because we only see those relevant information instead of their internal engineering.

Encapsulation

Simply binding the data with the operations on that data and it is the combination of abstraction and data hiding

As an example

```
Class Learn{
            private int a;
             private void show(){
                 console.writeline(a);
            }
}
```

Encapsulation is used to hide its member from outside class and interface. Using access modifiers like public,private, protected and default.

Polymorphism

Polymorphism is the ability to present the same interface for different underlying forms(data types). Polymorphism is basically when we can treat an object as a generic version of something, but when accessing to it, the code determines which exact type it is and calls the associated code.

```
Public abstract class Vehicle{
  Public abstract int Wheels;
}
```

```
Public class Car : Vehicle{

        Public override int Wheels(){
            Return 4;
        }
}
```

Polymorphism can be achieved using two ways,
They are,

- Method overriding – We use the method names in different classes ,that means parent class method is used in child class.
- Method overloading – Writing two or more methods in the same class using same method name but the passing parameters are different.

2. SDLC model

The process of planning, creating, testing, and deploying software is called Software Development Life Cycle or SDLC. Different tasks to be performed in each step of the software development process are explained well in SDLC. Different phases of SDLC are,

Planning and requirements analysis, design, development, testing, deployment, and maintenance.

Various SDLC models are the waterfall model, spiral model, V-shaped model, iterative model, big bang model, and agile model.

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an SRS (Software Requirement Specification) document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT-User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

3. Docker / Kubernetes and the use of it

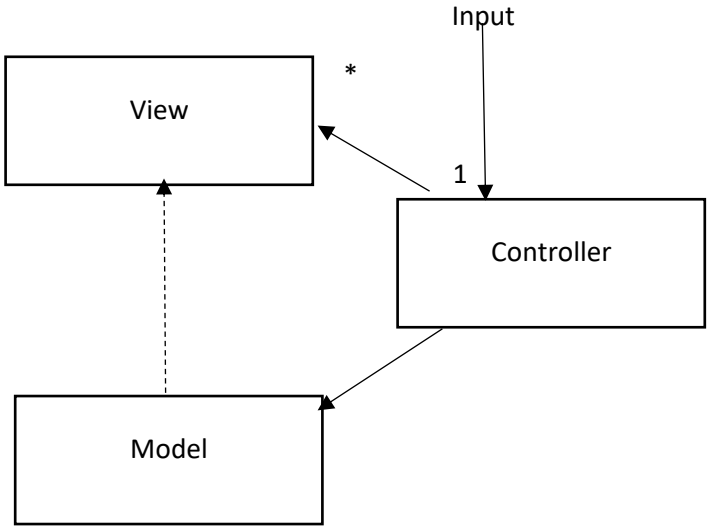
Today, both Docker and Kubernetes are leading container orchestration tools in the DevOps lifecycle. Docker uses a containerization platform for configuring, building, and distributing containers, while Kubernetes is an Ecosystem for managing a cluster of Docker containers.

- Docker provides a **containerization platform** which supports various operating systems such as Linux, Windows, and Mac. It allows us to easily build applications, package them with all required dependencies, and ship it to run on other machines. The advantage of using Docker is that it provides benefits for both developers as well as a system administrator.

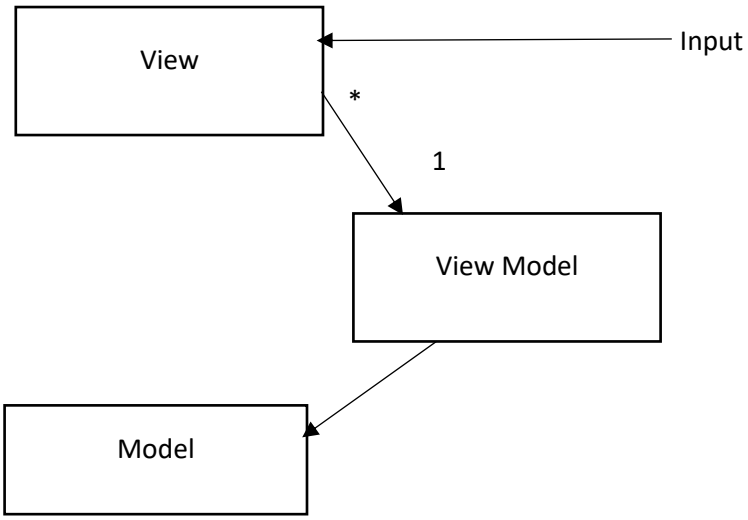
- Kubernets offers powerful, useful, and scalable tools for managing, deploying complicated containerized applications. The advantage of using Kubernetes is that it provides the best solution for scaling up the containers.Kubernetes includes various features such as runs everywhere, automated rollouts and rollback, storage orchestration, Batch execution, secret and configuration management, horizontal scaling, and offers additional services.

4. MVC and MVVM difference

MVC architecture



MVVM Architecture



MVC Architecture

The MVC framework is an architectural pattern that separates an applications into three main logical components Model, View, and Controller. Hence the abbreviation MVC. The full form MVC is Model View Controller. In this architecture, a component is built to handle specific development aspects of an application. MVC separates the business logic and presentation layer from each other. This architectural pattern mainly used for desktop graphical user.

MVVM Architecture

MVVM architecture facilitates a separation of development of the graphical user interface with the help of mark-up language or GUI code. The full form of MVVM is Model–View–ViewModel.The view model of MVVM is a value converter that means that it is view model's responsibility for exposing the data objects from the Model in such a way that objects are easily managed and presented.

Difference

- MVC framework is an architectural pattern that separates an applications into three main logical components Model, View, and Controller. On the other hand MVVM facilitates a separation of development of the graphical user interface with the help of mark-up language or GUI code
- In MVC, controller is the entry point to the Application, while in MVVM, the view is the entry point to the Application.
- MVC Model component can be tested separately from the user, while MVVM is easy for separate unit testing, and code is event-driven.
- MVC architecture has "one to many" relationships between Controller & View while in MVVCM architecture, "one to many" relationships between View & View Model.

5. Data Flow Diagram

A data flow diagram (DFD) maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination. Data flowcharts can range from simple, even hand-drawn process overviews, to in-depth, multi-level DFDs that dig progressively deeper into how the data is handled. They can be used to analyze an existing system or model a new one.

DFD has often been used due to the following reasons:

- Logical information flow of the system
- Determination of physical system construction requirements
- Simplicity of notation
- Establishment of manual and automated systems requirements

Practical

Commands

1. npx create-react-app@latest frontend
2. a. validation – Frontend
   useEffect, useState hooks
   validation – Backend
   mongoose
   const userSchema = mongoose.Schema
   userSchema.pre('save', async function()

   b.backend -usercontrollers.js

```
 const registerUser = asyncHandler(async(req, res) => {
    const {name, email, password, username} = req.body;

    const userExists = await User.findOne({email});
    const userExists1 = await User.findOne({username});

    if(userExists || userExists1){
        res.status(400);
        throw new Error('User Already Exists');
    }

    const user = await User.create({
        name,
        email,
        password,
        username,
    });
    if(user){
        res.status(201).json({
            _id: user._id,
            name: user.name,
            email: user.email,
            isAdmin: user.isAdmin,
            username: user.username,
            token: generateToken(user._id),
        });
    }else{
        res.status(400);
        throw new Error("Error occured!");
    }
});

const authUser = asyncHandler(async(req, res) => {
    const { email, password} = req.body;
```

```
    const user = await User.findOne({email});

    if(user && (await user.matchPassword(password))){
        res.json({
            _id: user._id,
            name: user.name,
            email: user.email,
            isAdmin: user.isAdmin,
            username: user.username,
            token: generateToken(user._id),
        })
    }else{
        res.status(400);
        throw new Error("Invalid Email or Password!");
    }

});
const updateUserProfile = asyncHandler(async (req, res) => {
    const user = await User.findById(req.user._id);

    if (user) {
      user.name = req.body.name || user.name;
      user.email = req.body.email || user.email;
      user.username = req.body.username || user.username;
      if (req.body.password) {
        user.password = req.body.password;
      }

      const updatedUser = await user.save();

      res.json({
        _id: updatedUser._id,
        name: updatedUser.name,
        email: updatedUser.email,
        username: updatedUser.username,
        isAdmin: updatedUser.isAdmin,
        token: generateToken(updatedUser._id),
      });
    } else {
      res.status(404);
      throw new Error("User Not Found");
    }
  });
```

```
module.exports = {registerUser, authUser, updateUserProfile};
```

userRoutes.js

```
const express = require('express');
const {registerUser, authUser, updateUserProfile} =
require('../controllers/userControllers');
const { protect } = require('../middlewares/authMiddleware');

const router = express.Router()

router.route('/').post(registerUser);
router.route('/login').post(authUser);
router.route('/profile').post(protect, updateUserProfile);

module.exports = router;
```

Frontend –

userConstants.js

```
export const USER_LOGIN_REQUEST = "USER_LOGIN_REQUEST";
export const USER_LOGIN_SUCCESS = "USER_LOGIN_SUCCESS";
export const USER_LOGIN_FAIL = "USER_LOGIN_FAIL";
export const USER_LOGOUT = "USER_LOGOUT";

export const USER_REGISTER_REQUEST = "USER_REGISTER_REQUEST";
export const USER_REGISTER_SUCCESS = "USER_REGISTER_SUCCESS";
export const USER_REGISTER_FAIL = "USER_REGISTER_FAIL";

export const USER_UPDATE_REQUEST = "USER_UPDATE_REQUEST";
export const USER_UPDATE_SUCCESS = "USER_UPDATE_SUCCESS";
export const USER_UPDATE_FAIL = "USER_UPDATE_FAIL";
```

userReducers.js

```
import { USER_LOGIN_FAIL, USER_LOGIN_REQUEST, USER_LOGIN_SUCCESS, USER_LOGOUT,
USER_REGISTER_FAIL, USER_REGISTER_REQUEST, USER_REGISTER_SUCCESS,
USER_UPDATE_FAIL, USER_UPDATE_REQUEST, USER_UPDATE_SUCCESS } from
"../constants/userConstants"
export const userLoginReducer= (state={}, action) => {
    switch (action.type) {
        case USER_LOGIN_REQUEST:
            return {loading: true};
        case USER_LOGIN_SUCCESS:
```

```
            return {loading: false, userInfo: action.payload};
        case USER_LOGIN_FAIL:
            return {loading: false, error: action.payload};
        case USER_LOGOUT:
            return {};
        default:
            return state;
    }
}
export const userRegisterReducer = (state = {}, action) => {
    switch (action.type) {
      case USER_REGISTER_REQUEST:
        return { loading: true };
      case USER_REGISTER_SUCCESS:
        return { loading: false, userInfo: action.payload };
      case USER_REGISTER_FAIL:
        return { loading: false, error: action.payload };
      default:
        return state;
    }
  };

  export const userUpdateReducer = (state = {}, action) => {
    switch (action.type) {
      case USER_UPDATE_REQUEST:
        return { loading: true };
      case USER_UPDATE_SUCCESS:
        return { loading: false, userInfo: action.payload, success: true };
      case USER_UPDATE_FAIL:
        return { loading: false, error: action.payload, success: false };
      default:
        return state;
    }
  };
```

userActions.js

```
import axios from 'axios';
import { USER_LOGIN_FAIL, USER_LOGIN_REQUEST, USER_LOGIN_SUCCESS, USER_LOGOUT,
USER_REGISTER_FAIL, USER_REGISTER_REQUEST, USER_REGISTER_SUCCESS,
USER_UPDATE_FAIL, USER_UPDATE_REQUEST, USER_UPDATE_SUCCESS } from
'../constants/userConstants';
export const login = (email, password) => async (dispatch) => {
    try {
      dispatch({ type: USER_LOGIN_REQUEST });
```

```
      const config = {
        headers: {
          "Content-type": "application/json",
        },
      };

      const { data } = await axios.post(
        "/api/users/login",
        { email, password },
        config
      );

      dispatch({ type: USER_LOGIN_SUCCESS, payload: data });

      localStorage.setItem("userInfo", JSON.stringify(data));
    } catch (error) {
      dispatch({
        type: USER_LOGIN_FAIL,
        payload:
          error.response && error.response.data.message
            ? error.response.data.message
            : error.message,
      });
    }
  };

  export const logout = () => async (dispatch) => {
    localStorage.removeItem("userInfo");
    dispatch({ type: USER_LOGOUT });
  };
  export const register = (name, email, password, username) => async (dispatch)
=> {
    try {
      dispatch({ type: USER_REGISTER_REQUEST });

      const config = {
        headers: {
          "Content-type": "application/json",
        },
      };

      const { data } = await axios.post(
        "/api/users",
        { name, username, email, password },
```

```
      config
    );

    dispatch({ type: USER_REGISTER_SUCCESS, payload: data });

    dispatch({ type: USER_LOGIN_SUCCESS, payload: data });

    localStorage.setItem("userInfo", JSON.stringify(data));
  } catch (error) {
    dispatch({
      type: USER_REGISTER_FAIL,
      payload:
        error.response && error.response.data.message
          ? error.response.data.message
          : error.message,
    });
  }
};
export const updateProfile = (user) => async (dispatch, getState) => {
  try {
    dispatch({ type: USER_UPDATE_REQUEST });

    const {
      userLogin: { userInfo },
    } = getState();

    const config = {
      headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${userInfo.token}`,
      },
    };

    const { data } = await axios.post("/api/users/profile", user, config);

    dispatch({ type: USER_UPDATE_SUCCESS, payload: data });

    dispatch({ type: USER_LOGIN_SUCCESS, payload: data });

    localStorage.setItem("userInfo", JSON.stringify(data));
  } catch (error) {
    dispatch({
      type: USER_UPDATE_FAIL,
      payload:
        error.response && error.response.data.message
```

```
            ? error.response.data.message
            : error.message,
      });
    }
  };
```

c. error handling

Frontend-

loginScreen.js

```
<div className="loginContainer">
                {error && <ErrorMessage variant="danger">{error}</ErrorMessage>}
```

RegisterScreen.js

```
 const submitHandler = async (e) => {
    e.preventDefault();
    if (password !== confirmpassword) {
      setMessage("Passwords do not match");
    } else dispatch(register(name, email, password, username));

  }
```

userReducers.js

```
        case USER_LOGIN_FAIL:
            return {loading: false, error: action.payload};
```

```
      case USER_REGISTER_FAIL:
        return { loading: false, error: action.payload };
```

```
      case USER_UPDATE_FAIL:
        return { loading: false, error: action.payload, success: false };
```

userActions.js

```
catch (error) {
      dispatch({
        type: USER_LOGIN_FAIL,
        payload:
          error.response && error.response.data.message
            ? error.response.data.message
            : error.message,
      });
```

```
    }
```

```
catch (error) {
      dispatch({
        type: USER_REGISTER_FAIL,
        payload:
          error.response && error.response.data.message
            ? error.response.data.message
            : error.message,
      });
    }
```

```
catch (error) {
      dispatch({
        type: USER_UPDATE_FAIL,
        payload:
          error.response && error.response.data.message
            ? error.response.data.message
            : error.message,
      });
    }
```

e.

apis – express.js

```
const asyncHandler = require('express-async-handler')
const User = require('../models/userModel');
const generateToken = require('../utils/generateToken');
const registerUser = asyncHandler(async(req, res) => {
    const {name, email, password, username} = req.body;

    const userExists = await User.findOne({email});
    const userExists1 = await User.findOne({username});

    if(userExists || userExists1){
        res.status(400);
        throw new Error('User Already Exists');
    }

    const user = await User.create({
        name,
        email,
        password,
```

```
        username,
    });
    if(user){
        res.status(201).json({
            _id: user._id,
            name: user.name,
            email: user.email,
            isAdmin: user.isAdmin,
            username: user.username,
            token: generateToken(user._id),
        });
    }else{
        res.status(400);
        throw new Error("Error occured!");
    }
});

const authUser = asyncHandler(async(req, res) => {
    const { email, password} = req.body;

    const user = await User.findOne({email});

    if(user && (await user.matchPassword(password))){
        res.json({
            _id: user._id,
            name: user.name,
            email: user.email,
            isAdmin: user.isAdmin,
            username: user.username,
            token: generateToken(user._id),
        })
    }else{
        res.status(400);
        throw new Error("Invalid Email or Password!");
    }

});
const updateUserProfile = asyncHandler(async (req, res) => {
    const user = await User.findById(req.user._id);

    if (user) {
      user.name = req.body.name || user.name;
      user.email = req.body.email || user.email;
      user.username = req.body.username || user.username;
      if (req.body.password) {
```

```
        user.password = req.body.password;
      }

      const updatedUser = await user.save();

      res.json({
        _id: updatedUser._id,
        name: updatedUser.name,
        email: updatedUser.email,
        username: updatedUser.username,
        isAdmin: updatedUser.isAdmin,
        token: generateToken(updatedUser._id),
      });
    } else {
      res.status(404);
      throw new Error("User Not Found");
    }
  });

module.exports = {registerUser, authUser, updateUserProfile};
```