Interview Preparation

Personality

1. What can you tell me about yourself?
2. What was your biggest success in software development?
3. What do you think is your biggest weakness?

Java Programming

1. What frameworks have you been working with in a professional environment?
    a. I’ve been working with Spring Boot for 4 years now.
2. What are the concepts of OOP?
    a. OOP concepts are: Encapsulation, Inheritance, Abstraction and Polymorphism
3. Explain Dependency Injection.
    a. **Dependency Injection (DI)** is a design pattern that means: Instead of a class creating its own dependencies, someone else gives them to it.
    b. Benefits:
        i. Easier to test
        ii. Easier to change
        iii. **Less tightly connected** (loosely coupled)
    c. **Dependency Injection is managed by Spring’s IoC (Inversion of Control) container**.
        i. **IoC** means **Spring controls the creation and wiring of objects**, not you.

ii. You just **declare what you need**, and Spring **injects it for you**.

iii. Spring's **IoC container** scans and creates a **bean** for that class.

iv. Spring IoC is the engine behind Dependency Injection.

v. It **creates**, **manages**, and **connects** all the parts of your application.

d. Simple Example

i. Imagine a car that needs an engine.

1. Without DI: the car **creates its own engine**.
2. With DI: the **engine is passed in** when the car is built.

ii. So, the car **depends** on an engine, and it **gets injected** with one from the outside.

e. In Spring Boot:

i. Spring handles dependency injection automatically. There are three different ways of injecting dependencies:

1. Constructor Injection - Recommended, it ensures immutability
2. Field Injection - Easier to write, but harder to test and less visible — **not recommended** for production code.
3. Setter Injection - Useful when the dependency is **optional** or when you want to **change it later**.

f. Summary:

i. **Dependency Injection** means a class **doesn't create what it needs**, it **receives it from outside** — making your code cleaner, testable, and easier to maintain.

4. What is polymorphism?

a. Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass or interface. It enables objects to exhibit different behaviors based on their specific types or the context in which they are used.

   b. There are two main types of polymorphism in Java: ***compile-time (or static) polymorphism and runtime (or dynamic) polymorphism.***
   c. Compile-time polymorphism is achieved through method overloading and method overriding.
   d. Runtime polymorphism means creating a new object by referencing its superclass, e.g:
      Animal animal = new Dog();
5. What is the difference between a class and an interface?
   a. A **class** is like a **blueprint** that can be used to **create real objects**. It can have:
      i. Code (methods with logic)
      ii. Data (fields or variables)
      iii. You can **create objects** from it using `new`.
   b. An **interface** is like a **contract** that says: "Any class that uses me must **implement these methods**.".
      It does
      not have full code, just **method names** (unless you're using default methods in newer Java versions).
      You
      cannot create an object directly from an interface.
   c. Summary
      i. A **class** is a full definition.
      ii. An **interface** is a set of rules that a class agrees to follow.
6. What is cohesion in OOP?
   a. **Cohesion** means how well the parts of a class **belong together**.
      i. If a class **does one thing and does it well**, it has **high cohesion**.
      ii. If a class tries to do **many unrelated things**, it has **low cohesion** — and that usually makes the code harder to understand and maintain.
   b. Example:

        i. A class called `InvoicePrinter` that only knows how to print invoices → **high cohesion**.
        ii. A class called `InvoiceHelper` that prints invoices, sends emails, and calculates taxes → **low cohesion**.
        iii. High cohesion = clean, focused, and easier to reuse.

7. What is a Hashtable?
    a. A **Hashtable** is a data structure in Java that stores **key-value pairs**, just like a `HashMap`. But the main difference is: It's **thread-safe** — all its methods are **synchronized**, so only one thread can use it at a time.
    b. Key Points
        i. It doesn't allow **null keys or null values**
        ii. It was part of **early Java (legacy class)** before `HashMap` and `ConcurrentHashMap` came along
        iii. It's slower than newer options because it **locks the whole table** for every operation
    c. Summary:
        i. A `Hashtable` is like a `HashMap`, but **older and slower** because it uses **full synchronization** to make it thread-safe.

8. What are the differences between Java 8 and Java 11?
    a. Both Java 8 and Java 11 are LTS (Long Term Supported)
    b. Java 11 has a new feature called ***Modular System***, Java 8 doesn't.
    c. Java 11 has a better overall performance compared to Java 8.
    d. Java 11 supports the "***var***" type of variable as a local variable, Java 8 does not.
    e. Java 11 offers a new API for handling HTTP requests.
    f. Java 11 offers new updated APIs for Files and Strings.
    g. Java 11 has quite some security enhancements.

9. What is Lambda Expression?

   a. Lambda expressions are a significant feature introduced in Java 8 that enable functional programming in Java. A lambda expression is a concise way to represent an anonymous function, which can be treated as a method argument, returned as a value, or stored in a variable.

10. What is SQL Injection, how does it work and how can you prevent it?
    a. **SQL Injection** is a security vulnerability where an attacker manipulates **user input** to execute **malicious SQL statements** on your database.
    b. When raw user input is **concatenated directly** into an SQL query without proper validation or escaping, an attacker can inject their own SQL.
    c. Here is how you can prevent SQL Injection:
        i. Use Prepared Statements / Parameterized Queries
        ii. **Use ORM frameworks** like **Spring Data JPA** or **Hibernate**, which use parameterized queries by default.
        iii. **Validate and sanitize inputs** (especially for dynamic queries).
        iv. **Limit database privileges** for the application user (e.g., no DROP or DELETE access unless needed).
        v. **Avoid building queries by string concatenation**—always bind parameters.
    d. SQL Injection happens when user input is unsafely included in SQL queries. Use **prepared statements**, **input validation**, and **ORMs** to stay secure.

11. How to make an object Thread-safe?
    a. To make an object **thread-safe** in Java, you need to ensure that it behaves **correctly when accessed by multiple threads at the same time**. Here are common techniques to achieve thread safety:
        i. Use Synchronized Methods or Blocks
        ii. Use Volatile (for visibility only)
        iii. Use Atomic Classes
        iv. Use Thread-safe Collections

1. ConcurrentHashMap
2. CopyOnWriteArrayList
3. BlockingQueue
4. Avoid non-thread-safe collections like `ArrayList`, `HashMap` in concurrent code.

v. Use Immutability

1. Make your object immutable: no setters, all fields `final`.
2. Immutable objects are naturally thread-safe.

b. Summary:

i. Use **synchronization** to control access

ii. Use **atomic variables** for simple concurrency

iii. Prefer **thread-safe classes**

iv. Make objects **immutable** when possible

12. How to identify a Memory Leak?

a. Identifying a **memory leak** in Java means detecting when memory that is **no longer needed** is **not being released**, often due to lingering references. Here's how you can spot one:

i. Observe JVM Memory Usage Over Time

1. Use **JVisualVM**, **JConsole**, or **Java Mission Control**
2. Look for a **steady increase** in heap usage that **does not drop** after garbage collection

ii. Analyze Heap Dumps

1. Generate a heap dump when memory is high
2. Open with tools like
   a. Eclipse Memory Analyzer Tool (MAT)
   b. VisualVM

iii. Use Profiling Tools

            1. Tools like **YourKit**, **JProfiler**, or **Eclipse MAT**

    b. Common Causes

        i. Static fields referencing heavy objects

        ii. Unclosed resources (e.g., streams, DB connections)

        iii. Caching without eviction

        iv. Forgotten event listeners or observers

        v. Improper use of inner classes

    c. Summary

        i. You can identify memory leaks by **monitoring heap usage**, **analyzing heap dumps**, and using **profilers** to locate **unreleased object references**. Regular testing and good practices (e.g., using `try-with-resources`, avoiding unnecessary statics) help prevent them.

13. How does Java Garbage Collector work?

    a. The **Java Garbage Collector (GC)** is a part of the Java Virtual Machine (JVM) that automatically manages **memory** by identifying and reclaiming objects that are **no longer reachable** by any part of your program.

    b. How It Works – In Simple Steps:

        i. Memory Allocation

            1. When you create a new object (`new`), it's stored in the **heap memory**.

        ii. Object Reachability

            1. The GC checks which objects are **still referenced** by your application.

            2. If an object can't be reached (no variable points to it), it's considered **garbage**.

        iii. Mark-and-Sweep

            1. The GC **marks** all live (reachable) objects.

2. Then it **sweeps** away the unmarked (unreachable) ones to free memory.

iv. Generational Collection (The heap is typically divided into)

1. **Young Generation**: where new objects are allocated.
2. **Old Generation (Tenured)**: where long-living objects are moved.
3. **Eden / Survivor Spaces**: inside the Young Gen for efficient GC.
4. GC collects **young gen** more frequently (minor GC), and **old gen** less often (major GC or full GC).

c. Types of Garbage Collectors in Java

i. Serial GC - Single-threaded, good for small apps

ii. Parallel GC - Multi-threaded, for throughput

iii. CMS (deprecated) - Concurrent, low pause

iv. **G1 GC** (default in many JVMs) - Balances pause time and throughput

v. ZGC / Shenandoah - Very low pause, scalable (Java 11+ and 12+)

d. As a developer, you don't manage memory manually, but:

i. Avoid unnecessary object creation

ii. Close resources (`try-with-resources`)

iii. Watch for memory leaks (objects kept alive too long)

e. Summary:

i. The **Java GC** tracks object usage, removes unreachable ones from memory, and helps prevent memory leaks and crashes — all **automatically**, so you can focus on writing code, not managing memory.

14. What does the keyword `synchronize` do?

a. The `synchronized` keyword in Java is used to **control access to a block of code or method** by allowing **only one thread at a time** to execute it.

b. It ensures **mutual exclusion**, meaning:
Only

one thread can access the **synchronized block or method** at any given time for the same object (or class, if static).

c. It prevents **race conditions** and ensures **thread safety** when multiple threads interact with shared data.

d. There are different types of synchronization:

   i. Synchronized Method - Locks on the **object's instance** (`this`)

   ii. Synchronized Block - Gives finer control by locking only part of the code. You can also lock on a specific object

   iii. Static Synchronized Method - Locks on the **class object**, not the instance

e. Summary: `synchronized` ensures that only one thread at a time can access critical code, helping prevent concurrency issues like race conditions in **multi-threaded Java applications**.

15. What are some collections that are thread-safe and how do they work?

   a. `ConcurrentHashMap` is one collection that is thread-safe. `ConcurrentHashMap` doesn't lock the **entire map** when a thread wants to write. Instead, it locks **only a small portion** — specifically, the **bucket or segment** related to the key. This lock is usually referred to as **Smart Locking.**

   b. So, how does it know which thread to let write? When two threads want to write to the map at the same time:

      i. **Each key is hashed**, and based on that, the map knows **which part (or bucket)** of the map to look at.

      ii. If the two keys belong to **different buckets**, both threads can write **at the same time**, because their parts are separate and don't interfere.

      iii. If they try to write to the **same bucket**, then one thread **gets the lock first**, and the other thread must **wait** until the lock is released.

      iv. This approach is called **fine-grained locking**.

   c. Why is this smart? Because it allows:

      i. **Multiple reads** to happen at the same time (no lock for reading)

        ii. **Multiple writes** as long as they are on **different keys/buckets**
        iii. Less waiting and better performance compared to locking the whole map
    d. Summary:
        i. `ConcurrentHashMap` is smart because it:
            1. Divides the map into many smaller parts internally
            2. Lets threads work in **parallel** if they access **different parts**
            3. Uses **locks only where needed**, making it fast and safe

Spring Boot

1. What is Spring Boot and why do you think it is so popular?
    a. **Spring Boot** is a framework built on top of the Spring Framework that simplifies the development of Java-based web and enterprise applications by offering:
        i. **Auto-configuration** (less boilerplate)
        ii. **Embedded servers** (like Tomcat, no need to deploy WAR files)
        iii. **Production-ready features** (like metrics and health checks via Actuator)
        iv. Convention over configuration
    b. Why it's popular:
        i. Speeds up development
        ii. Easy to get started with minimal setup
        iii. Integrates well with databases, security, messaging, and cloud services
        iv. Backed by the mature and powerful Spring ecosystem

2. What does the @Component annotation do in Spring Boot?

    a. The `@Component` annotation in Spring Boot marks a **Java class as a Spring-managed bean**. When Spring scans the application (during component scanning), it **automatically detects** classes annotated with `@Component` and registers them in the application context, so they can be **injected using `@Autowired`**.

    b. Annotations like `@RestController`, `@Service`, `@Repository`, and `@Controller` are all **specialized versions of `@Component`**, and they **inherit its behavior**.

3. What is JPA and what is it used for?

    a. **JPA (Java Persistence API)** is a **Java specification** used to manage **relational data** in Java applications. JPA will take care of **saving, updating, and fetching** this object from the database. JPA is used for:

        i. Mapping **Java objects to database tables** (Object-Relational Mapping – ORM)

        ii. Performing **CRUD operations** without writing SQL

        iii. Managing **entity relationships** (e.g., one-to-many, many-to-one)

        iv. Handling **transactions**, **caching**, and **querying**

        v. **Hibernate** is the most popular **implementation** of JPA.

4. What is the difference between Hibernate and JPA?

    a. **JPA** is a **specification** (a set of rules and interfaces).

    b. **Hibernate** is a **concrete implementation** of that specification.

    c. In simple terms:

        i. JPA: Specification/API - Hibernate: Implementation (framework)

        ii. JPA: Interfaces and annotations - Hibernate: Concrete code that runs your queries

        iii. JPA: Requires a provider to work (e.g., Hibernate) - Hibernate: Is itself a JPA provider

        iv. You write code using **JPA annotations**, and **Hibernate** handles the actual database interaction behind the scenes.

5. What are generics and what are they used for?
    a. **Generics** in Java allow you to write **classes, interfaces, and methods** that work with **any type** while providing **type safety** at compile time.
    b. Generics make Java code **cleaner**, **safer**, and **easier to maintain**.
    c. Generics are used for:
        i. **Reusability**: Write code once, use with any data type.
        ii. **Type Safety**: Catch type errors at compile time.
        iii. **Avoid Casting**: No need for manual casting.
    d. List of commonly used **generic types** and **generic concepts** in Java:
        i. List<T>
        ii. Set<T>
        iii. Map<K, V>
        iv. Queue<T>
        v. Optional<T>
    e. Wildcard Generics:
        i. <?> - Unknown type
        ii. <? extends T> - Any type that is a subtype of T
        iii. <? super T> - Any type that is a supertype of T
6. What is the difference between Spring and Java Enterprise Edition(JEE)?
    a. **Spring** is a lightweight, flexible, and developer-friendly framework that simplifies Java application development using built-in dependency injection, modular design, and embedded servers like Tomcat—especially through Spring Boot.
    b. In contrast, **Java EE** (now **Jakarta EE**) is a standardized set of specifications for building enterprise applications, traditionally relying on external application servers and offering less flexibility, but valued for its vendor neutrality and official support in large-scale enterprise environments.

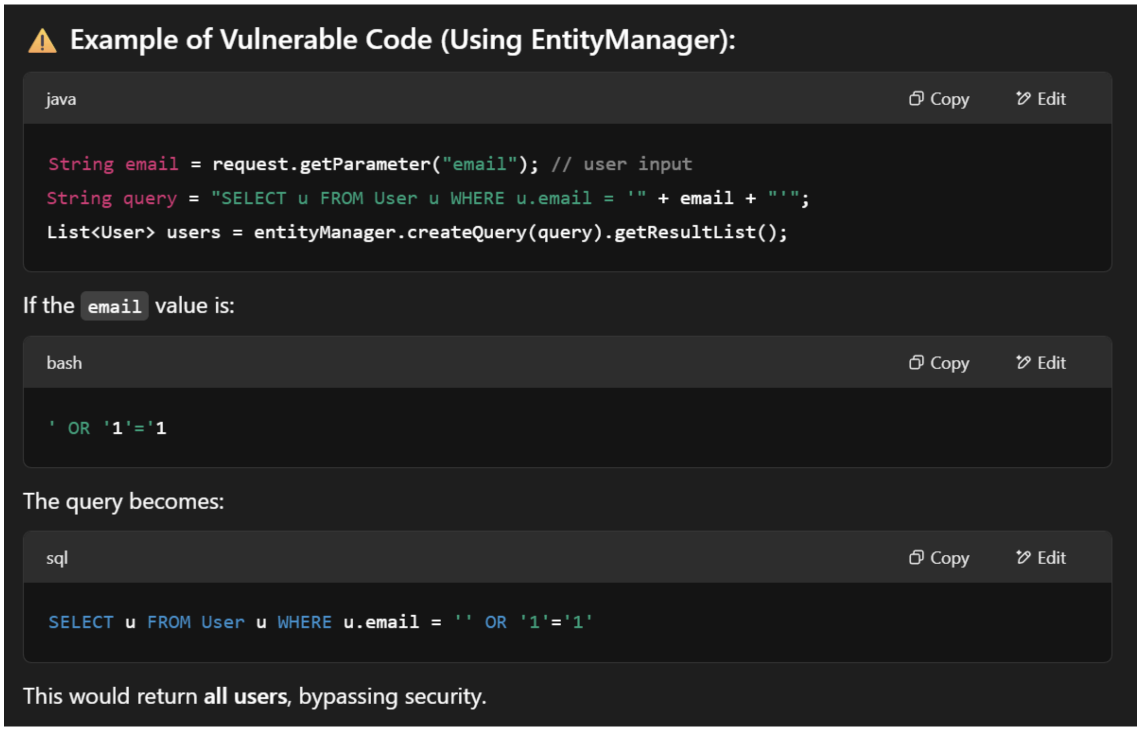
c. **Summary**: **Spring is not built on top of Java EE (now Jakarta EE)** — it is an **alternative** to it.
   However,
   Spring can use some Java EE technologies under the hood, such as **Servlets**, **JPA**, and **JTA**, because these are part of the standard Java ecosystem. But Spring provides **its own implementations and abstractions**, allowing you to build enterprise applications **without relying on a full Java EE server**.
   Spring and Java EE are
   independent approaches to building Java enterprise applications, but they can **interoperate** where needed.

7. Why is the implementation of ***hashCode*** and ***equals*** so important for Java Collections?
   a. The implementation of `hashCode()` and `equals()` is crucial for **correct behavior** when using Java collections like `HashMap`, `HashSet`, and `Hashtable`.
   b. **`hashCode()`** **determines the bucket** where an object will be placed in a hash-based collection (like `HashMap` or `HashSet`).
   c. **`equals()`** **determines equality** between two objects. It's used to check if a new object is considered *equal* to an existing one already in the collection.
   d. What Happens If They're Not Implemented Correctly?
      i. Objects may **not be found** even if they exist.
      ii. You might end up with **duplicate entries** in a `Set`.
      iii. You could violate the **contract** between `equals()` and `hashCode()`, leading to unpredictable behavior.
   e. Rule of Thumb:
      i. If you override `equals()`, **always override** **`hashCode()`**.
      ii. If `a.equals(b)` is `true`, then `a.hashCode() == b.hashCode()` **must also be true**.
8. How does SQL Injection happen using EntityManager?
   a. SQL Injection can happen when using `EntityManager` if **user input is directly concatenated into a query string**, especially in **native queries** or **JPQL**

without proper parameter binding.

b. SQL Injection in `EntityManager` occurs when you **manually build query strings** with user input. To avoid it, **always use named or positional parameters** instead of string concatenation.

c. Injection Example:



⚠️ **Example of Vulnerable Code (Using EntityManager):**

```java
String email = request.getParameter("email"); // user input
String query = "SELECT u FROM User u WHERE u.email = '" + email + "'";
List<User> users = entityManager.createQuery(query).getResultList();
```

If the `email` value is:

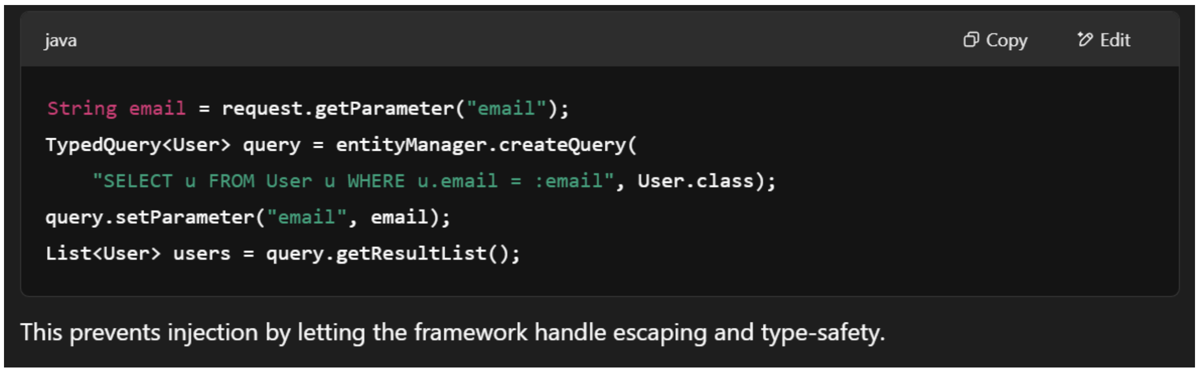
```bash
' OR '1'='1
```

The query becomes:

```sql
SELECT u FROM User u WHERE u.email = '' OR '1'='1'
```

This would return **all users**, bypassing security.

d. Safe Alternative: Use Parameter Binding:



```java
String email = request.getParameter("email");
TypedQuery<User> query = entityManager.createQuery(
    "SELECT u FROM User u WHERE u.email = :email", User.class);
query.setParameter("email", email);
List<User> users = query.getResultList();
```

This prevents injection by letting the framework handle escaping and type-safety.

9. What are EntityGraphs?

a. **EntityGraphs** in JPA are a way to **control how related entities are fetched** from the database — either eagerly or lazily — without changing the entity class annotations like `@OneToMany(fetch = ...)`.

b. Why Use EntityGraphs?

   i. To **optimize performance** by fetching only what you need

   ii. To **avoid N+1 select problems**

   iii. To customize fetching **at runtime**, instead of hardcoding it in entity mappings

c. Summary:

   i. **EntityGraphs** allow you to **dynamically specify fetch behavior**

   ii. They help optimize performance without changing entity annotations

   iii. Ideal for solving **lazy loading issues** and **over-fetching** problems in JPA

   iv. Other ways to solve lazy loading issues:

      1. JOIN FETCH - Simple, clean, good control over queries
      2. FetchType.EAGER - Easy, but can cause over-fetching
      3. Manual Initialization - Works inside transaction, not elegant for large use
      4. Open Session in View - Convenient, but risky and not ideal for scaling
      5. DTO Projection - Very efficient, clean API, avoids unnecessary loading
      6. EntityGraph - Flexible and declarative, especially for complex cases

SQL

1. What is the DISTINCT keyword used for in SQL?

   a. DISTINCT - is used to **remove duplicate rows** from the result set of a query.

2. What is an aggregation function in SQL? Do you know any examples?
   a. An **aggregation function** performs a calculation on a set of values and returns a single value.
      i. `COUNT()` – counts rows
      ii. `SUM()` – adds values
      iii. `AVG()` – calculates average
      iv. `MAX()` / `MIN()` – finds highest/lowest
3. What is the difference between an ***inner join***, a ***left/right join*** and a ***full join***?
   a. INNER JOIN - Returns only matching rows from both tables
   b. LEFT JOIN - Returns all rows from the left table, plus matching ones from the right
   c. RIGHT JOIN - Returns all rows from the right table, plus matching ones from the left
   d. FULL JOIN - Returns all rows from both tables; non-matching rows get NULLs
4. What is the difference between a left join and a left outer join?
   a. There is **no difference** — `LEFT JOIN` and `LEFT OUTER JOIN` are **synonyms**. "OUTER" is optional and usually omitted.
5. What could be the so called ***N+1*** problem regarding loading of data?
   a. The **N+1 problem** happens when:
      - 1 query loads a list of N parent records,
      - and then 1 **additional query per record** is executed to load related data.

      This results in
      N+1 total queries, which can be inefficient.

Common in ORMs like JPA/Hibernate when lazy loading is not managed properly.

VCS

1. Can you explain the git flow?
    a. Core Branches in Git Flow:
        i. **`main`** (or `master`)
            1. Always reflects **production-ready** code
            2. Deployments are made from here
        ii. **`develop`**
            1. Integration branch for **features**
            2. Represents the latest **in-progress** code for the next release
    b. Supporting Branches:
        i. **`feature/*`**
            1. For developing **new features**
            2. Branch from: `develop`
            3. Merge into: `develop`
            4. Naming: `feature/login-form`, `feature/payment-api`
        ii. **`release/*`**
            1. For preparing a **release** (bug fixes, polishing)
            2. Branch from: `develop`
            3. Merge into: `main` and `develop`
        iii. **`hotfix/*`**
            1. For fixing **critical bugs in production**
            2. Branch from: `main`

3. Merge into: `main` and `develop`

c. Git Flow Benefits:

    i. Clear separation of concerns (feature vs. production)

    ii. Safe and organized release management

    iii. Easy collaboration across teams