1. What is the difference between TDD and BDD?

TDD (Test-Driven Development) and BDD (Behavior-Driven Development) are software development methods that helps developer to collaborate and test the workload.

TDD(Test-Driven Development) is a process where developers write tests before writing the actual code.
It is driven by a cycle called "Red-Green-Refactor" cycle:
- *Red: Write a failing test for a specific piece of functionality.*
- *Green: Write the minimum amount of code necessary to make the test pass.*
- *Refactor: Improve the code's design without changing its behavior, guided by the test suite.*

Meanwhile Behavior-Driven Development (BDD) is pretty much the continuation where developers collaborate and test the work.
. The focus of BDD is on defining and understanding the behavior of a system in natural language, making it more accessible to non-technical team members. BDD uses a format called "Given-When-Then" to describe the behavior of a system:
- *Given: Describes the preconditions and initial state of the system.*
- *When: Describes the specific action or event that occurs.*
- *Then: Describes the expected outcome or behavior of the system.*

BDD frameworks like Cucumber allow these behavior specifications to be written in a human-readable format and then automatically translated into executable tests. On the other end TDD is primarily oriented towards ensuring code to be working and well maintain through unit testing, BDD focuses on building a shared understanding of the system's behavior across the entire team and ensures that the development process aligns with the expected business outcomes.

In summary, TDD is a testing methodology where tests are written before code to ensure individual units function properly, while BDD extends this concept to emphasize collaboration and communication by expressing system behavior in a more human-readable format, thereby promoting a clearer understanding of requirements and features.

https://phoenixnap.com/blog/tdd-vs-bdd

2.What is the value in separating your code into controller, service, and data access layers rather than keeping it all in the same files?

Separating code into distinct layers, such as controller, service, and data access layers, provides several advantages such as improved code organization, maintainability, and scalability in software development. Here are the main benefits of using this layered architecture:

1. Modularity and Reusability: By dividing the code into separate layers, each layer can focus on specific responsibilities. This modularity allows developers to reuse components in different parts of the application or even in other projects.

2. Maintainability: Separating concerns into different layers makes it easier to maintain and update the codebase. Changes made in one layer are less likely to affect the others, reducing the risk of introducing unintended side effects. This isolation of concerns also makes it easier to locate and fix bugs when they arise.

2. *Adaptability to Changes: As requirements evolve or change, a layered architecture allows developers to adapt more easily. New features or functionalities can be introduced at the appropriate layer without majorly impacting other parts of the application.*

3. Scalability: Layered architecture facilitates the scalability of the application. Different layers can be scaled independently, depending on the specific requirements of the application. For instance, if the data access layer experiences a performance bottleneck, it can be optimized or scaled separately from the other layers.

4. Testing and Debugging: Layered architecture supports better testing practices. Each layer can be tested independently using unit tests, allowing for more targeted and effective testing. Additionally, the separation of

concerns simplifies debugging, as it is easier to identify which layer is responsible for a particular issue.

5. Code Readability and Understandability: A well-structured layered architecture improves code readability and understandability. Developers can quickly grasp the purpose and responsibilities of each layer, making it easier for them to collaborate and work on the codebase as a team.

6. Enforcing Best Practices: Layered architecture naturally encourages the adoption of best practices, such as adhering to the Single Responsibility Principle (SRP) and the Separation of Concerns (SoC) principle. These principles promote cleaner code and make the codebase more maintainable over time.

7. Adaptability to Changes: As requirements evolve or change, a layered architecture allows developers to adapt more easily. New features or functionalities can be introduced at the appropriate layer without majorly impacting other parts of the application.

In conclusion, separating code into controller, service, and data access layers (commonly referred to as the three-tier architecture) provides a structured and organized approach to software development. It shows the code quality, simplifies maintenance, and supports the long-term growth and evolution of the application.

https://www.coreycleary.me/why-should-you-separate-controllers-from-services-in-node-rest-apis/