TASK

System Architecture

Visit our website

Introduction

WELCOME TO THE SYSTEM ARCHITECTURE TASK!

Software architecture refers to how a software system is designed and organised. It involves understanding the requirements of the system and selecting the best way to structure its components to achieve its desired functionality. In this task, we'll touch on some of the architecture patterns that developers use, such as layered architecture, repository architecture, client-server architecture, and pipe and filter architecture, which provide a set of design principles and guidelines to create more resilient and scalable software systems.

WHAT IS SOFTWARE ARCHITECTURE?

If you tried to build a house without planning it first, you can imagine things would not turn out very well. You might not buy the right amount of bricks (or even the right type), the walls may not align and the overall structure would have unstable foundations. The same would happen if you tried to build a system with code without system architecture — it is the blueprint based on your previously established requirements that will be the base of your designs for the system. This means that the decisions made in this phase of the design process will be the foundation for the development process later.

Software architecture aims to see how the components of a system are organised and put together, how they communicate with each other, and to be mindful of the constraints on the system as a whole. From there, the architect can decide on an architectural pattern that would best suit the system. We will be looking at a few different architectural patterns later in the task.

WHY IS SOFTWARE ARCHITECTURE IMPORTANT?

As mentioned above, it can be challenging to build something without planning how to do it. It could result in wasted time, wasted resources, and spending more money than necessary. Furthermore, Bass, Clements, and Kazman (2012, as cited by Sommerville, 2016, p. 169) discuss three advantages of implementing software architecture:

1. **Stakeholder communication:** because the architecture shows the big picture of a system, anyone involved in its creation should understand it and use it as a point of reference in discussions.
2. **System analysis:** documenting the architecture process early on helps to keep the process on track and ensure that all established requirements of the system are met. This level of analysis needed to document the process will ensure that everything is thought through thoroughly and planned accordingly.
3. **Large-scale reuse:** if an architectural model is designed and documented correctly, it could be reused for future systems with similar requirements.

ARCHITECTURAL PATTERNS

An architectural pattern is a description of a system of organisation developed informally over time (Garlan and Shaw, 1993, p. 5). It can be thought of as a stylised, abstract description of good practice which has been tried and tested in different systems and environments. The pattern chosen will depend on the non-functional requirements of the system, such as (Sommerville, 2016):

- **Performance:** keeping operations localised
- **Security:** keeping sensitive data buried
- **Safety:** keeping safety components centralised and secure
- **Availability:** including redundancies that are available for hot-swapping (implementing new components without needing to stop the system)
- **Maintainability:** creating components that are replaceable when updating or upgrading is needed

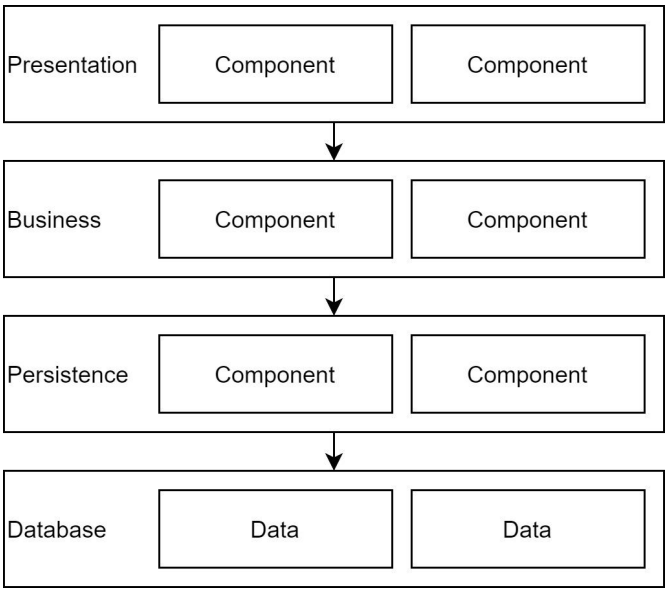
Below are summaries of some of the more common patterns used in system design.

LAYERED ARCHITECTURE

As the name suggests, in this pattern we separate all the components into layers. Each layer is only able to access the services of the layer directly below it. This means that the system can be developed incrementally. It also means that the outermost layers are easily changed because only the inner layers need to interface

with hardware or operating systems.

Most systems will have four layers: presentation, business, persistence, and database (Richards, 2015). Each layer has a designated purpose: the **presentation** layer is the user interface component, the **business** layer is responsible for aspects like user authentication or authorisation, the **persistence** layer deals with functional components, and the **database** layer, as the name suggests, is the backend component that contains the database. Below is an illustration based on Richards' description (2015):



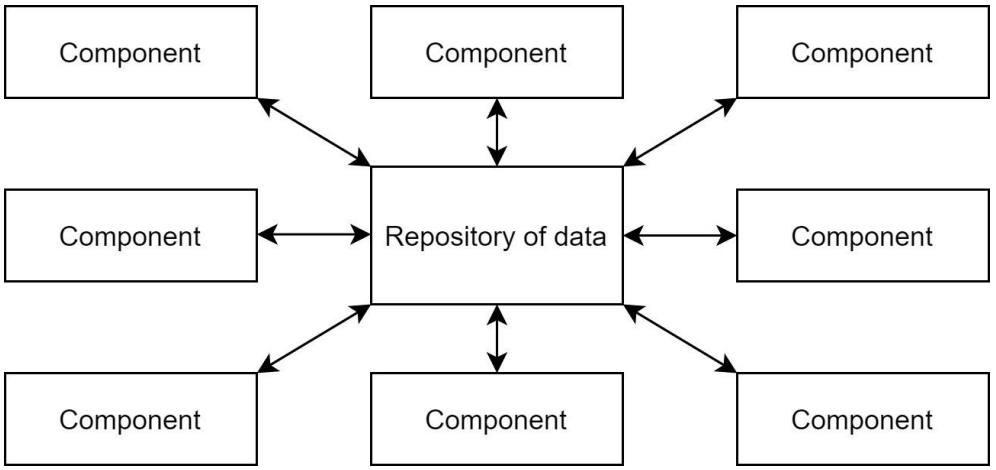
Note that each layer contains the components that apply to that particular element of the system. Furthermore, note how each layer has a level of separation and independence — this makes the components of systems with a layered architecture easy to develop and test independently from the rest of the system.

Take note:

The diagram above is known as a **block diagram**. Each block represents a component, and a block within a block represents a component that has been broken down into smaller components. You can also see how the arrows show the direction of the flow of data. These are simple diagrams that show a broad overview of a system — they do not show the types of relationships between components or any visible elements of a component — but that is what makes it a very useful tool for describing the architecture of a system to all stakeholders. However, when it actually comes to developing the system, the developers are going to need more detail.

REPOSITORY ARCHITECTURE

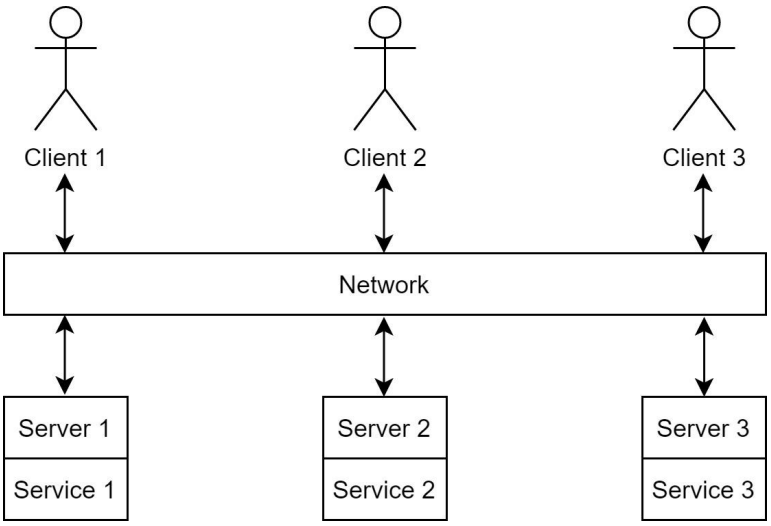
In this architecture pattern, all data is kept in a central repository and all components are separated — they can, however, interact through the repository. This is a particularly useful pattern if you have a lot of data. Have a look at the block diagram of the pattern below:



As you can see, there is a **bidirectional** flow of data to and from the repository, which means it is easy to input and output data as well as for components to interact with one another via the repository. However, one drawback is that the repository is a single point of failure. This means that a problem occurring in the repository could impact the whole system.

CLIENT-SERVER ARCHITECTURE

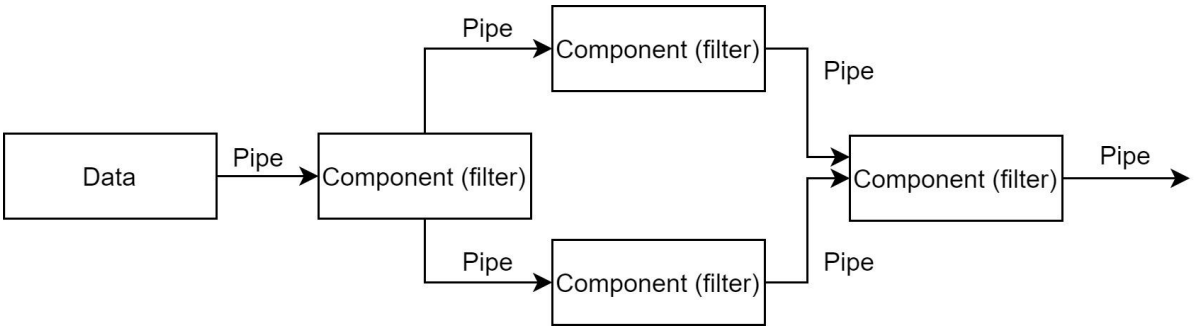
In this pattern, a set of services is developed to run on servers. The clients then access the servers over a network to make use of the system. The network usage allows multiple clients to access the servers at one time to access the services that the system provides. Look at the block diagram below:



You can see from the diagram above how vital a reliable network connection is. Without it, clients are unable to access the servers and, therefore, the system's services. This gives the client-server pattern a single point of failure, like the repository pattern.

PIPE AND FILTER ARCHITECTURE

In this pattern, the data provided goes through a series of transformations (filters) as it flows from one component to another. This pattern is particularly useful for data processing.



Looking at the diagram, you can see how transformations can occur in series as well as in parallel. This type of structure works well with the way that data-driven businesses already work, so the system would be fairly simple to integrate. However, because this system requires that input is parsed (broken up into smaller components) as input and unparsed for output through the transformations, this could lead to increased system computation time. This means that the system might be difficult to upgrade or update.

As a Software Engineer, it will be important for you to work with the Software Architect to determine the best pattern for your given system based on user and system requirements, the components needed, how they need to interact with one another, and the constraints of the system. Proper planning and documentation of the design process will make the development phase more straightforward and efficient and you will end up with a higher-quality system.

Compulsory Task

Create a text file called **architecture.txt**. Inside, answer the following questions:

- Do some research to find real-world examples of when each architecture pattern would be most appropriate to use. Write down your examples, and state your reasons why each pattern would be most appropriate for each of the examples you found.

- Give two examples of systems combining two or more patterns, and explain how these patterns complement each other.

Completed the task(s)?

Ask an expert code reviewer to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

Garlan, D., and M. Shaw. 1993. "An Introduction to Software Architecture." In *Advances in Software Engineering and Knowledge Engineering*, edited by V. Ambriola and G. Tortora, 2:1–39. London: World Scientific Publishing Co.

Richards, M. (2015). *Software Architecture Patterns* (1st ed., pp. 1-8). USA: O'Reilly Media, Inc.

Sommerville, I. (2016). *Software Engineering* (10th ed., pp. 167-195). Essex: Pearson Education Limited.