COMP2511 Tutorial 9

Architectural Characteristics,
Architectural Styles

Notices

- Assignment-ii (both parts) is due **next Friday 3pm**.
    - A reminder that no submissions are accepted past this time, including Special Considerations and ELPs.
    - Make sure you aren't pushing code right before the deadline!
- A sample exam environment will be running on the lab computers next week.
    - The aim is to get you comfortable with the **structure** and **layout** of the exam.
    - The difficulty of the exam is not indicative of the actual exam.

Software Architecture

- Suppose you've built an online banking app that looks great and has a bunch of features that all work correctly. However, it crashes if more than 100 users log on at the same time. Is this a successful system?
    - Not necessarily; the system is missing **scalability**, the ability to handle growth in workload or user traffic without compromising performance.
    - Even if all the features work perfectly, the application isn't going to be very useful if we can't serve many users concurrently.
    - For software to be considered 'good', it has to satisfy both functional *and* non-functional requirements.

Architectural Characteristics

- **Architectural characteristics** (*non-functional requirements*) define fundamental qualities that software architecture must support.
    - For example, we may consider the **maintainability** of a system: how easy is it to fix, update or extend the system?
- Unlike functional requirements, they are often not explicitly defined.
    - This adds another layer of complexity to designing software; these are considerations that we have to be aware of and account for accordingly, and are typically ‘implicitly there’.

Architectural Characteristics

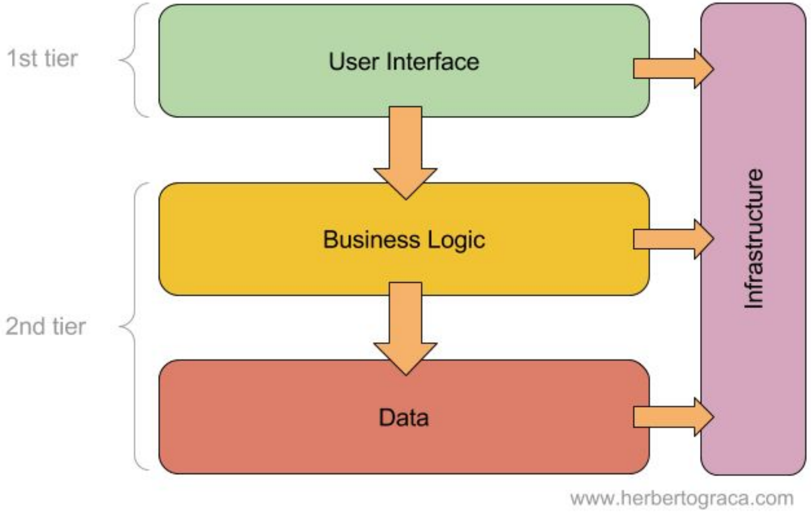
- More examples of some architectural characteristics, **not** exhaustive:
  - **Performance**: How fast does the system run?
  - **Availability**: How often is the system operational?
  - **Fault Tolerance**: How robust is the system against particular things not working?
  - **Reliability**: How error-prone is the system?
  - **Security**: How protected is the system against threats? Are there any vulnerabilities?
  - **Testability**: How easy is it to test the system?
  - **Data Consistency**: Do each of the components ‘agree’ on what data is in the system?
- Characteristics such as these influence how we decide to design software. Some characteristics will be more important than others, depending on the context and specific requirements.
  - At the end of the day, there are always **trade-offs** across characteristics. For example, more security → lower performance (more robust checking) and vice versa.

Architectural Styles

- **Architectural styles** are predefined patterns and philosophies guiding how software systems are structured.
- Common architectural styles include:
  - **Layered** architectures (**monolithic** - deployed as a single unit)
  - **Modular monolithic** architectures (also monolithic)
  - **Microservice** architectures (**distributed** - deployed as many smaller services that are independent and communicate with each other)
  - **Event-driven** architectures (also distributed)
- Each style comes with trade-offs for specific architectural characteristics.
- The style of a system will influence how it behaves and evolves. It is **very** difficult to convert a system from one style to another!

Layered Architecture

- A layered architecture is a monolith where distinct responsibilities are separated into different technical layers, defining a clear separation of concerns.

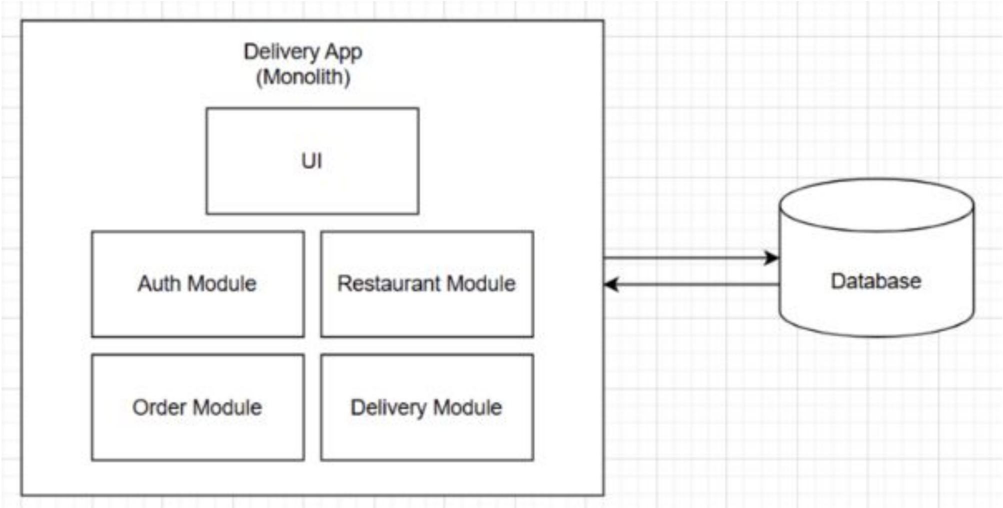


Layered Architecture

- Strengths of layered architectures:
  - **Simplicity** - in general, monoliths are typically the ‘most immediate solution’ and are the easiest to ‘get up and running’, as there is only one component to worry about
  - **Maintainability** - one technical layer can be modified without affecting other layers
  - **Data Consistency** - all layers share a single database, so data is consistent throughout
  - **Security** - each layer can handle its own separate protocols and be responsible for specific types of threats
- Weaknesses of layered architectures:
  - **Performance** - messages throughout the system have to pass through each of the layers, which may add a considerable overhead depending on the scenario/queries
  - **Scalability** - individual layers are hard to scale without changing other layers as well
  - **Fault Tolerance** and **Availability** - if one part of a monolith breaks, then the rest of the monolith is likely to also break; also another consideration for security

Modular Monolithic Architecture

- Modular monoliths are monoliths that are divided according to different business domains, rather than technical layers.
- Each module is responsible for a specific functional requirement.

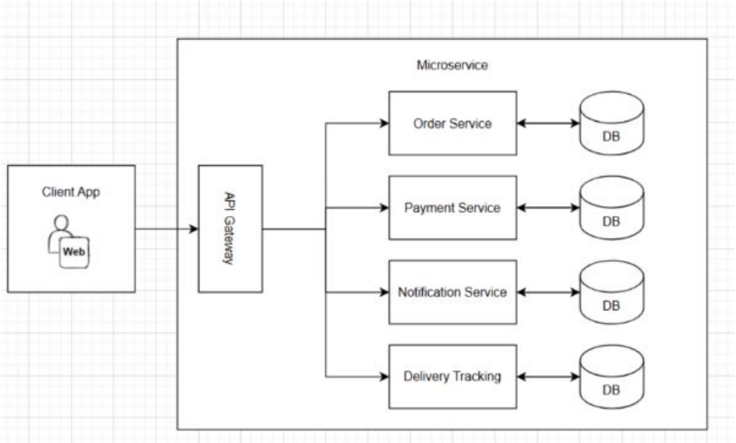


Modular Monolithic Architecture

- Strengths of modular monoliths:
  - **Maintainability** - code localised to a specific function/domain doesn’t need to interact with other modules
  - **Performance** - likely more performant than layered, since data processing for a specific domain occurs in a ‘localised’ spot rather than having to cross several layers
  - **Data Consistency** - same as layered architecture (single shared database)
  - **Security** - can separate security handling into its own module, and highly sensitive operations can be isolated
- Weaknesses of modular monoliths:
  - **Scalability** - may be easier to scale than layered architectures if particular modules need to be scaled, but the shared database still acts as a bottleneck
  - **Fault Tolerance** and **Availability** - same as layered architecture (monoliths in general)

Microservice Architecture

- Microservices are single-purpose, individually deployed units.
- Each microservice should manage its own data and perform one specific function very well.
- Microservice architectures organise systems around these microservices, which communicate with each other over a network (can be sync/async).

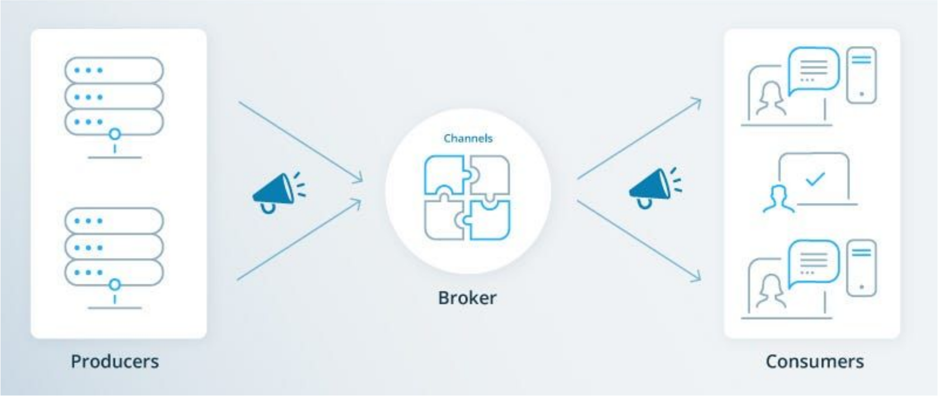


Microservice Architecture

- Strengths of microservice architectures:
    - **Scalability** - each microservice can be scaled independently
    - **Fault Tolerance** and **Availability** - a fault in one microservice does not mean the entire system will break down as each microservice is deployed independently
    - **Security** - a vulnerability in one microservice does not expose the whole system
    - **Maintainability** - easier to isolate specific issues due to separated responsibilities
- Weaknesses of microservice architectures:
    - **Cost** - a lot of infrastructure and networking is required compared to a monolith
    - **Data Consistency**: data within the same microservice is fine (each microservice manages its own data), but needs to be synchronised across different microservices
    - **Security** (again) - distributed systems have more ‘access points’ for attacks (‘higher attack surface’) and particular microservices could be ‘weak spots’

Event-Driven Architecture

- Event-driven architectures structure systems around components that respond to and broadcast particular *events*, which notify of significant changes in system state (somewhat similar to the Observer Pattern).
- Most communication inside of the system occurs *asynchronously* - system components send messages without waiting for responses.



Event-Driven Architecture

- Strengths of event-driven architectures:
  - **Responsiveness**: reacts to real-time events very quickly
  - **Performance**: asynchronous messaging prevents services from having to ‘wait’ on each other to complete a request
  - **Scalability**: easy to scale, since more events → more publishers and subscribers
  - **Decoupling**: publishers don’t need to know anything about subscribers due to the broker
- Weaknesses of event-driven architectures:
  - **Complexity**: asynchronous messages can be difficult to understand and reason with, since actions don’t necessarily occur in pre-defined orders
  - **Testability**: debugging asynchronous code is also complicated, as errors aren’t immediate and it can be hard to localise particular problems.
  - **Data Consistency**: data across components may be inconsistent at a fixed point in time, and will take time before data ‘converges’ to an agreed state