Chapter 4 – Architectural Design

Slides prepared by Ian Sommerville, modified by Ali El Hajj

Topics covered

- Architectural design decisions
- Architectural patterns
- Application architectures

Software architecture

- Architecture is the organization of a software system into sub-systems and components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
- Can be represented using block diagrams showing entities and relationships
- Architectural design is :
    - An early stage of the system design process.
    - Can be carried out in parallel with some specification activities.
    - Represents the link between specification and design processes.
    - An early stage of agile processes: Changing the architecture is expensive because it affects many components in the system.

Architecture has a fundamental influence on the non-functional system properties:

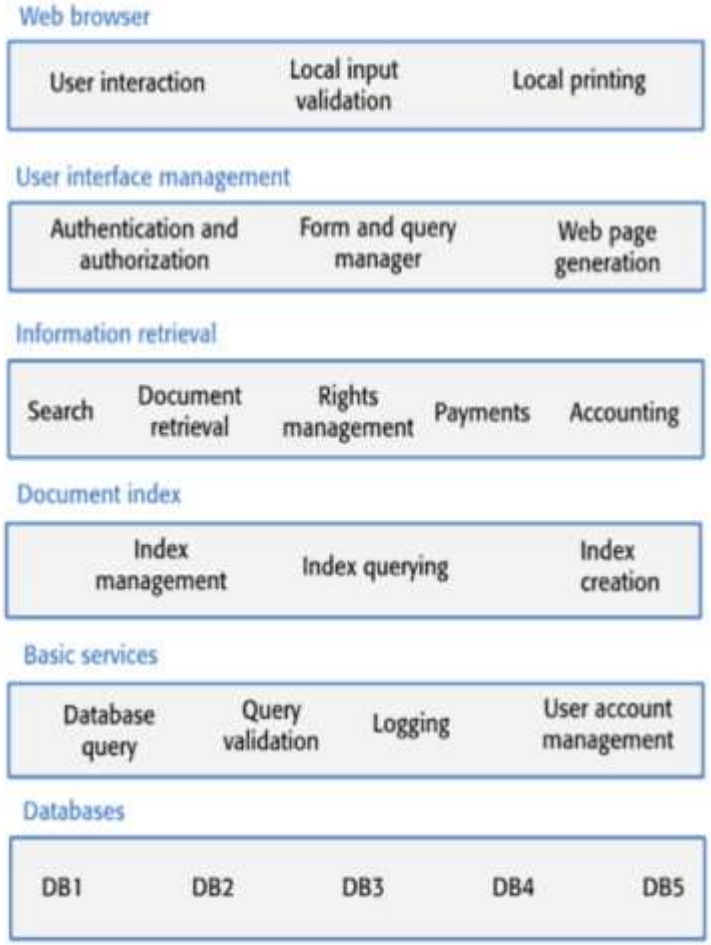
- Response time
- Reliability
- Availability.
- Security
- Usability.
- Maintainability.
- Resilience.

Advantages and use of explicit architecture

- Stakeholder communication
  - Architecture may be used in discussions by system stakeholders. Stakeholders can understand an abstract view of the system as a whole without being confused by details.
- System analysis
  - Whether the system can meet its non-functional requirements.
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Systems in the same domain often have similar architectures that reflect domain concepts.
  - The architecture of a system may be designed around one of more architectural patterns or 'styles'.
- Documenting an architecture
  - Showing the different components in a system, their interfaces and their connections.

Software Design Abstraction

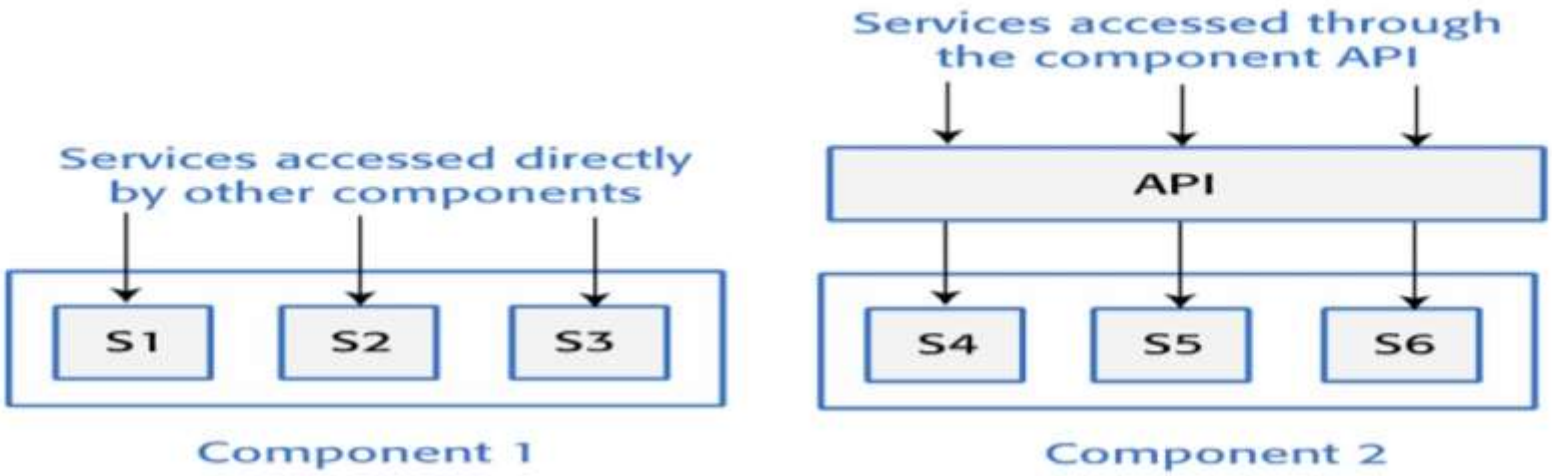
- At the architectural level, your concern should be on large-scale architectural components or sub-systems.
- You don't have to decide how an architectural element or component is to be implemented.
  - You design the component interface and leave the implementation to a later stage.
- Decomposition involves analysing these large-scale components and representing them as a set of finer-grain components.



Layered Architecture of a document retrieval system

Software Components

- A component is an element that implements a coherent set of functionality or features.
  - It can be a collection of one or more services that may be used by other components.



Access to services provided by components

Architectural design issues

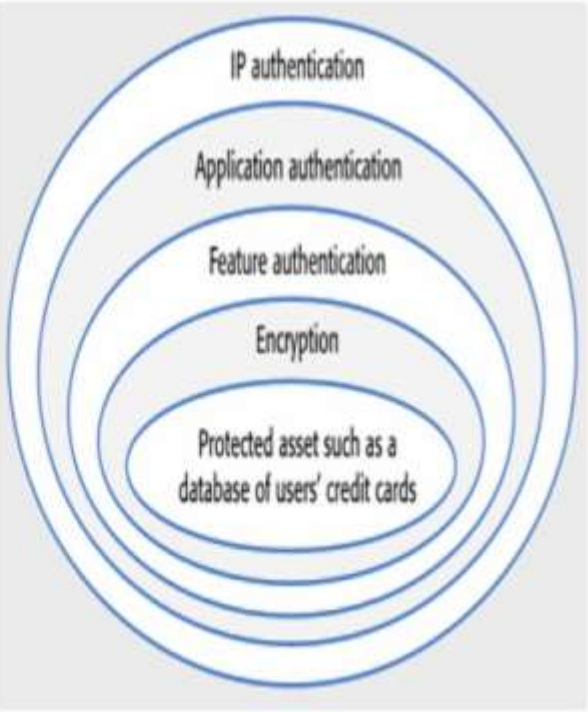
- ***Nonfunctional product characteristics***
  If you get Nonfunctional characteristics wrong, your product will not be a success. Some characteristics are opposing, so you can only optimize the most important.
- ***Product lifetime***
  If you anticipate a long product lifetime, you will need to create regular product revisions. You need an architecture that is evolvable.
- ***Software reuse***
  You can save time and effort, if you can reuse large components from other products or open-source software. You must fit your design around the software that is being reused.
- ***Number of users***
  If the number of users can change very quickly, you need to design your architecture so that your system can be scaled up and down.
- ***Software compatibility***
  For some products, it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system. This may limit architectural choices, such as the database software that you can use.

Architecture and system characteristics

- Maintainability
  - Use fine-grain, self-contained, replaceable components.
  - However, it takes time for components to communicate with each other. If many components implement a product feature, the software will be slower.
- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.
- Security
  - Use a layered architecture with critical assets in the inner layers.
- Safety:
  - Localise safety-critical features in a small number of sub-systems. These are well tested.
- Availability: Include redundant components and mechanisms for fault tolerance.
  - To make use of redundancy, you include sensor components that detect failure, and switching components that switch operation to a redundant component when a failure is detected.
  - Implementing extra components takes time and increases the cost of system development. It adds complexity to the system and therefore increases the chances of introducing bugs and vulnerabilities.

Security vs usability

- Security: Designing the system protection as a series of layers. An attacker has to penetrate all of those layers.
  - Layers might include authentication layers, an encryption layer and so on.
- A layered approach to security affects the usability of the software.
  - Users have to remember information, like passwords. Their interaction with the system is slowed down by its security features.



Architectural design decisions

- Is there a generic architecture that can be used?
  - What architectural patterns or styles are appropriate? (MVC, shared repository, layered, client-server…)
  - Most large systems are heterogeneous and do not follow a single architectural style
- What architectural design is best for delivering the non-functional requirements?
- How will the system be distributed across hardware?
- What approach to use to structure the system? (top-down, bottom-up)
- What control strategy should be used? (centralized, event-based…)
- How should the architecture be documented?

Control styles

- concerned with the control flow between sub-systems.
- Centralised control:
  - One sub-system has overall responsibility for control and starts and stops other sub-systems.
- Event-based control
  - Each sub-system can respond to externally generated events from other sub-systems or the system's environment.

Architectural patterns

Architectural patterns

- An architectural pattern is a description of good design practice, which has been tried and tested in different environments.
- Patterns are a means of representing, sharing and reusing knowledge.
- Patterns may be represented using graphical and tabular descriptions.

Layered architecture

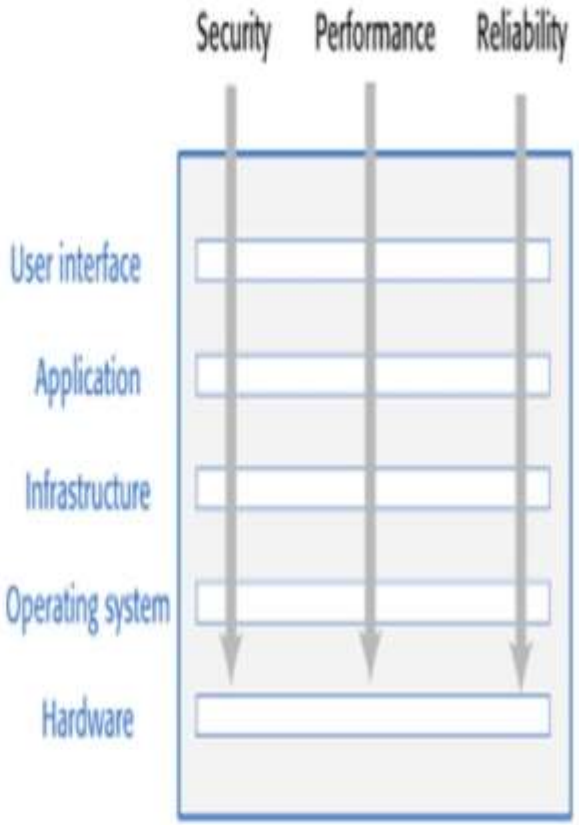
- Organises the system into a set of layers, each providing a set of services to the layer above.
  - Each layer is an area of concern and is considered separately from other layers.
  - Within each layer, the components are independent and do not overlap in functionality.
- Supports the incremental development of sub-systems in different layers.
  - When a layer interface changes, only the adjacent layer is affected.
  - Allows replacement of entire layers so long as the interface is maintained.
- Performance can be a problem because a service request is processed at each layer.
- Redundant facilities (e.g., authentication) can be provided in each layer to increase dependability.

The Layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a system for sharing copyright documents held in different libraries.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Cross-cutting concerns

- Cross-cutting concerns are concerns that affect the whole system.
- In a layered architecture, cross-cutting concerns affect all layers in the system.
  - They are different from the functional concerns.
  - There are interactions between the layers because of these concerns.
- Security as a cross-cutting concern: you need protection from attacks at each layer.
  - By distributing security across the layers, your system is more resilient to attacks and software failure
- The existence of cross-cutting concerns is the reason why modifying a system after it has been designed to improve its security is often difficult.



A generic layered architecture for a web-based application

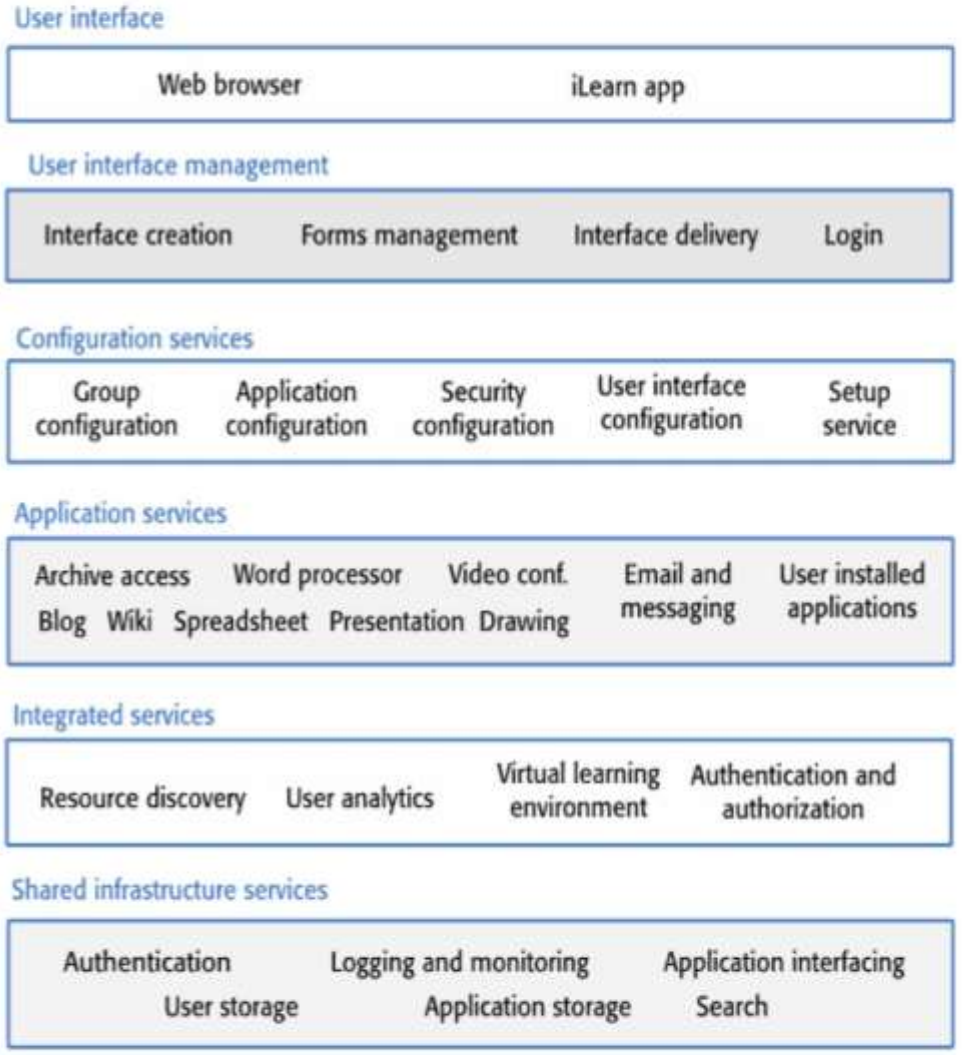
- ***Browser-based or mobile user interface***
  A web browser system interface in which HTML forms are often used to collect user input. Javascript components for local actions, such as input validation, should also be included at this level. Alternatively, a mobile interface may be implemented as an app.
- ***Authentication and UI management***
  A user interface management layer that may include components for user authentication and web page generation.
- ***Application-specific functionality***
  An 'application' layer that provides functionality of the application. Sometimes, this may be expanded into more than one layer.
- ***Basic shared services***
  A shared services layer, which includes components that provide services used by the application layer components.
- ***Database and transaction management***
  A database layer that provides services such as transaction management and recovery. If your application does not use a database then this may not be required.

iLearn architectural design principles

- *Replaceability*
  It should be possible for users to replace applications in the system with alternatives and to add new applications.
- *Extensibility*
  It should be possible for users or system administrators to create their own versions of the system, which may extend or limit the 'standard' system.
- *Age-appropriate*
  Alternative user interfaces should be supported so that age-appropriate interfaces for students at different levels can be created.
- *Programmability*
  It should be easy for users to create their own applications by linking existing applications in the system.
- *Minimum work*
  Users who do not wish to change the system should not have to do extra work.

iLearn as a service-oriented system

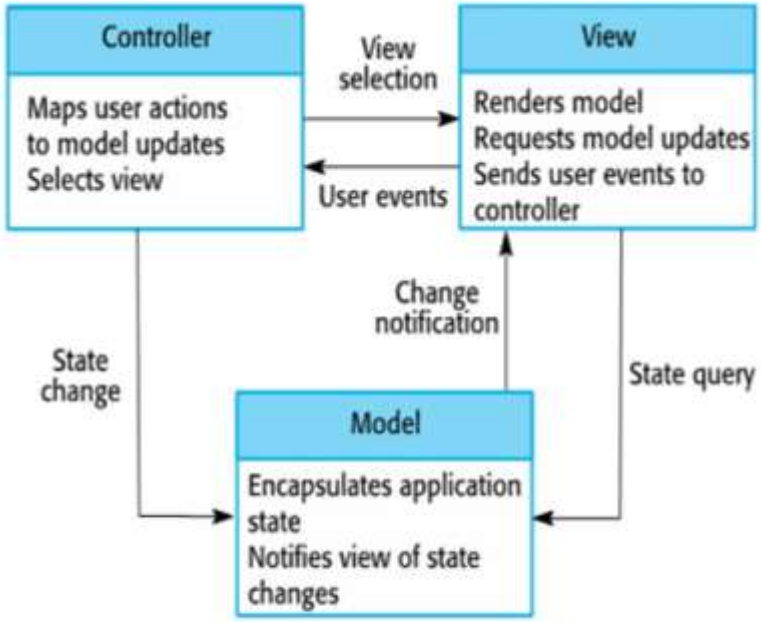
- These principles led us to an architectural design decision that the iLearn system should be service-oriented.
  - Every component in the system is a service. Any service is replaceable and new services can be created by combining existing services.
  - Different services delivering comparable functionality can be provided for students of different ages.

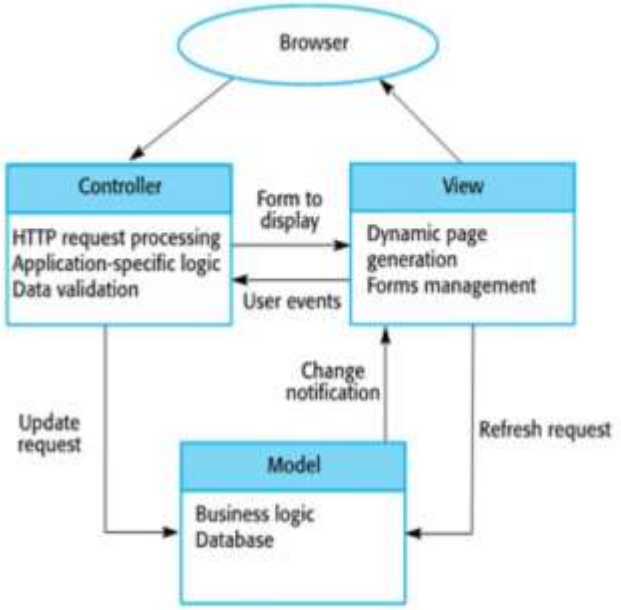


Model-View-Controller (MVC) Architecture

Three logical components:

- The Controller component manages user interaction.
- The Model component includes business logic and data.
- The View component defines and manages how the data is presented to user.





Web architecture of MVC pattern

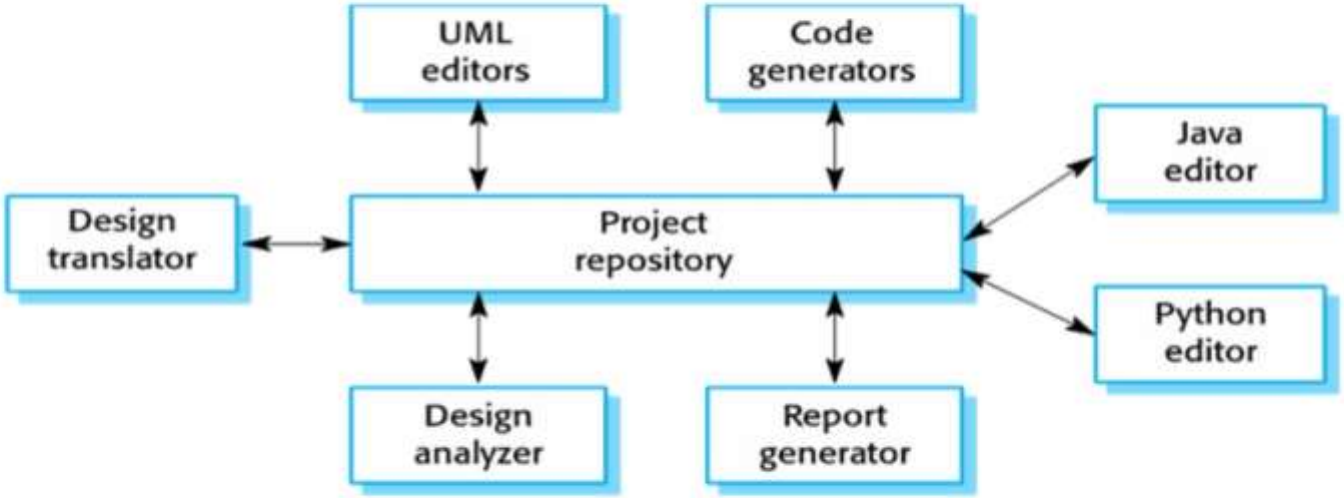
The Model-View-Controller (MVC) pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.
Example	Figure shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Repository architecture

- Sub-systems exchange data. This can be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model is commonly used. It is an efficient data sharing mechanism.

A repository architecture for an IDE

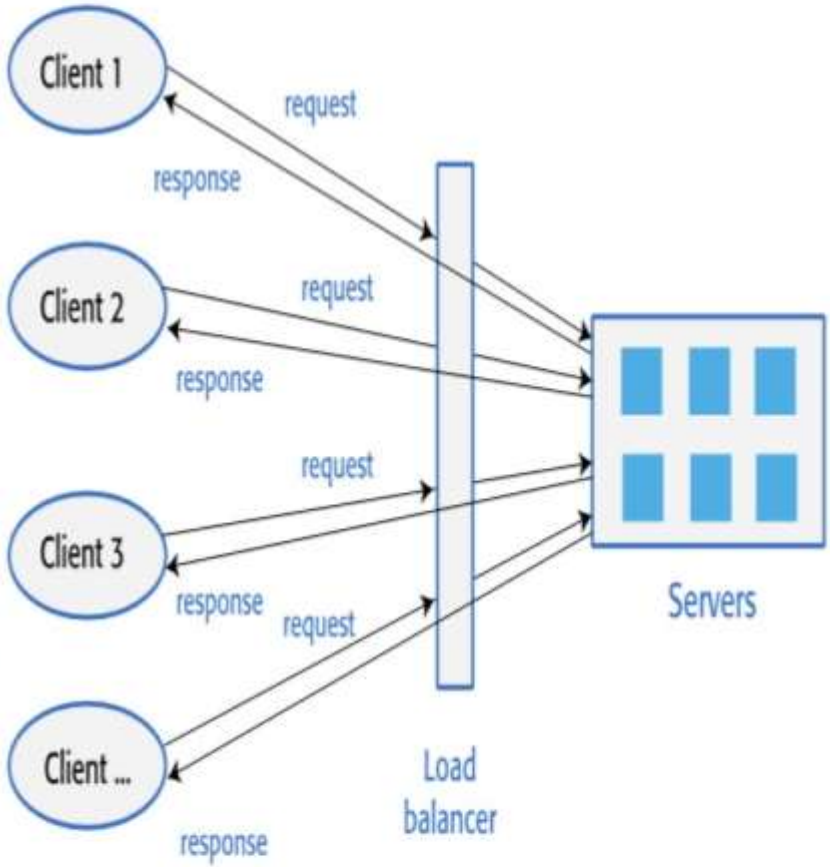


The Repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Client-Server Architecture

- The distributed architecture of a software system defines the servers in the system and allocation of components to servers.
- Client-server architectures are a type of distributed architecture where clients access shared services.
  - The user interface is implemented on the client computer.
  - Functionality is distributed between the client and one or more server computers.



The Client–server pattern

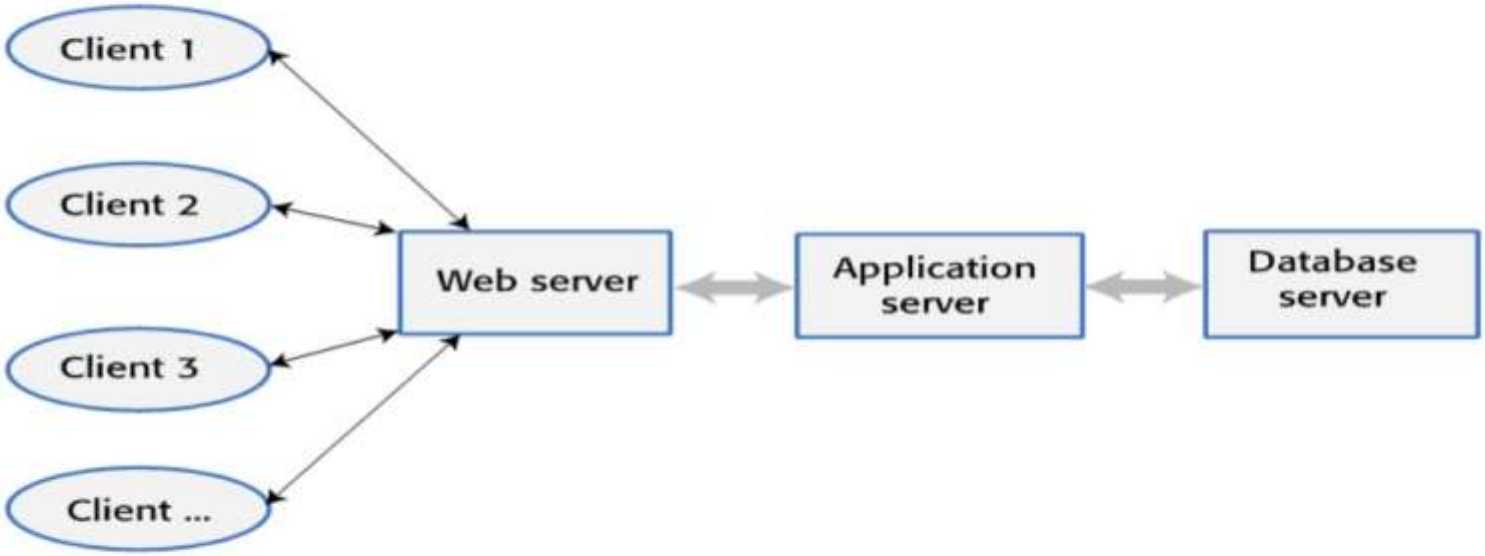
Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	A film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Client-server communication

- Client-server communication normally uses the HTTP protocol.
  - The client sends a message to the server that includes an instruction such as GET or POST along with the identifier of a resource (usually a URL) on which that instruction should operate. The message may also include additional information, such as information collected from a form.
- HTTP is a text-only protocol so structured data has to be represented as text. There are two ways of representing this data that are widely used, namely XML and JSON.
  - XML is a markup language with tags used to identify each data item.
  - JSON is a simpler representation based on the representation of objects in the Javascript language.

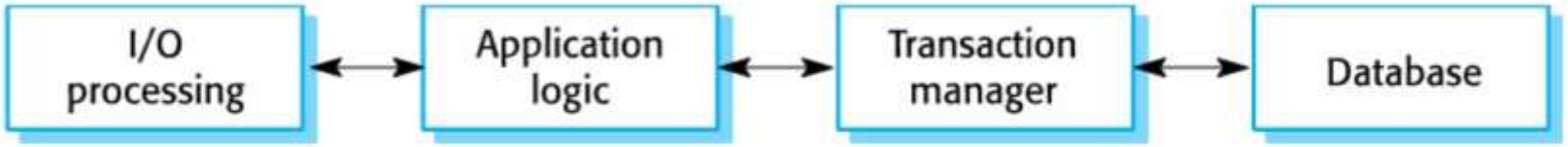
Multi-tier client-server architecture

- The web server is responsible for all user communications;
- The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
- The database server moves information to and from the database and handles transaction management.



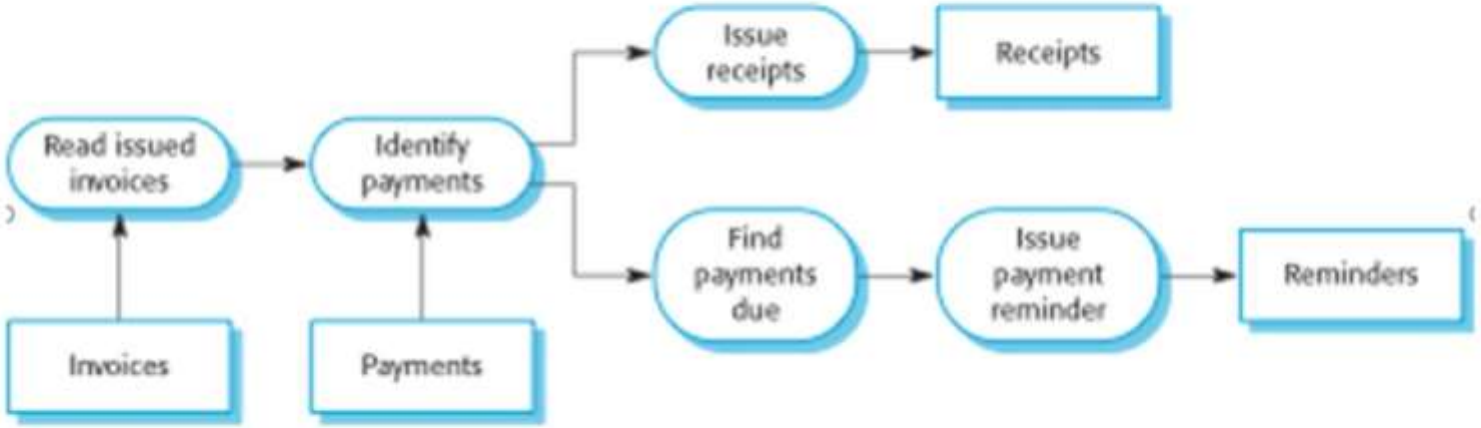
Transaction processing systems

- Process user requests for information from a database or requests to update the database. Examples: E-Commerce, Airline reservation
- From a user perspective a transaction is:
  - Any coherent sequence of operations that satisfies a goal;
  - For example – transfer funds from account A to account B
- Users make asynchronous requests for service which are then processed by a transaction manager.



Pipe and filter architecture

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model.
- When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not suitable for interactive systems.

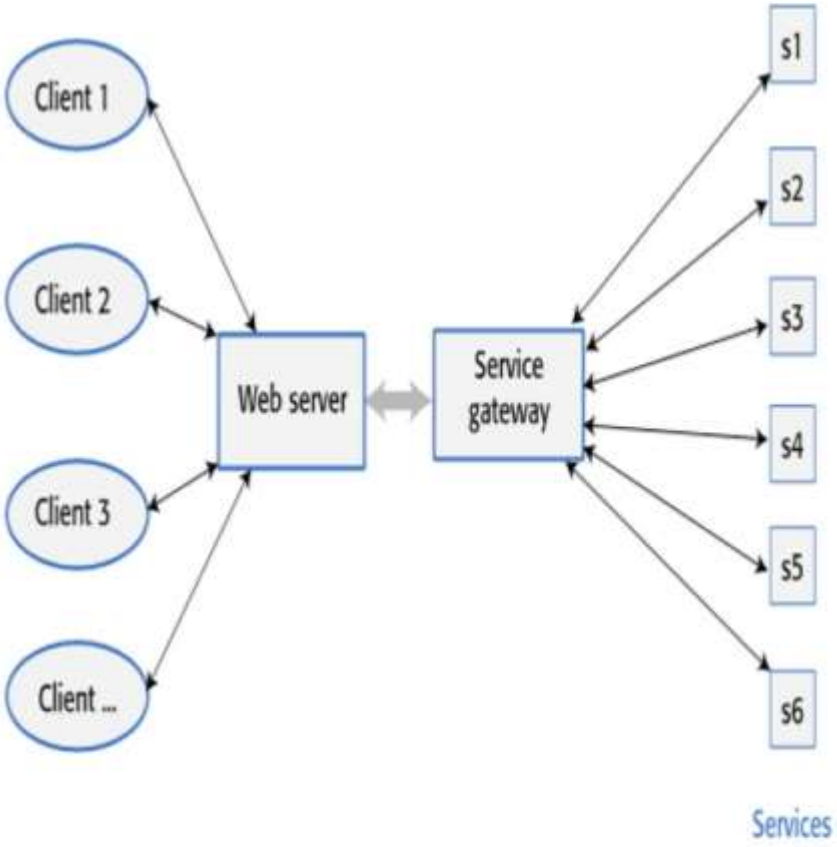


The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

Service-oriented architecture

- Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another.
- A service gateway is a single access point and acts as a proxy for multiple services.
  - It enables transformations, routing, and common processing across all the services.
  - Clients do not need to know what communication protocol is being used.
- Many servers may be involved in providing services
- A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure



Issues in architectural choice

- Data type and data updates
  - If you are mostly using structured data that may be updated by different system features, it is best to have a single shared database that provides locking and transaction management.
  - If data is distributed across services, you need a way to keep it consistent and this adds overhead to your system.
- Change frequency
  - If system components will be regularly changed or replaced, then isolating these components as separate services simplifies those changes.
- The system execution platform
  - If you plan to run your system on the cloud, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler.
  - If your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

Technology choices

- *Database*
  Relational databases, such as MySQL, are suitable for transaction management. NoSQL databases, such as MongoDB, are more flexible for data analysis.
- *Platform:*
  Should you deliver your product on a mobile app and/or a web platform? You may need a completely different decomposition architecture in different versions to ensure that performance and other characteristics are maintained.
- *Server*
  Should you use dedicated in-house servers or design your system to run on a public cloud? If a public cloud, should you use Amazon, Google, Microsoft, or some other option?
- *Open source*
  Are there suitable open-source components that you could incorporate into your products? Your choice should depend on the type of product that you are developing, license issues, your target market and the expertise of your development team.
- *Development tools*
  Do your development tools embed architectural assumptions about the software being developed that limit your architectural choices?