REST APIs
1.Stateless*
2.Cacheable
3.Layered System
4.Uniform Interface*
5.Code-on-Demand

Pros/Cons:

Systems that are less tightly coupled

Provides scalability, usability, astandardized, and mashup-ability ☺

Being stateless may decrease network performance by increasing the repetitive data sent in series of queries

Using URI may degrade efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs

Uniform Interface: There should be a uniform way of interacting with a given server irrespective of device or type of application

Principle: Generality to the component interfaceImplementations are decoupled from the services they provide

System is free from unnatural restrictions and limitations

Should be efficient for large-grained hypermedia data transfer Exploits HTTP/HTTPS requests and responsesRequests: GET, PUT, POST, DELETE

Responses: HTTP status codes (e.g., 200 OK, 404 NOT FOUND)Use the HTTP body to transfer the resource representation

Anything can be a Resource(Resource-based)Key abstraction of information in REST

E.g., document, image, user etc.,

•Resource is a conceptual mapping to a set of entities

•Resources are identified using stable, global, and unique resource identifiersE.g., / device-management/managed-devices/ {device-id}

•Components perform operations on resource representationsRepresentation consists of data, metadata describing the data

Client has representation of resource and contains enough information to modify the resource on the server

E.g., HTTP DELETE /device-management/ managed-devices/{device-id}

For interactions with resources through the uniform interface, REST needs self-describing messagesEach message includes enough information to describe how to process the message so that server can easily analyze the request.

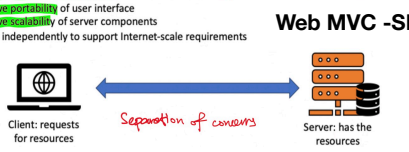
Components can get a complete understanding of resources or relevant states by just inspecting representations

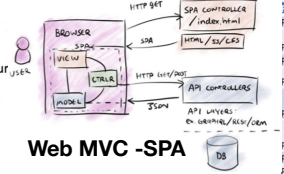
Hypermedia Driving Application State: representations that are exchanged should be linkedClients that need the resource can discover them and obtain them from the links

These links enable exposing new resources and application state transitions

Constraint: Client-server

- REST applications should have a client-server architectur
- Reason: Separation of concerns
  - E.g., Client not concerned with data storage of the resource; Server not concerned with user interface or user state
  - Improve portability of user interface
  - Improve scalability of server components
  - Evolve independently to support Internet-scale requirements





Web MVC -SPA

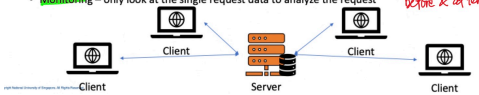
Ubiquitous Language: A shared language between domain experts(Business experts) and developers(Technical Experts)

Constraint: Stateless

- Each interaction between a client and the server has to be entirely self-contained
  - Full information contained as a part of the query params, headers or URI
- No client state (or "session") maintained on the server
- Server is bound by number of concurrent requests and not the number of clients interacting
- Reason: To improve
  - Scalability – server can quickly free resources to serve additional requests
  - Reliability – can recover from partial failures
  - Monitoring – only look at the single request data to analyze the request

The request should provide all the data needed

Can request again, dont worry about what happen before & after



FR: specifies sth system should do

- Specifying behaviour/function
- Eg. The passenger shall be able to print Boarding passes for all flights segments for which she has checked in

NFR: describes sth not directly related to the functionality of the system

- May describe how well the system works
- Can include attributes (Reliability, efficiency, usability)
- Eg) Mean time between failure >= 9000 hrs

FR Label	Description	Priority
FR1	Authentication	
FR1.1	System to provide a login page for the user to authenticate	H
FR1.2	System to provide a sign up page for the user to register	H
FR1.3	System to have authorisation - only authorised users should be able to add papers to db (F3.2)	
FR1.4	System automatically assign new users to the application as a public user with limited access	L
FR1.5	System should have an audit log that logs 'dangerous'/ 'significant' actions that have been taken by users.	L
FR2	Submission Data Visualization	
FR2.1	Shows visualisations of charts, graphs on a per paper, per topic basis	H
FR2.2	Visualization should be customisable, so that the same data can be viewed in a multitude of different ways based on the user preference,	M
FR2.3	Export generated visualizations and conference data in various formats.	L
FR2.4	ChairVise should show aggregated counts of submissions by category *authors/reviews/acceptance*	H
FR2.5	Allow reviewers to see basic level of visualization (e.g., completed reviews by topic, by area; pending revivews; review responses statistics)	L
FR3	Conference management	
FR3.1	Allows PC to review papers submitted	H
FR3.2	Allows PC members to view other's papers	H
FR3.3	The application allows researchers to submit and edit their paper	H
FR3.4	Enable Chairpersons to track progress of review process, including number of papers reviewed and review deadlines	H
FR4	Historical data preservation	
FR4.1	ChairVise should retrieve data from the database to show statistics of previous versions of the conference	L
FR4.2	ChairVise should generate appropriate statistics based on the submissions, acceptance buckets (e.g., strong accept/weak accept), rejection buckets etc.,	H
FR4.3	ChairVise should save the data from FR4.2 for persistence and retrieval (e.g., use case such as FR4.1)	M

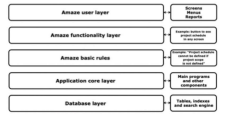
N1	Data security and privacy
N1.1	Only authenticated users can add, modify or delete papers
N1.2	Audience members have read-only access to the statistics/visualisations.
N2	Performance & Usability
N2.1	List of papers should be retreived within X seconds
N2.2	Visualisations and statistics should load within X seconds.
N2.3	Visualisations should be meaningful to chairperson
N3	Compatibility
N3.1	Able to run on all major browers (Chrome, Firefox, Edge etc)
N4	Scalability
N4.1	Able to manage 2000 papers per conference
N4.2	Scale data capacity to 1400 papers when number of submissions reach 1000
N5	Reliability & compliance
N5.1	Data backups to act as failsafe
N5.2	Data is stored following relevant data protection regulations

DevOps is a set of software development practices that combine development and operations. Intended to reduce the time between committing changes to the change hitting production, while ensuring quality.
• C-Int.: Auto build, unit test, deploy to staging, and acceptance test. Allows teams to detect problems early.
• C-Del.: Same as above, except with manual deployment to production. Ensures that every good build is potentially ready for production release.
• C-Dep.: Same as above but with auto deployment to production. Automates the release of a good build.

Layered Architecture

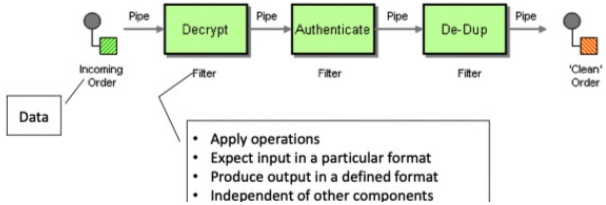
- Software is organized as layers of components
- Supports independent development and evolution of different system parts
- Comprises one or more layers for the software under development, with each layer having a distinct and specific responsibility
- Typical examples
  - 2-tiered/3-tiered/n-tiered architecture

Depends on how you have packaged the application code!

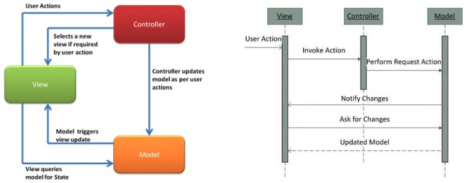


Pipe and Filter

- Divide the application's task into several self-contained data process steps and connect these steps to a data processing pipeline via intermediate data buffers
- Data flows in streams: good for processing of images, audio, video, or batch data
- Good for limited user interaction, like batch processing systems

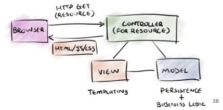


Model view controller

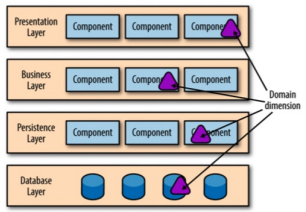


Web MVC

- Two communicating entities in Web applications
  - Server – holds the model
  - Client – interacts with the server (perform requests)
- Controller: responsible for handling user HTTP request, select the model, prepare the view
- View: renders the HTTP response
  - May use a template to render the contents of the model (obtained as HTTP response)
  - E.g., in ASP.net, views are defined in ASP files, which are HTML templates with server-side scripts that can generate dynamic content
- Model: business logic + persistence
  - Can use a database directly (e.g., MySQL) or via ORM
  - Needs to manage concurrent modifications



Layered architecture vs Microservices architecture



In a layered architecture, the focus is on the *technical* dimension, or how the mechanics of the application work: persistence, UI, business rules, etc. Most software architectures focus primarily on these technical dimensions.

However, an additional perspective exists. Suppose that one of the key domain concept (or business capability or a bounded context) in an application is *Checkout*. Where does it live in the layered architecture?

For example, some portion of *Checkout* exists in the UI, another portion lives in the business rules, and persistence is handled by the bottom layers.
Because layered architecture isn't designed to accommodate domain concepts, developers must modify each layer to make changes to domains.

If a software development team is organized into silos resembling their role in the layered architecture, then changes to Checkout require coordination across many teams.

Orchestration Example

Choreography Example

Orchestration :rely on a central brain to guide and drive the process.
Choreography: inform each part of the system of its job, and let it work out the details.

Some Concepts (from tutors)

- Sub-domains is a subset of a domain that is focused on a specific aspect of the problem space.
- Bounded contexts is a software boundary that defines a specific set of concepts, terminology, and business rules that apply within that context while excluding concepts and rules that apply in other contexts. It is part of the solution space.
- Single supporting subdomain and bounded context of **bank account management**
- Note that with this list of requirements, there is no explicit user management domain
  - We can assume user management is handled by an external service as a generic subdomain

Entities:
- Job Opening: Represents a job opportunity posted by a recruiter, including details like job title, description, skills required, etc.
- Job Seeker Profile: Represents the profile of someone looking for a job, including details like skills, experience, location, etc.
- Recruiter/Employer: Represents a business or organization looking to hire employees.
- Employee/Candidate: Represents a potential employee seeking job opportunities.

Aggregates:
- Recruiter Aggregate: This could include the concept of a recruiter and their associated job openings. The recruiter aggregate manages the creation and management of job openings.
  - Without a recruiter, there will not be any job openings
- Job Seeker Aggregate: This includes the concept of a job seeker and their profile. The job seeker aggregate manages the creation and management of job seeker profiles
  - Without a job seeker, there will not be job seeker profiles or job requests

Bounded Contexts:
- Recruitment Bounded Context: This context encompasses all the concepts related to job postings and recruiters. It includes the recruiter aggregate and job openings.
- Job Seeker Bounded Context: This context encompasses all the concepts related to job seekers and their profiles.
- Search Bounded Context: This context is responsible for handling search functionality for both employers and job seekers. It includes searching for candidates based on criteria and searching for jobs based on criteria.

Microservices:
- Recruitment Service: Responsible for managing job openings and interactions with recruiters. This microservice handles creating, updating, and listing job openings.
- Job Seeker Service: Manages job seeker profiles and interactions with job seekers. It includes creating and managing profiles.
- Search Service: Provides search functionality for both employers and job seekers. It handles searching for candidates based on various criteria and searching for jobs based on criteria.
- Communication Service (Microservice): Manages the communication between employers and job seekers via email when a match is found.

Microservice Scaling Cube

X-axis scaling

X-axis scaling consists of running multiple copies of an application behind a load balancer. If there are N copies then each copy handles 1/N of the load. This is a simple, commonly used approach of scaling an application.
One drawback of this approach is that because each copy potentially accesses all of the data, caches require more memory to be effective. Another problem with this approach is that it does not tackle the problems of increasing development and application complexity.

Y-axis scaling

Unlike X-axis and Z-axis, which consist of running multiple, identical copies of the application, Y-axis axis scaling splits the application into multiple, different services. Each service is responsible for one or more closely related functions.
There are a couple of different ways of decomposing the application into services. One approach is to use verb-based decomposition and define services that implement a single use case such as checkout. The other option is to decompose the application by noun and create services responsible for all operations related to a particular entity such as customer management. An application might use a combination of verb-based and noun-based decomposition.

Z-axis scaling

When using Z-axis scaling each server runs an identical copy of the code. In this respect, it's similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server. One commonly used routing criteria is an attribute of the request such as the primary key of the entity being accessed. Another common routing criteria is the customer type. For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity. Data partition across servers.

External vs Internal Quality Attributes

Internal

- Not directly observed when software is executing
- Perceived by the developers and maintainers
- Encompasses aspects of design that may impact external attributes

Availability	Performance	Efficiency	Scalability	Robustness
Safety	Security	Reliability	Integrity	Verifiability
Deployability	Compatibility	Installability	Portability	Maintainability
Usability	Testability	Modifiability	Reusability	Interoperability

Synchronous communication :

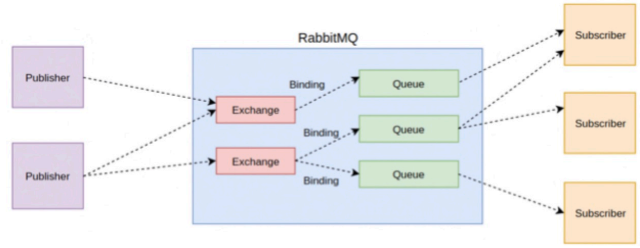
- the caller sends a message and waits for the receiver to respond eg in login action in which the caller must have a reply.
- sender sends a request to the receiver and receiver sends a response

e.g. HTTP/S protocol for synchronous communication

Asynchronous communication :

- a communication that has a lag between when a message is sent and when the receiver receives/interprets/responds to it.
- the caller skips the wait and continues executing whatever code is necessary.
- It enables independent functioning of sender and receiver
- It also enables one-to-many communication where sender can send request to multiple receivers at once.

e.g. AMQP for asynchronous communication.
AMQP: Advanced Message Queue Protocol



Creational Patterns

- Creational patterns help designers handling issues with creation of objects.
- e.g. Factory pattern

Structural Patterns

- Structural Patterns are used to provide a structure to the relationship between objects.
- e.g. Façade pattern

Behavioral Patterns

- Behavioral Patterns help define how objects interact with each other to deliver a task.
- e.g. Observer pattern

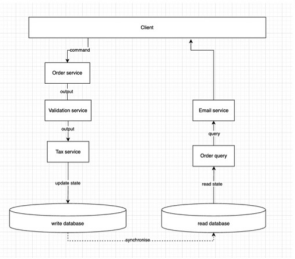
RabbitMQ: *RabbitMQ is a message broker that implements the Advanced Message Queuing Protocol (AMQP). It excels at asynchronous message communication and supports various messaging patterns.*
• RabbitMQ is known for its reliability and message durability, making it a great choice for applications where message persistence (including disk storage) is crucial.
ActiveMQ*: ActiveMQ is also a message broker, but it implements the Java Message Service (JMS) API. It is well-suited for Java applications.*
• ActiveMQ provides support for both point-to-point and publish-subscribe messaging patterns.
• ActiveMQ also supports message persistence to disk, which ensures that messages are not lost even in the event of a system failure.
Redis*: Redis is primarily an in-memory data store, but it can also be used for message queuing via its pub-sub (publish-subscribe) mechanism.*
• While Redis is fast and lightweight, it may not be as durable as RabbitMQ or ActiveMQ since it relies on memory for message storage.
• Since Redis is primarily relies on in-memory storage, it makes it extremely fast but less durable. It can be configured to periodically save data to disk, but this introduces latency and potential data loss in case of a crash.
• Redis is simple and easy to use due

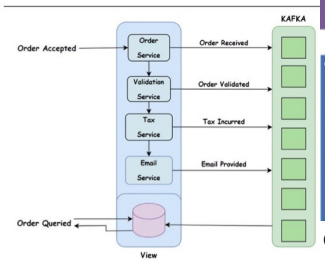
Feature	RabbitMQ	ActiveMQ	Redis
Messaging Patterns	**Point-to-point**, Publish/Subscribe, **Request/Reply**	**Point-to-point**, Publish/Subscribe, **Request/Reply**	Publish/Subscribe
Message Durability	Supports **persistent** and non-persistent messages	Supports both **persistent** and non-persistent messages	Not persistent by default, but can be configured
Clustering	Yes	Yes	Yes
High Availability	Yes	Yes	Yes
Message Acknowledgment	**Supported**	**Supported**	Not applicable (messages are received immediately)
Message Routing	Routing based on message attributes	Flexible routing options, including virtual topics	Messages sent to specific channels/subscribers
Supported Protocols	AMQP, MQTT, STOMP, HTTP, and more	JMS, MQTT, STOMP, AMQP, OpenWire	Pub/Sub via Redis protocol
Language Support	Multiple client libraries for various languages	Primarily Java clients, but supports other languages	Multiple client libraries for various languages
Performance	High throughput and low latency	High performance, optimized for Java applications	**Extremely** fast due to in-memory processing
Scalability	Highly scalable, supports large-scale deployments	Scalable, suitable for both small and large setups	Scalable, suitable for real-time applications
Speed	~10k msg/sec	~10k msg/sec	~10m msg/sec

Messaging Patterns
Message Construction. *A message contains:*
• Header: Contains metadata. Can contain message intent command message, which tells the receiver what to do; document message, which sends data but does not really tell the receiver what to do; event message, which simply notifies the receiver about a change.
• Properties: Optional. For message selection and filtering. Three kinds — application-related, providerrelated and standard properties.
• Payload: Body, data structures.
Message Channels. *Connect collaborating senders and receivers.*
A channel transmits one way. Two-way messages need two channels. Some concepts:
• Return Address: Request contains an address to tell the replier where to send reply to.
• Correlation ID: Specifies which request the reply is for. Can be chained.
• Message Sequence: Basically break the message down into smaller segments.
• Point to Point (P2P): Request processed by single consumer.
• Publish-Subscribe Channel: Request broadcasted to all interested parties via topics.
• Invalid Messages: Queue to move a message to when it cannot be interpreted.
• Dead Letter: Queue to move a message to when it cannot be delivered.
• Datatype Channel: Separate channel for each type of data, e.g. XML, byte array, etc.
Message Routing. *Consumes messages from one channel and reinserts them into different channels based on condition Some concepts:*
• Content-Based: Examines message content and routes. Message filter is a special kind of content-based router that discards messages based on content.
• Context-Based: Decides destination based on context, e.g. load-balancing.
• Message Splitter: Splits a single message into multiple.
• Message Aggregator: Aggregates correlated messages into a single message.
• Scatter-Gather: Broadcast to multiple participants and aggregates replies into a single message.
Message Transformation. *Transform application-layer data structures, data types of fields, data representations e.g. ASCII to Unicode, transport protocol, e.g. TCP/IP to sockets. Also called Message Translators or Channel Adapters. A Canonical Data Model may also be used, which is an "adapter" superset model.*
Message Endpoints. *Interface between app and messaging system. Channel-specific, one instance handles either send or receive. Can be synchronous, i.e. polling consumer, or can be asynchronous, i.e. event-driven receiver.*

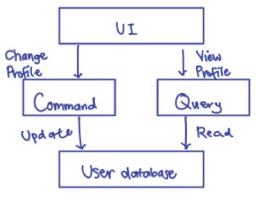
PART 1: Q1. same as Tutorial 6 Question 4: Event-driven service communication and CQRS:

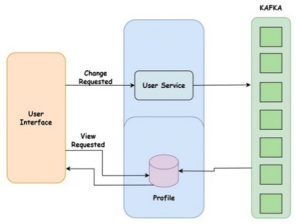
- Illustrate (using a simple diagram) the service communication and use of CQRS in the following scenario:
- An order is accepted in the order service. This is picked up by the validation service, where it is validated. Sales tax is added (Tax service). Information for an email is made available(provided) to an Email service. The updated order goes back to the orders service, where it can be queried via the orders view (Hint: CQRS). After being sent a confirmation email, the user can click through to the order.





Illustrate use of CQRS pattern for the PeerPrep User Service for use cases of the user making a change in their profile at the UI and the user viewing their profile in the UI.





Persistent Asynchronous

- Sender keeps executing without blocking.
- The message may take an arbitrary amount of time to reach the receiver.
- Sender may or may not be running by the time the message reaches receiver.
- Disks or multiple memory queues could be used for persistence at receiver's side.
- There is a guarantee that the message will eventually reach the receiver.

Persistent Synchronous

- Sender is blocked until an ack for receipt(not for delivery/response) is received.
- The message persists -stays in receiver's queue (or in any router along the way) for an arbitrary amount of time.

Transient Asynchronous

- Sender continues execution (nonblocking) after sending a message.
- Receiver has to be running, because if it is not running the message will be discarded.
- Even if any router along the way is down, the message will be discarded.

Transient Synchronous

- Receipt-based
  - Sender blocks (synchronous) until an ack is received. This ack is simply a receipt. It does not tell us anything about whether the process has started on receiver end.
- Delivery-based
  - Sender will block until receiver takes the delivery of the message.
  - The ack comes a little bit later than that in receipt-based.
  - Essentially an asynchronous RPC
- Response-based
  - Sender blocks until it receives a response.
  - This is traditional RPC. The client is blocked for the entire duration the reply comes

General design principles --
SoC, SRP, High Cohesion, Loose Coupling, Abstraction, Encapsulation, Information Hiding, Interface Segregation principle(ISP), Dependency Inversion, Open-close principle, Design for reuse
Principles of microservice design - Design with ubiquitous language, loose coupling, domain specific requirements/ around business capabilities, small team structures, containerised deployments, scaling components at different pace, decentarlized development and deployment
Principles in Event driven and messaging systems - asynchronous communication, decoupling(or loose coupling)

Design principles provide high level guidelines to design better software applications.
- They do not provide implementation guidelines and are not bound to any programming language.
-Modularity and the SOLID (SRP, OCP, LSP, ISP, DIP) principles are general set of design principles. There are specific principles for specific design style eg for Microservices or Event driven design approach.

Design Pattern provides low-level solutions related to implementation, of commonly occurring problems.
- Design pattern suggests a specific implementation for a specific problem.
- Eg. Creating a class that can only have 1 object at a time => singleton design pattern