Software Architecture

# Definition: Software architecture is the high-level design and structure of a software system, specifying how its components interact and meet requirements. It considers non-functional aspects like performance and scalability and guides development decisions.

“Software architecture is the building plan of software that describes structure, behavior and guiding principles such as: cross-cutting mechanisms, programming conventions, build and development practices, concurrency mechanisms”

# Good software architecture:

These principles describe key aspects of a good software architecture:

1. **High Cohesion:** Components should have well-defined and focused responsibilities, simplifying understanding and maintenance.
2. **Loose Coupling:** Components should have minimal dependencies on each other's internal details, promoting flexibility and easy modification.
3. **Domain-Driven:** Components should align with the problem domain, making the software more intuitive and maintainable.
4. **No Unnecessary Components:** Each component should have a clear purpose, and no component should be superfluous, reducing complexity.

# **Kruchten's 4+1 Architectural View:**

Kruchten's 4+1 Architectural View Model is a framework for defining and documenting the architecture of a software-intensive system. It was developed by Philippe Kruchten and is often used in software engineering to provide multiple perspectives on a system's architecture. The "+1" refers to an additional view that complements the four primary views.

Here's an overview of each view:

1. **Logical View:**
    - The logical view focuses on the system's functionality from the perspective of users, developers, and other stakeholders.
    - It describes the high-level, abstract structure of the system in terms of modules, components, classes, and their relationships.
    - Use case diagrams, class diagrams, and entity-relationship diagrams are common artifacts used in the logical view.
2. **Process View:**
    - The process view emphasizes the dynamic aspects of the system, including the system's behavior, processes, and interactions.
    - It describes how different components or modules collaborate and communicate to achieve the system's functionality.

- Tools like sequence diagrams and activity diagrams are often used to represent process views.

3. **Physical View:**

- The physical view provides insight into the system's deployment and distribution across hardware and network infrastructure.
- It describes how software components are mapped to physical resources, including servers, databases, and communication channels.
- Deployment diagrams and network topologies are typical artifacts in the physical view.

4. **Development View:**

- The development view focuses on the organization and structure of the development process itself.
- It includes details on how the software is built, divided into modules or components, and the dependencies between them.
- Component diagrams, package diagrams, and build scripts are used to represent the development view.

5. **Scenarios (Use Cases) View (the "+1"):**

- The scenarios or use cases view complements the other four views by providing concrete, real-world examples of how the system functions.
- It typically includes detailed use case descriptions, scenario diagrams, and sequence diagrams to illustrate specific interactions and workflows.
- This view helps stakeholders understand how the system will be used in practical terms.

Kruchten's 4+1 View Model is valuable because it offers a comprehensive way to describe and communicate the architecture of a software system from various perspectives, catering to the needs of different stakeholders, including developers, architects, project managers, and end-users. This multidimensional approach enhances the overall understanding of the system and aids in its design, development, and maintenance.

**** Explore some examples with proper visualization****

# Software Architecture Design:

Software architecture design is the process of planning and defining the structure, components, and interactions of a software system to meet specific requirements. Key steps include identifying stakeholders, decomposing the system, designing components, addressing scalability and security, and creating documentation. It forms the blueprint for successful software development.

"Software architecture design maps the problem domain to the solution domain considering forces and risks"

In software architecture design, "forces" are factors or constraints that influence decisions. Key forces include:

1. **Functional Requirements:** Desired system features.
2. **Non-Functional Requirements:** Performance, security, etc.
3. **Budget/Resources:** Cost and available assets.
4. **Technology Stack:** Existing tools and platforms.
5. **Regulatory Compliance:** Legal standards.
6. **Scalability:** Handling growth in users/data.
7. **Security:** Protecting against threats.
8. **Usability:** User-friendly design.
9. **Integration:** Interfacing with external systems.
10. **Time-to-Market:** Speed of development.
11. **Maintainability:** Ease of future updates.
12. **Risk Management:** Mitigating uncertainties.
13. **Costs:** Development and operational expenses.
14. **Team Expertise:** Skills of development team.
15. **Environmental Factors:** Deployment conditions.
16. **Competitive Landscape:** Gaining an edge.

# Iterative vs Incremental Development:

Iterative Development: Builds the entire system incrementally in repeated cycles, focusing on feedback and adaptability. This is a sequence of activities performed within a period of time (4-8 weeks), typically from requirements to testing.

Incremental Development: Expands the system's functionality in discrete, sequential increments, delivering usable features early. This is a stable executable and testable software release that makes one step forward.

Comparison:

- **Feedback vs. Structured Planning:** Iterative development relies on feedback and adapts to changes, while incremental development follows a predefined plan.
- **Risk Management vs. Predictability:** Iterative development is often used for risk mitigation and adapts to uncertainty, while incremental development offers more predictability and structure.
- **Value Delivery:** Both approaches aim to deliver value early, but incremental development provides usable features incrementally, whereas iterative development refines the entire system increment by increment.
- **Flexibility vs. Stability:** Iterative development is highly flexible and adaptable to changing requirements, while incremental development is suitable for projects with relatively stable requirements.
- **Deployment Frequency:** Iterative development may lead to more frequent deployments of smaller changes, while incremental development results in less frequent but more substantial releases.

Iterative Development (Agail) vs. Waterfall:

1. Approach:

- **Iterative Development:** Divides the project into small, repeated cycles (iterations) with feedback loops.
- **Waterfall:** Follows a sequential, linear approach with distinct phases.

2. Feedback:

- **Iterative Development:** Emphasizes continuous feedback and adaptation throughout the project.
- **Waterfall:** Provides limited feedback, primarily at the end of the project.

3. Flexibility:

- **Iterative Development:** Highly adaptable to changing requirements and evolving project needs.
- **Waterfall:** Best suited for projects with stable and well-understood requirements.

4. Risk Management:

- **Iterative Development:** Identifies and mitigates risks incrementally, with a focus on early risk reduction.
- **Waterfall:** Addresses risks mostly in the early phases of the project.

5. Deployment:

- **Iterative Development:** May lead to more frequent deployment of smaller changes or increments.
- **Waterfall:** Results in less frequent but more substantial releases.

6. Suitability:

- **Iterative Development:** Suitable for projects with evolving or unclear specifications and a need for adaptability.
- **Waterfall:** Appropriate for projects with stable and well-understood requirements.

7. Project Phases:

- **Iterative Development:** Phases overlap, and each iteration goes through the full development cycle.
- **Waterfall:** Phases are distinct, and one phase is completed before the next begins.

8. Cost and Time Estimation:

- **Iterative Development:** May require more effort in terms of planning and monitoring due to ongoing iterations.
- **Waterfall:** Provides a structured plan from the start, making cost and time estimation more predictable.

9. Client Involvement:

- **Iterative Development:** Clients and stakeholders are actively involved throughout the project, providing continuous input.
- **Waterfall:** Client involvement is more concentrated at the beginning and end of the project.

10. Adaptation to Change: - **Iterative Development:** Easily accommodates changes in requirements and priorities. - **Waterfall:** Changes can be challenging and costly to implement once the project has progressed past the requirements phase.

- Thinking in Services -

# What is Service: *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*

So, in modern terms, a web service is like a helpful internet-based assistant that follows a menu of instructions and understands your requests, making it easy for different computer systems to communicate and share data seamlessly.

For example, imagine you want to know the weather forecast for your location. You send a message to this digital assistant, and it sends back the weather information in a way that your computer or app can easily read. This happens using the language of the internet (HTTP), and the information is neatly organized (in a format called XML) so that computers can process it effortlessly.

# Important "parts" of a service:

Service Implementation:

- **What it is:** This is the core component of the service, consisting of the actual software code that performs the service's business logic and handles remote communication.
- **Explanation:** The service implementation is where the functionality of the service resides. It contains the code that carries out the specific tasks or operations that the service is designed to perform. This code can include data processing, calculations, database access, and any other necessary logic. Additionally, it manages how the service communicates with other systems or clients over a network.

2. **Service Port:**

- **What it is:** The service port represents the network address or location where the service can be accessed.
- **Explanation:** Think of the service port as the entry point or contact information for the service. It's like the street address of a physical store. Other systems or clients use this network address to reach and communicate with the service. The service port specifies where the service is hosted and how external entities can connect to it over the network.

3. **Service Interface:**

- **What it is:** The service interface defines the set of messages or requests that can be exchanged with the service and any associated constraints or rules.
- **Explanation:** The service interface acts as a contract that specifies how other systems or clients can interact with the service. It outlines the types of requests or commands that can be sent to the service and the format and structure of these messages. It also defines any restrictions or requirements for using the service, such as authentication or input validation.

4. **Interface Description (Optional):**

- **What it is:** Interface description refers to a potentially machine-processible description of the service interface. It's not always required but can be helpful for automation and compatibility.
- **Explanation:** An interface description provides additional information about the service interface in a format that computers can understand. This description can be written in a standardized format like WSDL (Web Services Description Language) for web services. It aids in automating interactions with the service and ensures that other systems can understand and correctly communicate with the service without relying solely on human interpretation.

In summary, these "parts" of a service work together to define, implement, and facilitate the functionality and accessibility of the service. The service implementation contains the actual code, the service port serves as the network address, the service interface defines how interactions occur, and an optional interface description can enhance automation and interoperability. Together, they enable effective communication and utilization of the service within a software system or across different systems.

# So, is a Web Service an Object?

No, a web service is not an object. A web service is a standalone software component that provides specific functionality over the internet or a network, facilitating inter-machine communication, whereas objects are instances of classes used within a single program to model and organize code. They serve different purposes in software development.

# Is Web Services Programming = Java Programming?

No, web services programming is not the same as Java programming. Web services programming is a broader concept that includes various programming languages for creating and using web services. Java programming is a specific type of programming that uses the Java language for various software development tasks, including web services but not limited to them.

# What is Service Oriented Architecture (SOA):

SOA is a paradigm or architectural approach for structuring and utilizing distributed capabilities or services. These services can be owned and operated by different entities or domains, and SOA provides a framework for making them work together seamlessly to support the needs of a larger software system or organization. SOA promotes modularity, reusability, and interoperability by breaking down complex systems into smaller, independent services that communicate with each other.

# SOA implications:

1. **Contract Definition:**
   - Contracts specify how services can be accessed, what data they require, and what they return. Well-defined contracts enable interoperability and smooth communication between services.
2. **Service Granularity:**
   - The granularity of services decides whether services should be fine-grained (providing small, specific functions) or coarse-grained (providing larger, more comprehensive capabilities) impacts how services are designed, managed, and reused.
3. **Dependency Management:**
   - Organizations using SOA must manage dependencies between services. Understanding how services depend on each other helps in ensuring smooth operations and preventing disruptions when changes are made to one service.
4. **Risk Management:**
   - SOA introduces new challenges and risks, including security vulnerabilities, compatibility issues, and performance bottlenecks. Proper risk management strategies are crucial to identify, assess, and mitigate these risks effectively.
5. **Implementation Challenges:**
   - Implementing SOA can be complex and may involve challenges such as selecting appropriate technologies, designing effective service contracts, ensuring security, and managing the service lifecycle. Organizations need to be prepared for these implementation challenges.
6. **Common Delivery Platform:**
   - SOA often involves the creation of a common platform or infrastructure that supports service development and execution. This shared platform helps standardize service deployment, monitoring, and management.
7. **SOA Governance:**
   - SOA governance is a critical aspect of managing an SOA implementation. It involves defining and enforcing policies, standards, and best practices for service development, deployment, and usage. Governance ensures that services align with organizational goals and maintain quality.

In short:

1. **Contract Definition:** Clearly define how services can be used.
2. **Service Granularity:** Decide on the size and scope of services.
3. **Dependency Management:** Handle inter-service dependencies.
4. **Risk Management:** Address security and operational risks.
5. **Implementation Challenges:** Prepare for complexities in SOA adoption.
6. **Common Delivery Platform:** Create a standardized service platform.
7. **SOA Governance:** Enforce policies and standards effectively.

#SOA Creation Rules:

-->**Rules on Component Cut:**

1. **Domain-driven components (R1):** Components should mirror real-world business domains.
2. **One service category per component (R2):** Each component serves a specific service category.
3. **Dependencies follow service categories (R3):** Components should relate based on service categories.
4. **No cyclic dependencies (R4):** Avoid circular dependencies between components.
5. **High Cohesion, Loose coupling (R5):** Components should work together closely but not be tightly bound.
6. **Data encapsulation (R6):** Data should be hidden within components, accessed through interfaces.

-->Rules on Service Design:

7. **Technology neutral (R7):** Design services without being tied to specific technologies.
8. **No references (R8):** Avoid direct references between services.
9. **Normal, complete, and no redundancies (R9):** Design data structures efficiently.
10. **Coarse granularity (R10):** Services should have broader functions.
11. **Idempotent (R11):** Repeated operations yield the same result. In other words, always return same results to prevent operation which operation is already executed.
12. **Context-free (R12):** Services work independently of specific contexts.

-->Rules on Coupling:

13. **Coupling mechanisms follow domain coupling (R13):** How components connect should match real-world relationships.
14. **Transaction control follows domain coupling (R14):** Transactions align with domain relationships.
15. **Data types (R15):** Use consistent data types for compatibility and data integrity.

Basic Service Implementation

# HTTP (Hypertext Transfer Protocol): This is the most widespread protocol for accessing services. It is an application layer protocol.

** Important characteristics
>> **Synchronous:** Call blocks until response.
>> **Stateless:** Two subsequent requests are separate transactions (yet, cookies possible).
>> **Content-agnostic:** You can transfer almost any content via HTTP.

# Important Request Types:

Here are the most common HTTP request methods:

1. **GET:** Used to request data from a specified resource. GET requests should only retrieve data and should not cause any side effects on the server. It is the most commonly used method for fetching web pages and resources.
2. **POST:** Used to send data to be processed to a specified resource. It is often used for submitting form data to a server, such as when submitting a web form.
3. **PUT:** Used to update a resource or create a new resource if it does not exist at the specified URL. It replaces the existing resource with the new one sent in the request.
4. **DELETE:** Used to request the removal of a resource at a specified URL. It is used to delete the resource indicated by the URL.

# Important HTTP status codes in short:

1xx (Informational):

- **100 Continue:** The server has received the initial part of the request, and the client can proceed with sending the rest of the request.

- **2xx (Successful):** Request was successful.
    - 200 OK: Successful request.
    - 201 Created: New resource created.
    - 204 No Content: Request successful, no data to return.
- **3xx (Redirection):** Resource has moved.
    - 301 Moved Permanently: Permanent move.
    - 302 Found: Temporary move.
    - 304 Not Modified: Cached copy still valid.
- **4xx (Client Errors):** Client-side issues.
    - 400 Bad Request: Syntax error.
    - 401 Unauthorized: Authentication required.
    - 403 Forbidden: Access denied.

    - 404 Not Found: Resource not found.
- **5xx (Server Errors):** Server-side issues.
    - 500 Internal Server Error: Server problem.
    - 501 Not Implemented: Server can't handle request.
    - 503 Service Unavailable: Server temporarily unavailable.
    - 504 Gateway Timeout: Server didn't get a timely response.

# In Java-based JAX-RS (Java API for RESTful Web Services) applications, annotations like **@QueryParam**, **@PathParam**, **@CookieParam**, and **@HeaderParam** are used to extract data from different parts of an HTTP request.

Here's a brief explanation of each of these annotations:

1. **@QueryParam:**
    - This annotation is used to extract data from query parameters in the URL of an HTTP request.
    - Query parameters are typically included in the URL after a "?" character, such as `http://example.com/resource?param1=value1&param2=value2`.
    - Example usage:

```
@GET
public Response getResource(@QueryParam("param1") String param1) {
    // Access the value of "param1" from the query parameter
    return Response.ok("Value of param1: " + param1).build();
}
```

2. **@PathParam:**
    - This annotation is used to extract data from path parameters in the URL of an HTTP request.
    - Path parameters are parts of the URL path enclosed in curly braces `{}`, such as `http://example.com/resource/{id}`.
    - Example usage:

```
@GET
@Path("/{id}")
public Response getResource(@PathParam("id") String id) {
    // Access the value of "id" from the path parameter
    return Response.ok("Value of id: " + id).build();
}
```

3. **@CookieParam:**
    - This annotation is used to extract data from cookies included in the HTTP request.
    - Cookies are often used for maintaining session data or storing client-specific information.
    - Example usage:

```
@GET
```

```
public Response getResource(@CookieParam("sessionToken") String
sessionToken) {
    // Access the value of the "sessionToken" cookie
    return Response.ok("Session Token: " + sessionToken).build();
}
```

4. **@HeaderParam:**

- This annotation is used to extract data from HTTP headers included in the request.
- HTTP headers carry metadata about the request, such as "Authorization," "Content-Type," or custom headers.
- Example usage:

```
@GET
public Response getResource(@HeaderParam("Authorization") String
authorizationHeader) {
    // Access the value of the "Authorization" header
    return Response.ok("Authorization Header: " +
authorizationHeader).build();
}
```

Note: Review all the important code from slide-04.

Describing Services

# Definition: A service description is a structured document or metadata that provides comprehensive information about a particular service or API, including its functionality, endpoints, data formats, authentication methods, and usage guidelines. It serves as documentation for developers, clients, and users to understand and interact effectively with the service.

# Why Service description: Service descriptions are needed to provide clear, standardized information about a service or API, making it easier for developers to understand, integrate with, and use the service effectively while promoting interoperability and reducing ambiguity.

# Use cases:

1. Dedicated API:

♣ Describe a dedicated, well-defined functionality
♣ Need a compact representation

A dedicated API service description, as a form of documentation and metadata for an API, is neither context-free nor idempotent. Because,

1/ Service descriptions are not programming languages or grammars; they are documents or metadata that provide information about an API's usage, endpoints, and behavior. They are context-specific by nature, as they describe how a particular API should be used.

2/ Service descriptions are not requests or operations; they are static documents or metadata that do not perform actions themselves. Therefore, the concept of idempotent does not apply to service descriptions.

2. **Business document exchange**
    - ♣ Describe a legally binding interaction between companies
    - ♣ Need a standardized format that can be adapted to concrete companies

→ **Is it actually a service:** A "Business Document Exchange" is typically not a standalone service itself but rather a process or capability within a broader business or organizational context. It refers to the exchange of various types of documents, such as invoices, purchase orders, contracts, and reports, between different entities, such as businesses, partners, suppliers, or customers.

3. **Message hub / Crosscutting service**

♣ Describe recurring service integration tasks
♣ Need a very precise format and interaction description

# OpenAPI:

"The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection."

In other word, OpenAPI is a standardized way to describe and document RESTful APIs, making it easier for developers to understand, use, and automate interactions with APIs. It's like a blueprint for APIs.

Certainly, here are some answers to the questions about OpenAPI:

1. **What is OpenAPI, and what is its primary purpose?**
    - OpenAPI is a standard format for describing RESTful APIs, and its primary purpose is to provide a structured and machine-readable way to document and define the behavior of APIs.
2. **What are the key components of an OpenAPI specification document?**
    - Key components include `info` (metadata), `paths` (endpoints and operations), `components` (reusable schemas and parameters), and more.
3. **Explain the difference between JSON and YAML as formats for OpenAPI specifications.**
    - JSON and YAML are two formats for writing OpenAPI specs. JSON is more concise, while YAML is more human-readable and allows comments.
4. **Why is OpenAPI considered important in API development and documentation?**
    - OpenAPI enhances collaboration, automation, and consistency in API design and documentation, making it easier for developers to work with and understand APIs.
5. **How does OpenAPI help improve the developer experience when working with APIs?**
    - OpenAPI provides clear documentation, code generation, and testing tools, simplifying API consumption and reducing integration challenges.

6. **What is the relationship between OpenAPI and the Swagger UI?**
   - Swagger UI is an interactive web interface that allows developers to explore and test APIs described by OpenAPI specifications.
7. **Can you provide an example of a simple OpenAPI specification, and explain its sections?**
   - Here's an example: `info` (metadata), `paths` (endpoints and operations), `components` (schemas and parameters).
8. **How does OpenAPI support versioning of APIs?**
   - OpenAPI allows for versioning within the specification, ensuring that changes and updates to APIs can be managed while maintaining backward compatibility.
9. **What role does code generation play in the context of OpenAPI?**
   - Code generation automatically creates client and server code based on OpenAPI specs, reducing manual coding efforts.
10. **How does OpenAPI assist in ensuring API consistency and interoperability?**
    - OpenAPI enforces standardized API descriptions, promoting consistency and making it easier to integrate APIs from different sources.

# JSON Schema:

JSON Schema is a vocabulary that allows you to annotate and validate JSON documents. It provides a structured way to describe the expected format and constraints of data in a JSON document, which is particularly useful for ensuring data consistency and validation in various applications.

Here's a brief explanation:

- **Annotation:** JSON Schema allows you to define the structure of JSON data by specifying the expected properties, their data types, and additional constraints. This serves as an annotation or metadata for JSON documents, describing how the data should be structured.
- **Validation:** JSON Schema also serves as a validation tool. It enables you to check if a given JSON document complies with the defined schema. This means you can verify if the data adheres to the expected format, data types, and constraints.
- **Examples:** Here's a simple JSON Schema example:

```
jsonCopy code
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "integer",
      "minimum": 0
    }
  },
  "required": ["name"]
}
```

In this example, the schema defines that the JSON data should be an object with a "name" property (which must be a string) and an optional "age" property (which must be a non-negative integer).

REST and RESTful Services

# What is REST?

REST, short for representational state transfer, is a type of software architecture that was designed to ensure interoperability between different Internet computer systems.

REST (Representational State Transfer) is an architectural style for designing networked applications that emphasizes simplicity, statelessness, and the use of standard HTTP methods such as GET, POST, PUT, and DELETE, for interacting with resources identified by unique URIs. It's widely used for building web services and APIs.

# REST Principles:

1. "Addressable Resources" in REST means that every resource, like a book in a library, should have a unique web address (URI). Clients use these URIs to access and interact with specific resources, creating a consistent and structured way to navigate and manipulate data.

Example: protocol://host:port/path?queryStr# anchor
♣ **protocol**: Names the protocol such as HTTP/S, FTP, …
♣ **host**: IP address or DNS name
♣ **port**: well, the port, default values depend on the protocol
♣ **path**: "/"-delimited series of path elements
♣ **queryStr**: parameters as "&"-delimited series of name=value pairs
♣ **anchor**: reference in the respective resource

2. Representation orientation: In REST, components (like clients and servers) interact with resources by using a representation. This representation is like a snapshot of the resource's current or desired state. It's a package that contains the actual data (sequence of bytes) and additional information about that data (metadata). These representations are sent back and forth between components to perform actions on resources, enabling communication and manipulation in a standardized way.

¯ **Metadata means**
♣ Control data determines the purpose of a message, e.g., an HTTP method (GET, POST, etc.) or a parameter
◊ NOT really specific to a user application
♣ Data format is defined by a media type, e.g., a mime type

◊ NOT really specific to a user application

- **This is different from WSDL services where you identify actions by means of portType and operation!**

Example: Scenario: Imagine a RESTful online bookstore.

1. **Resource:** A book titled "The Great Gatsby."
2. **Representation:** The bookstore wants to provide information about this book to a client in JSON format. So, it creates a JSON representation of the book's details:

   ```
   {
     "title": "The Great Gatsby",
     "author": "F. Scott Fitzgerald",
     "genre": "Fiction",
     "year_published": 1925
   }
   ```

   Here, the JSON representation encapsulates the book's information (current state) and is structured in a well-defined format.
3. **Transfer:** When a client (e.g., a web browser) requests information about "The Great Gatsby," the server transfers this JSON representation to the client over HTTP. The client receives the representation.

3. Self-Descriptive Messages: REST constrains messages between components to be self-descriptive in order to support intermediate processing of interactions.

Overall, the "Self-Descriptive Messages" principle ensures that RESTful systems are easy to comprehend, allowing clients to interact with resources effectively and independently, even in distributed and evolving environments.

Example: Consider a RESTful API for a weather service. When a client requests weather data for a specific location, the server provides a self-descriptive response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "location": "New York",
  "temperature": 72°F,
  "conditions": "Partly cloudy",
  "humidity": 50%
}
```

In this example:

- The HTTP status code "200 OK" indicates a successful response.
- The "Content-Type" header specifies that the response body is in JSON format.

- The JSON representation is self-contained, with clear labels for each piece of data, making it easy for the client to understand the weather conditions in New York without needing external information.

4. Stateless Communication: All REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it.

In simple terms, in a RESTful system, every interaction is like starting with a clean slate. Each time you make a request, you provide all the information needed for the server to understand that request. It doesn't remember anything about previous requests you made.

Examples: Consider a RESTful e-commerce website where a client wants to add a product to the shopping cart:

- **Stateless Request:** The client sends a stateless request to the server to add a specific product to the cart. The request includes all necessary information, such as the product ID, quantity, and authentication token.

```
POST /cart/add
Host: example.com
Content-Type: application/json
Authorization: Bearer [token]

{
  "product_id": "12345",
  "quantity": 2
}
```

- **Stateless Response:** The server processes the request based solely on the information provided in the request. It responds with a confirmation or an error message.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "message": "Product added to cart successfully."
}
```

In this example:

- The server does not rely on any prior knowledge of the client's shopping history or session. It processes the request based on the provided data and authorization token.
- The client sends a complete request with all the information needed for the server to understand and fulfill the request.

5. HATEOAS (Hypermedia as the engine of Application state):

HATEOAS, which stands for "Hypermedia as the Engine of Application State," is a principle and architectural style often associated with RESTful web services. It defines how clients can interact with a web service by using hypermedia links provided in the responses.

In simpler terms, HATEOAS enables a self-descriptive and discoverable API by embedding links to related resources and actions directly within the API responses.

Let's break down HATEOAS (Hypermedia as the Engine of Application State) with a straightforward example:

Imagine a Web-Based Store:

1. You enter an online store's website to buy products.
2. On the store's homepage, you see a list of categories: "Electronics," "Clothing," and "Books."
3. You click on the "Electronics" category.
4. In the "Electronics" section, you find a list of products like "Smartphones," "Laptops," and "Headphones."
5. You click on "Smartphones."

Now, here's where HATEOAS comes into play:

HATEOAS in Action:

In a system following HATEOAS principles:

1. When you clicked on "Smartphones," the server not only sent you a list of smartphones but also included hyperlinks or buttons for related actions:

   - **"Add to Cart"** - You can click this to add a smartphone to your shopping cart.
   - **"See Details"** - This link leads to more information about a specific smartphone.
   - **"Explore Other Categories"** - This takes you back to the list of categories (e.g., "Electronics," "Clothing").

2. If you click "Add to Cart," the server provides new links, such as **"Proceed to Checkout"** or **"Continue Shopping."** If you click "Proceed to Checkout," it guides you through the checkout process.
3. At each step, the server includes links to possible actions, helping you navigate and interact with the store without needing to know specific URLs or steps in advance.

In essence, HATEOAS turns your interactions with the online store into a guided and self-explanatory experience. You're given links to the next steps at each stage, making it easy to explore the store, add items to your cart, and complete your purchase without having to memorize or guess URLs. It's like a store where each aisle has signs pointing to where you can go next, ensuring a smooth shopping experience.

# Does HTTP Overlook the REST:

No, HTTP is the protocol for data transfer, while REST provides principles for designing networked applications, often using HTTP as its communication medium.

Key Differences:

- **Scope:** HTTP is a specific protocol for data transfer, while REST is a broader architectural style for designing networked applications.
- **Focus:** HTTP focuses on the technical aspects of communication, such as request methods and status codes. REST focuses on how resources are organized, identified, and manipulated.
- **Applicability:** HTTP is used in various applications beyond the web, including APIs and data transfer. REST is specifically used for designing web services and APIs.
- **State:** HTTP itself is stateless, while REST promotes statelessness as a design principle.

Linux Containers

# What is Linux container?
Linux Containers, often referred to as LXC, are a lightweight form of virtualization technology that enable the creation and management of isolated, self-contained environments within a single Linux host operating system. These containers package applications and their dependencies, providing a consistent and efficient runtime environment while sharing the host's kernel.

# Why Linux container?
Linux containers are like digital packages for software. They help keep different programs separate and make it easy to move and manage them, saving time and resources.

In other words, it runs many applications in a fast, portable and isolated way in many different environments!

# Hypervisors vs Linux Containers

Hypervisors:

- **Virtualization:** Hypervisors create and manage virtual machines (VMs), each with its own full operating system.
- **Resource Allocation:** VMs have fixed resource allocations, which can lead to resource inefficiency if not fully utilized.
- **Isolation:** Strong isolation between VMs, as each runs its own kernel and has its own complete OS.
- **Overhead:** Higher overhead due to running multiple OS instances, which can limit scalability and performance.
- **Boot Time:** Longer boot times as each VM needs to start a full OS.
- **Use Cases:** Suitable for scenarios requiring strong isolation and compatibility with different operating systems, often used for server virtualization and running legacy applications.

Linux Containers:

- **Virtualization:** Containers create isolated environments within a single Linux host OS, sharing the host's kernel.
- **Resource Allocation:** Containers are highly resource-efficient and can dynamically allocate resources based on demand, reducing waste.

- **Isolation:** Containers provide process and resource isolation, with some shared components. Isolation is usually sufficient for most applications.
- **Overhead:** Lower overhead due to sharing the host OS kernel, enabling efficient resource utilization and faster startup times.
- **Boot Time:** Shorter boot times as containers don't require starting a full OS, often starting almost instantly.
- **Use Cases:** Ideal for resource-efficient scenarios, rapid application deployment, and scalability. Commonly used in microservices, DevOps, and cloud-native applications.

Summary:

- Hypervisors offer stronger isolation but come with higher resource overhead and longer boot times.
- Linux Containers prioritize resource efficiency, rapid deployment, and scalability while providing adequate isolation for most use cases.
- The choice depends on specific requirements, with containers being popular in modern application deployment and microservices architectures.

# Near bare-metal runtime performance:

Near bare-metal runtime performance is a characteristic of virtualization and containerization technologies that aim to provide high-performance execution of applications with minimal overhead, making them suitable for demanding and performance-sensitive use cases.

#LXCs are built on modern kernel features:
♣ **cgroups**; limits, prioritization, accounting & control
♣ **namespaces**; process based resource isolation
♣ **chroot**; apparent root FS directory
♣ **Linux Security Modules (LSM);** Mandatory Access Control (MAC)

# Linux cgroups – Core Tool for Resource Isolation

- **Functionality**

♣ Access;- which devices can be used per cgroup
♣ Resource limiting;- memory, CPU, device accessibility, block I/O, etc.
♣ Prioritization;- who gets more of the CPU, memory, etc.
♣ Accounting;- resource usage per cgroup
♣ Control;- freezing & check pointing
♣ Injection;- packet tagging

- **Usage**

♣ cgroup functionality exposed as “resource controllers” (aka “subsystems”)
♣ Subsystems mounted on FS
♣ Top-level subsystem mount is the root cgroup; all procs on host
♣ Directories under top-level mounts created per cgroup
♣ Procs put in tasks file for group assignment
♣ Interface via read / write pseudo files in group

Docker

# Definition: Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. The software that hosts the containers is called Docker Engine. It was first released in 2013.

In formal way, "Docker is a platform for developing, shipping, and running applications in containers. Containers are lightweight, standalone, and executable packages that contain everything needed to run a piece of software, including the code, runtime, system tools, and libraries. Docker simplifies the process of creating, deploying, and managing applications by utilizing containerization technology."

#What does Docker do?

- "Docker allows you to package an application with all of its dependencies into a standardized unit for software development."

- "Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in."

# Containers vs images: Containers and images are fundamental concepts in containerization technologies like Docker. They are closely related but serve distinct purposes. Here's a brief comparison:

Images:

1. **Definition:** An image is a lightweight, standalone, and executable package that contains everything needed to run a piece of software, including the code, runtime, system tools, and libraries. It is a read-only template.
2. **Purpose:** Images serve as blueprints for creating containers. They define the application and its dependencies in a portable and consistent format. Images are typically created during the development or build phase.

Example: Think of an image as a snapshot of a cake recipe. The recipe contains all the instructions and ingredients needed to bake a cake, but it's not a cake itself. You can make copies of the recipe (images) and distribute them to others.

Containers:

1. **Definition:** A container is a runnable instance of an image. It is a lightweight and isolated environment where an application and its dependencies can run.
2. **Purpose:** Containers are used to execute applications in an isolated and reproducible manner. They are created from images and can be started, stopped, and deleted as needed.

Example: Consider a scenario where you have an image of a cake recipe. When you follow the recipe to bake a cake, you're creating a container from the image. You can bake multiple cakes (create multiple containers) from the same recipe (image), each with its unique cake (running application).

Summary:

- **Images** are static, immutable, and serve as blueprints for creating containers. They define what an application consists of and its dependencies.
- **Containers** are running instances of images. They are dynamic, mutable, and provide isolated environments for executing applications.

→ From Lecture:

- **Image:** Imperative description (DOCKERFILE) of a collection of filesystem layers – read-only
- **Container:** Instance of an image – runtime adds a top writable layer

- The major difference between a container and an image is the top writable layer
- You can have many running containers of the same image
- Changes to the top writable layer like writing new files, changing existing ones are ephemeral. (When container exits, changes are gone ◊ See Volumes to persist data)

# How does Docker Achieve Isolation?
Docker achieves isolation through Linux containerization, kernel namespaces, control groups (cgroups), read-only filesystem layers, and security profiles. These technologies work together to create secure, isolated environments for running containers, ensuring that each container is separated from others and the host system.

# How can Isolated Containers Communicate?
Isolated containers can communicate with each other using various networking methods and technologies provided by containerization platforms like Docker. Here are some common ways isolated containers can communicate:

1. Bridge Networking: Docker creates a private internal network bridge by default, known as the "bridge network." Containers connected to this bridge network can communicate with each other using their container names or IP addresses. For example:

```bash
docker run --name container1 -d myapp1
docker run --name container2 -d myapp2
```

Containers `container1` and `container2` can communicate using their container names.

2. Custom Bridge Networks: You can create custom bridge networks in Docker and connect containers to them. Containers within the same custom bridge network can communicate with each other using container names or IP addresses. This provides network isolation from containers in other networks.

```bash
docker network create mynetwork
docker run --name container1 -d --network=mynetwork myapp1
docker run --name container2 -d --network=mynetwork myapp2
```

Containers `container1` and `container2` within the `mynetwork` can communicate.

3. **Host Networking:** Containers can use the host's network stack, effectively sharing the host's network namespace. This allows containers to communicate with each other as if they were running directly on the host system. Be cautious with this method, as it reduces network isolation.

```bash
docker run --name container1 -d --network=host myapp1
docker run --name container2 -d --network=host myapp2
```

Containers `container1` and `container2` share the host's network stack.

4. Overlay Networks: Docker supports overlay networks for container communication across multiple hosts in a cluster. This is useful for distributed applications and services running in isolated containers.

Remember that the specific networking method you choose depends on your use case, whether you need communication between containers on the same host, containers on different hosts, or containers in a distributed environment. Docker provides flexibility in networking configurations to suit various application architectures.

-- Docker Storage --

# “External” storage – why?

- Data written in the top writable layer only persists as long as the container is running (ephemeral data).

- Top writable layer is tightly coupled to the host, moving data is not that easy.

- When writing into the top writable layer, you need a storage driver, which consumes extra resources and reduces performance.

# Types of Storage: There are two types of permanent storage:

<u>1. Data Volumes (managed by docker host – special location within host’s filesystem, preferred option):</u>

- Data volumes are managed by docker engine
- A data volume is a directory or file in the host’s filesystem that is mounted directly into a container
- When a container is deleted, any data written to the container that is not stored in a data volume is deleted along with the container
- You can mount any number of data volumes into a container
- Multiple containers can also share one or more data volumes

<u>2. Bind Mounts (arbitrary folder on docker host, mounted in a container):</u> It can be security critical when sharing config or sensetive data.

- Managed by the user (!!)
- Read-only access might be a good choice in many cases
- Sharing configuration files might be beneficial (they normally do not reside within docker’s filesystem).

# Docker Engine: There is three parts in docker engine,

1. Server (Deamon)
2. Rest API
3. Client

# Some important commands of docker engine:

images - List images
rmi - Remove one or more images
tag - Tag an image into a repository
inspect - Return low-level information on an image

pull - Pull an image from a registry
push - Push an image to a registry
search - Search the Docker Hub for images

ps - List containers
run - Run a command in a new container
start - Start one or more stopped containers
stop - Stop a running container
commit - Create a new image from a container
rm - Remove one or more containers

diff - Inspect changes on a container's filesystem
inspect - Return low-level information on a container
logs - Fetch the logs of a container
stats - Display a resource usage statistics
top - Display running processes of a container

# Docker File: A docker file is a text file used to define a series of instructions for building a Docker image. It specifies what should be included in the image, such as the base image, files to copy, commands to run, and more.

-- Important Commands:

FROM -- sets the Base Image for subsequent instructions (mandatory).

RUN -- will execute any commands in a new layer on top of the current image and commit the results.

CMD -- The main purpose of a CMD is to provide defaults for an executing container (only one allowed).

EXPOSE -- informs Docker that the container listens on the specified network ports at runtime.

ENV -- sets an environment variable

ADD or COPY -- copies files, directories or remote file URLs to the filesystem of the container

ENTRYPOINT -- allows you to configure a container that will run as an
executable – first command which is executed. Also see CMD.

VOLUME -- creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

WORKDIR -- sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions.

ONBUILD -- adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build, e.g., an application
runtime container

#Best practices for images:

Here are the best practices for Docker images made easy to understand:

1. **Ephemeral Containers:** Containers should be disposable, meaning they can be stopped and replaced with minimal setup and configuration. This ensures flexibility and resilience.
2. **Avoid Unnecessary Packages:** Don't install unnecessary packages in your containers. This reduces complexity, dependencies, image sizes, and build times.
3. **Domain-Driven Containers:** Separate your applications into multiple containers based on their domains. This simplifies scaling and reusing containers. Use container linking when services depend on each other.
4. **Minimize Layers:** Find a balance between readability and minimizing the number of layers in your Dockerfile. Fewer layers are more efficient.
5. **Keep Images Small:** Smaller images result in faster pull times, quicker loading into memory, and faster container startup. Start with the right base image that includes only what you need.
6. **Use Multi-Stage Builds:** Multi-stage builds help create smaller final images by separating the build and runtime environments. This reduces the image size while keeping the build tools isolated.
7. **Share Layers:** Define your own base images if you have a common basis across multiple images. Docker caches layers, so sharing common layers reduces download times.
8. **Meaningful Image Tags:** Use meaningful tags for your images to make them easily identifiable and versioned.

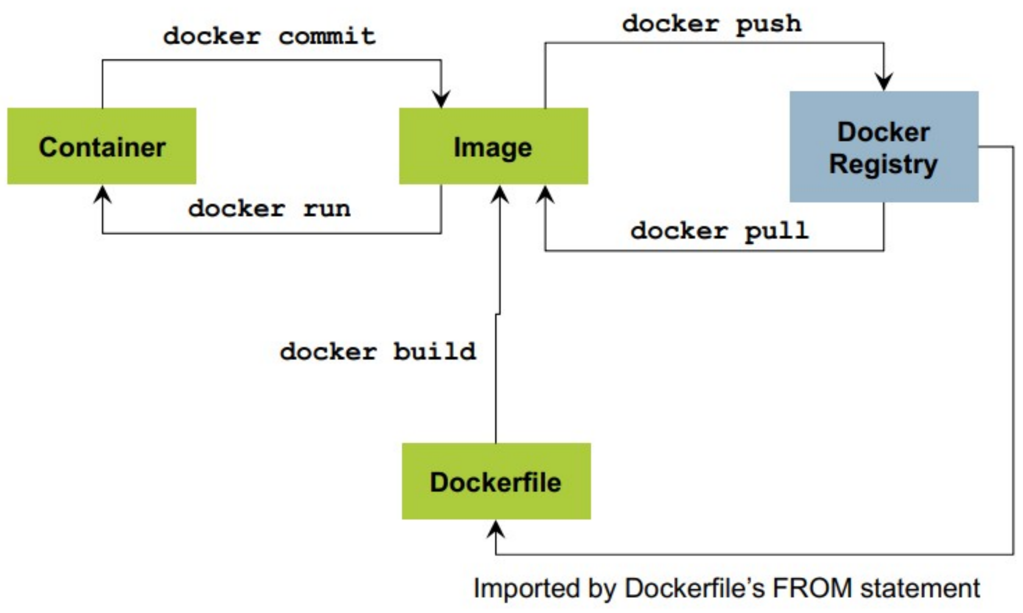
# Best practices for Dockerfiles:

Here are the best practices for Dockerfiles in an easy-to-understand format:

1. **One Instruction, One Layer:** Each instruction in your Dockerfile creates a separate layer. Try to minimize the number of layers to keep things efficient.

2. **Use .dockerignore:** Just like .gitignore for Git, .dockerignore helps exclude unnecessary files from being added to the Docker image.
3. **Multi-Stage Builds:** Think of multi-stage builds like using multiple workbenches to build a project. You can select, copy, and alter files from one stage to another. Name your build stages for clarity.
4. **External Images as Stages:** You can use existing external images as stages in your build process. This can be handy when you need certain tools for building, but you don't want them in the final image.
5. **Last Stage Only:** Remember that only the layers from the last stage are included in the final Docker image. The previous stages are used for building and preparing, not for the final image.

# Docker Process:



Sure, here's a simplified Docker process with the required commands:

1. Create Dockerfile: Write a Dockerfile to define your app's environment.

```dockerfile
# Dockerfile example

FROM python:3.9
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

2. Build Image: Use `docker build` to create an image from the Dockerfile.

```bash
docker build -t myapp-image .
```

3. Run Container: Start a container from the image using `docker run`.

```bash
docker run -d --name myapp-container myapp-image
```

4. Interact with Container: You can access your running app inside the container.

```bash
docker exec -it myapp-container /bin/bash
```

5. Stop and Remove Container: When you're done, stop and remove the container.

```bash
docker stop myapp-container
docker rm myapp-container
```

These commands cover the basics of creating, running, and managing Docker containers. Remember them to work with Docker effectively.

# **Docker Registry:**

- Stateless and scalable server-side application.
- Stores and distributes Docker images.
- Contains image layers and their descriptions.
- Can be hosted locally for full control of the image pipeline.
- Many users prefer Docker's public registry for convenience.

In summary, the Docker Registry is a scalable system for storing and sharing Docker images, and users can choose to use Docker's public registry or host their own for greater control.

# **Docker Hub:**

- Centralized platform for all things related to Docker containers.
- Offers image discovery, distribution, and version management.
- Supports user and team collaboration.
- Facilitates workflows in the development pipeline.

Key Features:

- **Image Repositories:** Easily find, manage, push, and pull container images from various sources, including community, official, and private libraries.
- **Automated Builds:** Automatically generate new container images when you update a source code repository. This streamlines the image creation process.

- **Organizations:** Create teams or groups to efficiently manage user access to image repositories, promoting collaboration among team members.

In essence, Docker Hub is your go-to platform for everything related to Docker containers, from finding images to managing workflows and collaborating with others.

#"Multi-container apps are a hassle": This means that managing applications composed of multiple containers can be complex or challenging. When you have an application split into multiple containers, each container may have different configurations, dependencies, and communication requirements. Coordinating and maintaining these containers can become more intricate compared to managing a single-container application.

#**Docker Compose:** For simpler applications, Docker Compose is a tool that allows you to define and manage multi-container applications using a single YAML file. It simplifies the process of starting and linking containers. All of that can be done by Compose in the scope of a single host.

-- Commands:

```
build -- Build or rebuild services
logs -- View output from containers
port -- Print the public port for a port binding
ps -- List containers
rm -- Remove stopped containers
run -- Run a one-off command
scale -- Set number of containers for a service
start -- Start services
stop -- Stop services
up -- Create and start containers
```

#Code & Explanation:

```
# Stage 1: Build stage
FROM gradle:7.3-jdk17 AS build-env
RUN mkdir -p /root/dev/beverage
WORKDIR /root/dev/beverage
COPY . /root/dev/beverage
RUN rm -rf /root/dev/beverage/build
RUN gradle build

# Stage 2: Final stage
FROM amazoncorretto:17
WORKDIR /root/
COPY --from=build-env /root/dev/beverage/build/libs/group2.beverage-all.jar .
EXPOSE 8080
CMD ["java", "-jar", "group2.beverage-all.jar"]
```

Explanation:
Stage 1: Build Stage

- `FROM gradle:7.3-jdk17 AS build-env`: This sets the base image to a Gradle image with JDK 17.
- `RUN mkdir -p /root/dev/beverage`: Creates a directory for your project.

- `WORKDIR /root/dev/beverage`: Sets the working directory to your project directory.
- `COPY . /root/dev/beverage`: Copies your application source code and files into the container.
- `RUN rm -rf /root/dev/beverage/build`: Removes any existing build artifacts if they exist.
- `RUN gradle build`: Builds your Gradle-based Java application.

Stage 2: Final Stage

- `FROM amazoncorretto:17`: Sets the base image to Amazon Corretto (a distribution of OpenJDK) with JDK 17.
- `WORKDIR /root/`: Changes the working directory to `/root/`.
- `COPY --from=build-env /root/dev/beverage/build/libs/group2.beverage-all.jar .`: Copies the built JAR file from the previous build stage into the current stage.
- `EXPOSE 8080`: Exposes port 8080 for the application to listen on.
- `CMD ["java", "-jar", "group2.beverage-all.jar"]`: Specifies the command to run when the container starts, which is running your Java application JAR file.

Kubernetes

# Definition: *Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications.* It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a robust and flexible framework for container orchestration, making it easier to manage and scale container-based applications, especially in complex and dynamic environments.

K8 is so developer friendly, because containerization transforms the data center of being machine-oriented to being application oriented.

# Benefits of Kubernetes:

1. Abstraction of Infrastructure:

- K8s abstracts away the complexities of hardware and operating system details. You don't need to worry about the underlying infrastructure.
- It provides a higher-level API, making it easier to manage applications using containers.
- This abstraction enhances portability, allowing you to move applications easily between different environments (e.g., development, testing, production).

2. Resource Efficiency:

- K8s helps you use your computing resources more efficiently.
- It reduces the cost of running servers by optimizing their utilization and simplifying management.

- Containers, which K8s orchestrates, are lightweight and efficient compared to traditional virtual machines (VMs) or bare-metal servers.

3. Development Speed:

- K8s accelerates development by enabling iterative improvements and updates to your applications.
- You can introduce new services with zero downtime, ensuring a seamless user experience during updates.
- Declarative configuration means you specify "what" your application should do, not "how" to do it. K8s takes care of the "how" part, making your system more reliable.

4. Supports Scaling Development:

- K8s supports the growth of your development efforts.
- It encourages decoupled architectures, allowing different parts of your system to scale independently.
- Each development team can be responsible for a single microservice, promoting agility and faster development.

In summary, Kubernetes simplifies infrastructure management, optimizes resource utilization, speeds up development, and supports scalable software architecture. It's a valuable tool for modern application deployment and management.

# **Kubernetes cluster –** two types of resources:

- Master

♣ Managing the cluster
♣ Coordinates scheduling, maintaining desired state, scaling, rolling out new updates etc.

- Node

♣ VM or physical computer
♣ Serves as a worker within the cluster

- Kubelet

♣ Agent for managing node
♣ Node process to communicate with the master (Kubernetes API)

# Kubernetes (K8s) Deployment and App:

Deployment:

- A Deployment in K8s provides instructions on what to create and update within the cluster.
- When a Deployment is created, the K8s master schedules the associated apps (containers) on worker nodes in the cluster.
- The Deployment Controller continuously compares the desired state (defined in a declarative configuration) with the observed state of the cluster and takes actions if necessary.

    - For example, if a machine fails, the Deployment Controller automatically replaces failed instances, ensuring self-healing.
- You can create and update Deployments using the `kubectl` command-line tool.
    - Example: `kubectl create deployment DEPLOYMENT-NAME --image=ADDRESS`
- To connect to the private network and access management information, you can use the `kubectl proxy` command. It exposes K8s management information via a REST-based API.

App:

- In the context of Kubernetes, an App typically refers to an application or microservice that needs to be containerized using technologies like Docker.
- You can specify the desired number of replicas for your App within a Deployment. It's often recommended to have at least three replicas for redundancy and availability.
- Other settings related to your App can be specified or changed using `kubectl` commands.

In summary, a Kubernetes Deployment defines how applications (Apps) should be managed within the cluster. It ensures that the desired state is maintained, handles automatic recovery in case of failures, and allows for easy scaling and management of containerized applications.

# PODS: A Kubernetes Pod is the smallest unit of deployment, containing one or more containers that share the same network and storage. It's used to group related processes and is managed by higher-level controllers for scalability and reliability.

--A Kubernetes Pod's lifecycle includes stages:

Pending: Awaiting node assignment.
Running: Containers executing.
Succeeded: Containers completed successfully.
Failed: Container failure.
Unknown: State unclear.
Terminated: Not running. Managed by controllers for auto-restart.

# **Kubernetes Service:** A Kubernetes Service is like a wrapper around a group of Pods. Here's what it does in an easy and simple way:

- **Abstraction:** It's like a logical grouping of Pods, each with its own IP address. This helps with automatic healing if a Pod fails.
- **Selects Pods:** The Service selects a set of Pods based on a defined LabelSelector.
- **Traffic Management:** It lets applications receive incoming traffic.
- **Types of Services:**
    - **Cluster IP (default):** Makes the Service accessible only within the cluster.
    - **NodePort:** Exposes the Service on each node's IP, mainly for debugging.
    - **LoadBalancer (recommended):** Sets up a load balancer and assigns a fixed, external IP for external access.

- **ExternalName:** Exposes the Service with an external name, like kube-dns.

In short, a Kubernetes Service makes it easy to manage and access groups of Pods, ensuring your applications stay available and can be accessed both internally and externally.

# Kubernetes label and selector: In Kubernetes, Labels and Selectors are used for organizing objects. Labels are key/value pairs for identifying metadata, while Selectors help group and filter objects based on these labels. There are two types of Selectors: equality-based and set-based.

# ReplicaSets and DeamonSets:

ReplicaSets:

- Used for running multiple replicas (copies) of the same container.
- Reasons: Redundancy (avoiding service outage), scaling (handling varying requests), and sharding (parallel task partitioning).
- Treats a group of pods as a logical entity.
- Enables Kubernetes' self-healing property by rescheduling pods during failures.
- Monitors pods using labels, allowing ReplicaSets to manage them effectively.

DaemonSets:

- Ensures a single copy of a Pod runs on every node in the Kubernetes cluster.
- Commonly used for tasks like running logging or monitoring agents on each node.
- Managed by a replication controller.

Difference:

- ReplicaSets can place multiple replicas on the same node, while DaemonSets ensure only one instance runs on each node in the cluster.

In summary, ReplicaSets are for managing multiple replicas of containers, while DaemonSets are for ensuring specific Pods run on every node, making them suitable for tasks like logging and monitoring agents.

#Data Integration: This is the most complicated step in building duistributed systems in container and container orchestration environments. Because due to legacy systems and evaluation of cloud, data services are often externalized in cloud services and it is difficult to host them in local cluster.

There is two options for data integation,

1. Externalized cloud service like dynamoDb
2. Using storage solutions within K8:
   a. Running reliable singletons: Using K8s features like ReplicaSets, BUT not replicate the storage. As reliable as running a single database VM or physical server, i.e. more reliable since eplication controller handles failures or other outages of your singleton.

   b. Using StatefulSets: StatefulSet is “headless”, means that the service has no ClusterIP since all pods are unique and have its own access point.

# StatefulSets and volume: StatefulSets in Kubernetes are used for managing stateful applications with ordered and unique Pods. Volumes, on the other hand, provide data storage options within Pods, ensuring data persistence and availability. Together, they enable the management of complex stateful workloads in Kubernetes.

# Monitoring: Monitoring and getting information is not that easy in a containerized application (all entities are encapsulated).

- Use Pod's container information via:

♣ $ kubectl describe pod <pod-name>
♣ $ kubectl logs <pod-name> -c <container-name>

To monitor system usage the Prometheus Architecture, where a combination of Kubernetes for container orchestration, Prometheus for collecting metrics and Grafana for visualizing this data.

#Health Check:

Checks are application specific and included in YAML files.

Liveness>>
If container does not respond in appropriate time, container is restarted (that's the default; dependent on restart policy) Result: application is running properly

Readiness>>
If container does not respond in appropriate time, container is removed from service load balancers Result: application is ready to serve requests..

# Persistent Volumes Subsystem : "The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed"

-
-PersistentVolume (PV):♣ Created by a cluster administrator
♣ Piece of Storage in your cluster (handled like any other Kubernetes object)
♣ Lifecycle independent of individual pods

--PersistentVolumeClaim (PVC):
♣ Request for storage by a user
♣ PVCs consume PV resources
♣ Claims can access specific size and access modes on a PV (decoupling)

# **ConfigMaps and Secrets**

- ConfigMaps: Set of external variables for container
- Secrets: For sensitive environment variables/data

Three ways of using a ConfigMap
♣ Filesystem: mount ConfigMap in a Pod
♣ Environment variable: set values of environment variables
♣ CLI argument: K8s creates CLI argument based on ConfMap

# Kubernetes (K8s) System Architecture consists of several key components:

apiVersion: apps/v1kind: Deployment metadata:

etcd: This is the core state store for Kubernetes, which stores all the configuration data and state information about the cluster. It's like the brain of the system, but it's hidden and only accessible via the Kubernetes API Server.

API Server: The API Server is like the central command center of Kubernetes. It provides a simple REST API that users and applications interact with. It handles important tasks like authentication, authorization, and admission control (which decides whether to accept or modify incoming requests). It also manages API versioning.

Scheduler/Controller Manager: These components are responsible for making the Kubernetes system work smoothly. The Scheduler assigns Pods to specific nodes, ensuring efficient resource allocation. The Controller Manager handles various controllers that reconcile the desired state of the system with the actual state, ensuring that everything runs as expected.

Kubelet: Kubelet is like the local agent on every node in the cluster. It communicates with the API Server and ensures that the containers in its node are running as specified. It handles tasks like starting, stopping, and monitoring containers.

Microservices

# What are Microservices?

Microservices are a software architectural approach where a complex application is broken down into smaller, independent services that communicate with each other through well-defined APIs. These services are designed to be small, focused on specific business capabilities, and run as separate processes or containers. Each microservice operates independently and can be developed, deployed, and scaled independently.

In other words, The term “Microservices” refers to an architectural style that organizes applications as loosley coupled, independently manageable services. Here Target application area are enterprise applications. The word “micro” in Microservices is misleading and does not mean creating as small services as possible, instead services are aligned with business capabilities.

# Why Microservices?

--> To get a modular, evolvable and robust architecture.
--> Decouple development
  - ♣ Speed up dev / build / release
  - ♣ Enable individual infrastructure and code upgrades per service

--> Flexibly choose between programming frameworks / persistence

# Essential principles of microservices with more details and relevant scenarios:

1. **Componentization via Services:**
    - *Explanation:* Microservices break down a complex application into smaller, independent services that can be developed, deployed, and scaled separately.

- *Scenario:* Consider a ride-sharing platform. Instead of building a monolithic app that handles everything from user registration to ride booking, you create separate microservices for user authentication, ride matching, payment processing, and driver tracking.

2. **Organized around Business Capabilities:**
   - *Explanation:* Microservices are organized based on specific business functions or capabilities, allowing teams to focus on what they do best.
   - *Scenario:* In an e-commerce system, you might have services like "Product Management," "Order Fulfillment," and "Customer Reviews," each responsible for a distinct business capability.
3. **Products not Projects:**
   - *Explanation:* Microservices are considered long-term products rather than short-lived projects, requiring ongoing maintenance and improvement.
   - *Scenario:* Think of a messaging service like WhatsApp. It constantly evolves with updates, bug fixes, and new features to meet user demands.
4. **Smart Endpoints and Dumb Pipes:**
   - *Explanation:* Microservices communicate through simple, standardized channels (dumb pipes), while the services themselves are responsible for complex processing (smart endpoints).
   - *Scenario:* Services communicate via RESTful APIs or message queues, where the sender and receiver understand the data format. The services independently process and respond to messages.
5. **Decentralized Governance / Polyglot X:**
   - *Explanation:* Microservices allow development teams to choose the best tools and technologies for their specific service, promoting flexibility and expertise.
   - *Scenario:* In a media streaming platform, video encoding services may use specialized tools optimized for video processing, while user profile services might use a different technology stack that suits their needs.
6. **Decentralized Data Management:**
   - *Explanation:* Each microservice manages its own data store, reducing inter-service dependencies and enabling scalability.
   - *Scenario:* A hotel booking application may have separate services for booking management, customer reviews, and user profiles. Each service has its own database tailored for its data requirements.
7. **Infrastructure Automation:**
   - *Explanation:* Automation tools like Kubernetes or Docker Swarm handle the provisioning, scaling, and management of microservices infrastructure.
   - *Scenario:* These tools ensure that microservices are deployed consistently, replicated for high availability, and scaled based on demand, all without manual intervention.
8. **Design for Failure:**

- *Explanation:* Microservices should anticipate and gracefully handle service failures, network issues, and other unexpected problems.
- *Scenario:* An e-commerce checkout service might implement retry mechanisms when connecting to a payment gateway. If the payment service is temporarily unavailable, the checkout service retries the request or provides an alternative payment option.

9. **Evolutionary Design:**
   - *Explanation:* Microservices evolve based on real-world usage and changing business requirements. Feedback, metrics, and insights drive continuous improvement.
   - *Scenario:* A social media platform constantly updates its recommendation algorithm based on user interactions and content trends, ensuring a personalized user experience.

These principles guide organizations in building microservices architectures that are adaptable, scalable, and well-suited to modern software development needs.

# **Polyglot Persistence** means using multiple database technologies in a single application based on the specific data storage needs, ensuring optimal performance and scalability for each data type.

# Best Practices for Cloud (Web) Development:

1. Codebase
One codebase tracked in revision control, many deploys

2. Dependencies
Explicitly declare and isolate dependencies, no system-wide packages

3. Config
Store config in the environment

4. Backing Services
Treat backing services as attached resources

5. Build, release, run
Strictly separate build and run stages

6. Processes
Execute the app as one or more stateless processes

7. Port binding
Export services via port binding

8. Concurrency
Scale out via the process model

9. Disposability
Maximize robustness with fast startup and graceful shutdown

10. Dev/prod parity

Keep development, staging, and production as similar as possible

11. Logs
Treat logs as event streams

12. Admin processes
Run admin/management tasks as one-off processes

# Liabilities of using Microservices

Liabilities of using microservices include increased complexity, operational overhead, data consistency challenges, communication overhead, testing complexity, security concerns, service discovery, monitoring and debugging complexity, deployment challenges, versioning and compatibility issues, scalability considerations, higher costs, and the need for developer expertise.

It's important to note that while these are liabilities, they are not necessarily prohibitive. Many organizations successfully adopt microservices by addressing these challenges with careful planning, architecture, and the use of appropriate tools and best practices. The decision to use microservices should be based on a careful consideration of both their benefits and liabilities in the context of your specific application and organization.

# Do Microservices comply with SOA characteristics?

Yes, Microservices share some similarities with the characteristics of Service-Oriented Architecture (SOA), but they also have distinct differences.

Microservices and SOA share some common principles, such as decentralization, fine-grained services, and flexibility in technology choices. However, Microservices tend to emphasize these characteristics more strongly and are often associated with modern, lightweight communication protocols and smaller, independently managed services. SOA, in contrast, has a broader history and can encompass a wider range of service architectures, including both fine-grained and coarse-grained services, with varying degrees of centralization. The choice between Microservices and SOA depends on the specific goals and requirements of the software system and organization.

In this analogy, Microservices represent highly specialized, independent services (food trucks) with a focus on flexibility and autonomy. SOA, on the other hand, provides a structured approach to services (restaurant sections) that may offer a broader range of options but comes with more centralized control and coordination. The choice between Microservices and SOA depends on your project's specific needs and constraints.

# Does Service Oriented Architecture comply with Microservices Characteristics?

Service-Oriented Architecture (SOA) and Microservices share some similarities, but they are not the same, and SOA does not fully comply with all of the characteristics of Microservices.

Technical Difference:

1. SOA often relies on WS-* standards like SOAP, WSDL, and UDDI for service definition and communication.

2. Microservices typically use HTTP/RESTful APIs for communication and may implement containerization and orchestration using tools like Docker and Kubernetes.

3. SOA can involve more heavyweight enterprise service buses (ESBs) for mediation and routing.

4. Microservices often embrace lightweight service discovery and API gateways for routing.

In summary, while both SOA and Microservices aim to create modular and reusable services, they differ in their granularity, communication style, and approach to data management. Microservices tend to be more fine-grained, use lightweight protocols, and promote decentralized data, whereas SOA services are often more coarse-grained and can involve heavier communication standards. The choice between them depends on the specific requirements and constraints of a project.

# Do Microservices dictate usage of REST?

Microservices don't dictate the exclusive use of REST, but they align well with REST principles, such as addressable resources, representation-oriented manipulation, self-descriptive messages, and stateless communication. HATEOAS is beneficial but not mandatory in microservices architecture.

REST is often a preferred choice for Microservices due to its simplicity and alignment with Microservices principles, it is not a strict requirement. The selection of a communication protocol should be based on the project's specific technical and business requirements.