Backend Developer Roadtrip

Random Collection of Knowledge

EDUARDO CARDENAZ

Part I: Basics

1 APIs

Application Programming Interface (API) An API is basically an intermediary that allows two applications to talk to each other.

It's useful to think of API communication in terms of requests and responses between a client and a server. The application submitting the request is the client, and the server provides the response.

API Specification Is a document or standard that describes how to build or how to use an API. A system that meets this standard its said to be *implementing* or *exposing* an API. The term API can refer to both the implementation and the specification.

1.1 RESTful APIs

REST stands for Representational State Transfer, and it's an architectural style for designing networked applications. A RESTful API (Application Programming Interface) adheres to the principles of REST.

"The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture." source

To understand REST, we'll be expanding and building on top of each constraint that composes this architecture.

Violating any constraint other than Code on Demand means that the service is not strictly RESTful.

1.1.1 Uniform Interface

Resource Based Individual resources are identified using URIs as resource identifiers. The resources themselves are conceptually different from the *representation* that is returned to the client. For example, the server doesn't return its database but rather some HTML, JSON or XML that represents some database records.

Manipulation of Resources Through Representations When a client holds a representation of a resource given by the server, including any metadata attached to it, it should have enough information to modify or delete said resource on the server, given it has the right permissions.

Self-Descriptive Messages Each message includes all the necessary information to describe how to handle that message.

1.1.2 Client-Server

Separation of Concerns is the principle behind this constraint. By separating the user interface (UI) concerns from the data storage concerns, we improve the portability of the UI across multiple platforms.

1.1.3 Stateless

This constraint stablish that the interaction between the client and the server must be stateless in nature, by that, meaning that any request made by the client must contain all the necessary information so that the server can understand the request. It shall not take advantage of any stored context.

Stateless Imagine that you're buying coffee at a shop. Each time you want to order a coffee you need to tell the cashier exactly what you want and how you want it. The cashier doesn't remember you or what you ordered the last time you went there. Each visit is like starting from scratch.

Stateful The cashier remembers you and remember what you usually ask for as you're a frequent client. So, you could say 'the usual' and they'd know exactly what you want.

1.1.4 Cacheable

Clients should can cache responses. Responses must, implicitly or explicitly, define themselves as cacheable or not. Clients should be able to negotiate wether to cache or not to prevent reusing stale or inappropriate data in response to further requests.

1.1.5 Layered System

A client cannot tell whether its connected to the main server or an intermediary along the way. The layered system style allows an application to be composed of hierarchical layers by constraining component behavior such that each component can't see beyond the immediate layer with which they are interacting.

1.1.6 Code on Demand

This is a kind-of unique thing about RESTful systems. Code on Demand is optional. Basically means that a server can temporarily extend functionality to a client by transferring logic to the client. As an example, a request can return client-side scripts such as Javascript code.

1.2 JSON APIs

1.3 SOAP APIs

1.4 GraphQL APIs

1.5 gRPC APIs

1.6 Authentication & Authorization

Authentication is the process that an individual, process or system goes through to **prove** their identity before gaining access to digital systems.

Authorization is the process of determining whether a user, system or application has the necessary permissions to perform a particular action within a system.

API Authentication validates the identity of the client attempting to make a connection by using an authentication protocol. The protocol sends the credentials from the remote client requesting access to the server for verification, usually the credentials are in either plain text or some form of encrypted format.

1.6.1 Session Authentication

Session A session, refers to a temporary, stateful interaction between a user and a server. A session, is a period of interaction between a user an a server, during which the user's actions and data are temporarily stored and managed by the server.

- **Duration**: A session begins when the user logs into the server and ends when they log out, or after a period of inactivity.
- **Session Data**: During the sessions, the server stores information relevant to the user's interaction, such as their authentication status, user ID, preferences, etc.
- **Session ID**: To uniquely identify each sessions, the server assigns a session identifier. This session id is used to to associate the Session Data with the User

Example When you log in (authenticate) into a web application, the server creates a session and then keeps track of it itself. Then it creates a Session ID and gives it to you, subsequently, the client (you) pass this Session ID to the server with each request. Then, the Server looks this Session ID up in its Session Log, and if it finds it, it knows who you are and what you're allowed to do.

How the client passes the Session ID to the server, depends on the implementation, but it's usually done through Cookies.

1.6.2 JWT

1.6.3 Basic Auth

1.6.4 Token Auth

1.6.5 OAuth

1.6.6 Cookie Based

1.6.7 OpenID & SAML

2 Caching

2.1 Client Side

2.2 Server Side

2.3 Content Delivery Network (CDN)

2.4 Redis

2.5 Memcached

3 Web Security

3.1 Hashing Algorithms

3.2 API Security Best Practices

4 Testing

4.1 Unit Testing

4.2 Integration Testing

4.3 Functional Testing

5 CI/CD

6 Databases

7 Software Design & Architecture

7.1 Design and Development Principles

7.1.1 Separation of Concerns

7.1.2 Reusability

7.1.3 Keep It Simple Stupid (KISS)

7.1.4 Don't Repit Yourself (DRY)

7.1.5 Scalability

7.1.6 Security

7.2 GOF Design Patterns

7.3 Domain Driven Design

7.4 CQRS

7.5 Event Sourcing

8 Architectural Patterns

9 Message Brokers

9.1 RabbitMQ

9.2 Kafka

10 Containerization Vs Virtualization

10.1 LXC

10.2 Docker

10.3 Kubernetes

10.4 Elasticsearch

10.5 Solr

11 Web Servers

11.1 Server Sent Events

11.2 WebSockets

11.3 Long Polling

11.4 Short Polling

12 GraphQL

12.1 Apollo

13 NoSQL Databases

13.1 Document DBs - MongoDB

13.2 Time Series - InfluxDB

13.3 Realtime - Firebase

13.4 Column DBs - Cassandra

13.5 Key Value - Redis

13.6 Graph DBs - Neo4j

14 Building for Scale

14.1 Difference + Usage

14.2 Mitigation Strategies

Part II: Infraestructure Knowledge

15 Go Programming Language

15.1

15.2

16 Networking and Protocols

16.1

16.2

17 Docker

Container A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

A container is a way to package our applications including all the dependencies that it needs, including its configuration files. This makes them portable. Making development and deployment much faster.

Image A Docker Image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

An image is a packaging containing all the dependencies and the code. This is what's gonna be shared. Finally a container is an image we configured which its going to have all the dependencies running, alongside with environment variables, etc. A container, is basically layers upon layers of images, where usually the bottom layer is a Linux image.

A container is an instance of an image.

17.1 Virtualization

At its core, Docker is a virtualization tool.

Virtualization Is the process of creating a virtual version of something, such as an operating system, a server, a storage device or network resources.

Host Being the physical machine that runs the virtualization software.

Guest The virtual machine or OS that runs on the host.

Layers of Virtualization The concept of virtualization is based on three layers.

- **Hardware**: Where the physical hardware is located.
- **Kernel**: The kernel is the core of the operating system. It's the bridge between the hardware and the software.
- **Application**: This is your software, the applications you run on your computer.

A Virtual Machine (VM) is a software that emulates a physical computer. It creates a virtualized environment that behaves like a separate computer. When you run a VM, the Kernel and the Application layers are what's being virtualized.

As you may realize, this is very resource intensive.

Docker , on the other hand, lives on the Application layer. It doesn't virtualize the hardware, nor the kernel. It just virtualizes the application. This makes it much more lightweight than a VM.

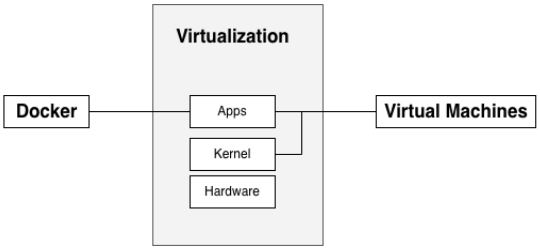

Figure 1: Docker vs VMs

Types of Virtualization Exists three types of virtualization:

- **Para-Virtualization**: In para-virtualization, the host is trying to give the guest most of their hardware.
- **Partial Virtualization**: Where some components of the host's hardware are virtualized and given to the guest.
- **Full Virtualization**: In full virtualization, the guest is given a full copy of the host's hardware.

For each of this cases, Docker is going to be superior, as Docker uses the kernel of the host directly.

17.2 Docker Network

Network A Network in simple terms is a group of two or more devices that can communicate with each other either physically or virtually.

Docker Network The Docker Network is a virtual network created by Docker to enable communication between Docker Containers. Container Networking refers to the ability for containers to connect to and communicate withe ach other, or to non-Docker workloads.

Containers have networking enabled by default, and they can make outgoing connections. A container has no information about what kind of network it's attached to, or whether their peers are also Docker workloads or not.

Port Mapping is a way to allow traffic to flow from the outside world into a container. Since containers are isolated environments, if you want to access a web application inside a container from your web browser, you need to map the port on which the application is running inside the container to a port on the host machine.

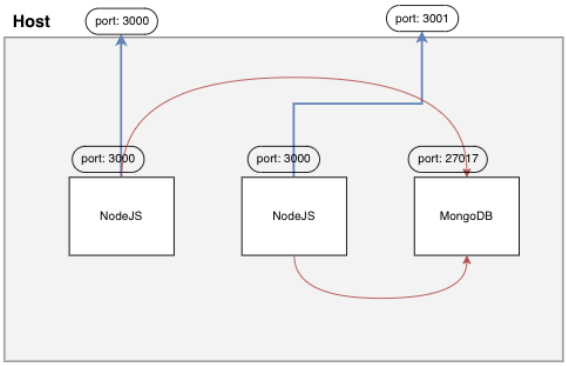

Figure 2: Docker Port Mapping

As per the figure above, the Node JS container is running on port 3000. To access this container from the host machine, we need to map the port 3000 of the container to a port on the host machine. In this case, we're mapping the port 3000 of the container to the port 3000 of the host machine.

Usually, you would find that in cases where you have multiple containers running on the same port, you would map the port of the container to a different port on the host machine. So, if you have two containers with Node JS each with a web application running on port 3000, you would map the port of the first container to port 3000 of the host machine, and the port of the second container to port 3001 of the host machine. Although this is not mandatory, it's a good practice to avoid conflicts.

17.3

17.4

18 Amazon Web Services

18.1

18.2

19 Terraform

19.1

19.2

20 Ansible

20.1

20.2

21 Github Actions

21.1

21.2