What is a Backend? A selective guided tour through terminology, approaches, and technology

Victor Adossi

April, 2019

Roadmap

- What is a backend?
- Past (AKA the "distant" past)
  - C programs
  - Java & Servlets
  - Common Gateway Interface
  - The LAMP stack
  - Reverse Proxies
- Present (AKA the very recent past)
  - Virtual Machines (VMs)
  - Platform as a Service (PaaS)
  - Infrastructure as a Service (IaaS)
  - Containers
- Future?
- How does one keep up?

Disclaimer

I wasn't personally writing software in 1990/1997, and I'm not a computer science/sofware engineering historian. We're going to discuss in very broad strokes, and there will be mistakes.

If you notice some inaccuracies, write them down/keep them in mind until the end and my email will be available so you can enlighten me (and I'll update the slides).

What is a backend?

"Backend" can refer to a lot of things, but we'll be mostly focusing on backends as used by/created for serving web/mobile traffic, in the "three tier architecture" style.

In interview question form

"What happens when someone types somewebsite.com into their browser presses the enter key?"

From a frontend engineer's perspective

"Where can I get the data?"

From an operations engineer's perspective

"How many instances do you need?"

A Backend (in Python)

A backend may look like many things to many people, but they must do one very specific thing – **answer requests sent by clients over ther internet**.

```
import socket

HOST, PORT = '', 8888

listen_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
listen_socket.bind((HOST, PORT))
listen_socket.listen(1)

while True:
    client_connection, client_address = listen_socket.accept()
    request = client_connection.recv(1024)

    client_connection.sendall("HTTP/1.1 200 OK\n\nHello, World!")
    client_connection.close()
```

Building A Web Server, Part 1 - https://ruslanspivak.com/lsbaws-part1

A working definition

Fundamentally, a backend needs to:

- Bind to a socket (at a certain address and port)
- Accept socket connections
- Receive some bytes for a given request
- (Optionally) Parse the received request
- Return some bytes as a response (in this case "Hello, World")

Put simply, we need to **receive a request, and return a response**

Too easy to be true? It is. There are many layers of abstractions below that enable this code to be relatively simple:

- Python runtime
- Operating System
- HTTP, TCP/IP, Ethernet, Networking, etc
- Hardware

Past

What did backends look like in the “distant” past?

The Past - C programs (~1990)

Cern's `httpd` is considered one of the first web servers.

- It's written in C (there were relatively few choices)
- C is powerful
- C is fast
- C is unsafe
- You probably shouldn't use C to write your web server in <year after 1990>

Let's avoid looking at C code today and pretend this server worked much like the previous example.

https://en.wikipedia.org/wiki/CERN_httpd

https://www.w3.org/Daemon

What did we gain?

Not to be understated, but as you might expect, building web servers is fundamental to the emergence of the World Wide Web.

We've achieved the basic functionality required of any backend – **receiving requests and returning responses**.

The Past - Java & Servlets (~199x)

Java arose as a safer, reasonably performant, cross-platform alternative to C, and people started using it to write web servers. Here's a current example:

```java
import java.io.IOException;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletLifeCycleExample extends HttpServlet {
    @Override
    public void init(final ServletConfig config) throws ServletException { ... }

    @Override
    protected void service(
        final HttpServletRequest request,
        final HttpServletResponse response
    ) throws ServletException, IOException {
        ...
    }

    @Override
    public void destroy() { ... }
}
```

https://en.wikipedia.org/wiki/Java_servlet

What did we gain?

Building servers in Java enables:

- Memory safe web servers
- More modular, cross-platform server programs
- Making web server development more approachable for developers

The ability to use a safer, memory-managed language like Java makes it easier to build things on the early internet

At this point, backends can **receive requests and return responses in a safe-yet-performant manner**, along with access to Java's easy-to-use rich ecosystem of libraries.

The Past - Common Gateway Interface (~1997)

Write *any* program that receives information about a web request through `stdin` and environment variables. Write the webpage you want to render to `stdout`.

In this era we get to witness:

- The rise of performant, modular web servers (ex. Apache)
- PHP, Python, and Perl rapidly gain popularity
- `.cgi` showing up in URLs all over the internet

PHP (1994) is what happens when we optimize a language for web development, and just sprinkle in some dynamic behavior:

```
<html>
 <head>
   <title>Hello World</title>
 </head>
 <body>
   <?php echo "Hello, World";?>
 </body>
</html>
```

The Past - CGI (continued)

What does our journey to receive a request and produce a response look like now?

- A request comes in to the Apache webserver (C)
- Apache finds the relevant handler program
- The CGI protocol is used to call the handler program
- The results of the program are sent to the user as a response

We've got at least 3 things to worry about:

- Apache
- CGI's Protocol
- The handler program

What did we gain?

Apache & CGI introdce a stepwise productivity improvement:

- Even more modular web servers
- The ability to easily use new, more focused programming langauges

Backends we write with the support of CGI can **focus more on business logic** and offload request/response wrangling completely, while achieving all our previous goals.

The Past - LAMP (2000+)

Another stepwise change in productivity after CGI was the assembly/adoption of the LAMP stack:

- **L**inux - free, customizable server operating system
- **A**pache - capable, modularized web server
- **M**ySQL - advanced application data management
- **P**HP - low-friction programming language built for the web

This paradigm is/was *dominant* – it reportedly made up for over 50% of internet traffic at one point.

Example: Wordpress

https://en.wikipedia.org/wiki/LAMP_(software_bundle)

What did we gain?

With the adoption of LAMP:

- Writing complex applications is now *much* easier
- Adoption of LAMP and the web form a virtuous cycle
- Linux, MySQL, PHP, and open source in general benefit greatly from the adoption/support

Backends we write now can **more easily render webpages, perform complex data operations, and run cheaply on commodity hardware**, while achieving all our previous goals.

The Past - Reverse Proxies (2004+)

"C10K" - 10,000 concurrent connections

Apache was fast, but didn't do so well with concurrent connections, which became a problem as more people joined the internet.

NGINX was written with the explicit goal of outperforming Apache, and solving the C10K problem

NGINX's worker thread & asynchronous event loop driven approach (versus Apache's thread-per-request*) allowed massive scale with limited resources, but it achieves this by doing *less* (it didn't handle dynamic content).

* https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations

The Past - Reverse Proxies (continued)

NGINX example:

```
http {
  upstream myproject {
    server 127.0.0.1:8000 weight=3;
    server 127.0.0.1:8001;
    server 127.0.0.1:8002;
    server 127.0.0.1:8003;
  }

  server {
    listen 80;
    server_name www.domain.com;
    location / {
      proxy_pass http://myproject;
    }
  }
}
```

Paradigm shift Programs bound to `localhost` (AKA `127.0.0.1`) on different non-reserved ports, multiplexed by NGINX

https://www.nginx.com/resources/wiki/start/topics/examples/loadbalanceexample/

What did we gain?

With the adoption of reverse proxies we gained:

- Relatively massive scale
- A simpler model for multiplexing applications
- Simplification of the "outer" web server (Apache vs. NGINX)

The emphasis on reverse proxying instead of CGI spurred investment in better software libraries/ecosystems for various languages.

Barrier to entry for new langauges to be used is now an ecosystem that contains *at least* the means to listen on `localhost` at some port, and speak HTTP. Many new languages have this in their *standard libraries*.

Now backends can be used for **websites with larger userbases, with flexibility in implementation language, and with *slightly* simpler architecture**.

Before we cross over into a more recent past. . .

We've come a long way – let's step back and remember our fundamental description of a backend: **receive a request, return a response**.

We've developed technology that makes it easier to do this in a safer and less error-prone manner than just writing completely custom applications in C:

- Easy use cases are easy (ex. `apache` and a folder of files),
- More complicated use cases (ex. dynamic content, database queries) are very managable
- Safer languages like Java, PHP, Perl, Python and Ruby
- Easy use of commodity hardware for servers w/ Linux
- Widely adopted advanced data manipulation via MySQL
- Focus on reverse proxying means simpler ("isomorphic"), more scalable systems

Present

What do backends look like in the very-recent past?

The Present - VMs

More computing power, memory, and hard-drive space is available than ever before – you can run *even more* programs!

How do you stop one (possibly compromised) running program from bringing down your entire web server?

System-level Virtual Machines (VMs) arise as a solution to this problem – allowing one or more processes to run in a completely virtualized machine, offering increased *isolation* and *security*, **without requiring application-level changes**.

With separate processes running in VMs, one crashed (or malicious) process can now no longer crash others or fatally harm the host system. New hardware with more resources can be safely utilized more efficiently.

The Present - VMs (continued)

But wouldn't it be really hard to make a completely virtual "machine" that mimics a whole computer? Yes – so we optimized:

- CPU manufacturers extend CPUs with new Virtual Machine Extensions ("VMX") instructions (Intel's VT-x and AMD's AMD-V)
- Linux gets Kernel-based Virtual Machine ("KVM") support in 2007

To streamline the process of creating VM images, tools like Packer were created

https://en.wikipedia.org/wiki/X86_virtualization

https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine

https://www.packer.io/intro/getting-started/build-image.html

What did we gain?

Most of the complexity of VMs is hidden from the backend developer. The process now looks something like this:

- A process in a language of our choice that started in a VM
- An "outer" web server like Apache (which can reverse proxy) or NGINX receives a request
- The outer web server processes and passes some request to the VM
- The process (which may succeed or crash) inside the VM processes the request

Now we can write backends that are **more secure and better isolated, which use abundant resources in a safer manner**.

We've also increased reproducibility – you can build the VM that's meant to run in production right on your own system.

The Present - Infrastructure as a Service (IaaS)

Now that you can easily and safely use server hardware it makes a *lot* more sense to sell extra capacity. Maybe one could even buy and maintain capacity for the express purpose of selling it!

Amazon Web Services (~2002/2006), by far the most popular "cloud provider" today is the culmination of that idea (and some others).

Elastic Compute Cloud ("EC2") and other offerings are only possible to operate reasonably safely with the isolation provided by VMs!

The Present - Platform as a Service (PaaS)

Simple concept: If we have well known "stacks" like LAMP and Ruby on Rails, well known databases like MySQL, and VMs as a reasonable way to isolate them, why not automate the experience of deploying software?

Heroku (2007) burst onto the scene offering exactly this – you provide the code, they provide the infrastructure and the scale.

Today, the idea of a build pack has become so well known that it's been spun out of Heroku all together and is being adopted as a near-universal interface.

https://buildpacks.io

The Present - Containers

Simple concept: Lighter weight isolation for processes.

VMs are the gold standard, but in many cases they're not completely necessary – due to the way linux is built, simply obscuring/hiding interfaces at the kernel boundary (user privileges, filesystems, syscalls), can offer "good-enough" isolation without sacrificing performance.

The Present - Containers (continued)

As the saying goes, "Containers do not exist" – they are a combination of two Operating System features:

- C-Groups (control of resources; compute, RAM, disk I/O, etc)
- Namespaces (visibility of various resources; process IDs, **filesystem**s, etc)

Containers are *not* VMs, but they achieve better-than-nothing isolation by using these two tools to make it *very difficult* for containerized processes to reach host resources.

Namespaces are especially effective means of isolating UNIX-y operating systems, since "everything is a file". If a spinning hard drive is represented by `/dev/sda`, and you can't *see* `/dev/sda`, you (usually) cannot write to that disk.

The Present - Containers (continued)

Containers are generally by-definition *less secure* than VMs and have had their share of growing:

- Container runtimes (`docker`, `containerd`, `cri-o`) gaining non-root user support
- `seccomp` support (limiting syscalls)
- `apparmor` / `SELinux` support (enforcing access control on system resources)
- `user` namespaces (`root` inside a container but *not* `root` outside)
- Capabilities & capability dropping support (reduce blast radius of compromised programs)
- Various kernel-level features, integrations, and bugfixes

I've purposefully left out Linux Containers (`lxc`) here – they're similar but slightly different.

User namespaces progress (2012) - https://lwn.net/Articles/528078

The Future

So where are backends going now/next?

Disclaimer

A few things to keep in mind:

- I know even less about the future than I know about the past
- You (probably) should not base your outlook on 1 slide in a short talk
- There are no silver bullets, and there are no werewolves. Good engineering is elegant, not magical.

The Future - More client-side rendering

Why?

- Offloading rendering to more powerful clients means smaller, simpler, more focused backend services
  - What about weaker clients? Don't worry, client-side rendering has also gone full circle with "isomorphic" apps and Server Side Rendering ("SSR")
- Specialization of *types* of backends (some serve data, some serve webpages)
- Faster iteration for separate teams (see: microservices)

Why not?

- Maybe you should just use your Rails/Django templating layer, instead of setting up 1/2+ new frontend tools

The Future - More shades of isolation

Why?

- There is space to be explored between raw process and VM
- Minimal VM Managers (AKA “Hypervisors”); NEMU, Firecracker, etc
- GVisor
  - A program wraps yours and pretends to be the kernel
- Unikernels
  - Leave out the bits of the kernel you don’t need
  - Go faster with more direct/optimizable hardware access
  - Much smaller attack surface, less code normally means less bugs

Why not?

- VMs might be good enough (for your use case)
- Containers might be good enough (for your use case)

The Future - Container Orchestration

Why?

- Containers simplify deployment (basically fat binaries v2.0)
- Containers offer better isolation that raw processes, and no need to manage OS primitives on a machine (users, groups, etc)
- Automation is (generally) good
- Consistent management of a pool of machines as one
- Research-theoretical resource efficiency via bin-packing
- Consistent deployment across clouds (Kubernetes is supported by AWS, Azure, Google Cloud Platform, and others)

Why not?

- Essential complexity is high for container orchestration systems
- Accidental complexity is also often non-zero

The Future - Functions as a Service

Why?

- Intense focus on business logic often means faster business success
- Near-optimal potential iteration speed

Why not?

- Not as portable as containers, due to different provider implementations (this is changing)
- Granularity - too much of a good thing?
- Cold start complexity
- Persistent server vs function tradeoff becomes tricky at scale (watch out for network/supporting infrastructure cost)

KNative - https://cloud.google.com/knative
Serverless cross-platform Framework - https://serverless.com
OpenFaaS - https://www.openfaas.com

The Future - Service Oriented Architecture (SOA) & Microservices

Why?

- Simpler, independently scalable systems
- For most businesses flexibility initially trumps robust design (see: Mongo)

Hot Take

The only discernable difference between SOA and microservices is a relience on a central message bus

Hot Take #2

All well-designed monoliths are actually just extremely closely co-located bundles of micro-services

The Future - Continuous Deployment

Why?

- Automation is important
- Systems that are deployed more frequently spend less time being broken*
- It's easier and safer to deploy than it's ever been before

*assuming errors go *down* over time

Why Not?

- The tooling is hard to build
- Attempting to automate exposes you to some fundamental hard problems (automated release quality checks in noisy environments)
- Essential complexity is even higher (see: Istio)

The Future - ???

No one knows what the future will actually hold, but it's actually here already, it's just unevenly distributed*.

* https://en.wikiquote.org/wiki/William_Gibson

How does anyone manage to keep up?

No one keeps up with *everything*. Things are moving very fast, but remember that fundamentals move slowly (if at all), since everything is built on them.

When evaluating new tools:

- What is the thing *supposed* to do and why?
- What did people use before?
- What does this new tool/approach bring to the table?
- What are the alternatives?

Know at least the "what" and "why", make a one-sentence summary in your own head, and refine your knowledge as things shift.

If the new tool is important, you'll see it again.

Keeping up case study: Prometheus

What is the thing supposed to do and why?

Collect (application|infrastructure|...) metrics. People need metrics to make decisions (ex. how much hard drive space do you have left?).

What did people use before?

- literal human intervention
- bash scripts (+/- `cron`, +/- automated emails)
- Nagios
- Graphite

Keeping up case study: Prometheus (continued)

What does this new tool/approach bring to the table?

- Labels versus hierarchical metric names, better support for high cardinality
- Raw data queriying (no forced aggregations)
- Simplicity of deployment, with most batteries included (gathering, displaying, alerting)

What are some alternatives?

- Graphite
- Nagios
- InfluxDB + Kapacitor
- Fleet focused tools:
  - netdata
  - Zenoss
  - Zabbix

Inclusion of a comparison page in the documentation of a tool is a *very good* sign (Prometheus has one of these).

Recap

At the end of the day, backends **must receive requests and return responses**

The industry has gone through many paradigms and abstractions:

- C
- Java & Servlets
- Common Gateway Interface
- The LAMP stack
- Reverse Proxies
- Virtual Machines (VMs)
- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Containers

The future is still being written, and whether the writing looks fast or slow is (generally) a matter of perspective.

The End

Thanks for listening

whoami

If you've got any corrections, complaints, or comments, feel free to reach me using the information below:

Victor Adossi (vados@vadosware.io, vados@gaisma.co.jp)

GPG: ED874DE957CFB552

I run a couple very small consultancies to support businesses in Japan and the USA:

- GAISMA G.K. (https://gaisma.co.jp)
- VADOSWARE LLC (https://vadosware.io)

Need help getting your organization to the present/future? I can help with that.

Need help getting your organization to the past? I probably can't help with that.

Blooper Reel: Hot takes on interviewing (YMMV)

Interviews are tests of an employer as well as an employee.

Some things good companies do:

- Layered questions that increase in difficulty
- Questions at are very relevant to daily tasks
- Optional take-home assignments
- Train employees to be good interviewers
- Give feedback on interviews when possible
- Ask fizzbuzz

Some things bad companies do:

- Read your resume for the first time in the interview room
- Ask puzzle questions like how many X will fit in some Y
- Treat interviews like timed tests
- Ask questions completely unrelated to job responsibilities
- Never ask for feedback on their interview process
- Ask fizzbuzz