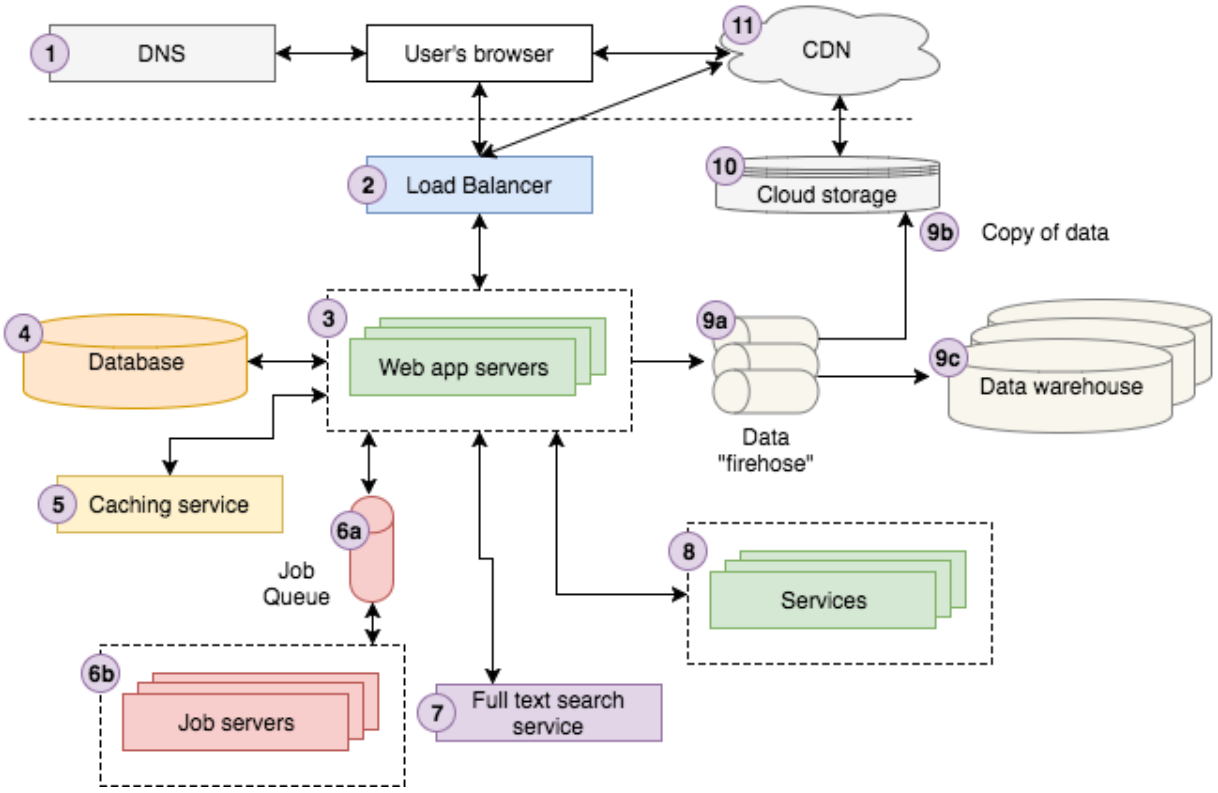
Modern Web Application Architecture



1. DNS

DNS stands for “Domain Name System” and it’s a backbone technology that makes the World Wide Web possible. At the most basic level DNS provides a key/value lookup from a domain name (e.g., google.com) to an IP address (e.g., 85.129.83.120), which is required in order for your computer to route a request to the appropriate server. Analogizing to phone numbers, the difference between a domain name and IP address is the difference between “call John Doe” and “call 201-867–5309.” Just like you needed a phone book to look up John’s number in the old days,

you need DNS to look up the IP address for a domain. So you can think of DNS as the phone book for the internet.

Before diving into details on load balancing, we need to take a step back to discuss horizontal vs. vertical application scaling. What are they and what's the difference? Horizontal scaling means that you scale by adding more machines into your pool of resources whereas "vertical" scaling means that you scale by adding more power (e.g., CPU, RAM) to an existing machine.

In web development, you (almost) always want to scale horizontally because, to keep it simple, stuff breaks. Servers crash randomly. Networks degrade. Entire data centers occasionally go offline. Having more than one server allows you to plan for outages so that your application continues running. In other words, your app is "fault tolerant." Secondly, horizontal scaling allows you to minimally couple different parts of your application backend (web server, database, service X, etc.) by having each of them run on different servers. Lastly, you may reach a scale where it's not possible to vertically scale any more. There is no computer in the world big enough to do all your app's computations. Think Google's search platform as a quintessential example though this applies to companies at much smaller scales. It would be challenging to provide that entire compute power via vertical scaling.

Ok, back to load balancers. They're the magic sauce that makes scaling horizontally possible. They route incoming requests to one of many application servers that are typically clones / mirror images of each other and send the response from the app server back to the client. Any one of them should process the request the same way so it's just a matter of distributing the requests across the set of servers so none of them are

overloaded. That's it. Conceptually load balancers are fairly straight forward.

3. Web Application Servers

At a high level web application servers are relatively simple to describe. They execute the core business logic that handles a user's request and sends back HTML to the user's browser. To do their job, they typically communicate with a variety of backend infrastructure such as databases, caching layers, job queues, search services, other microservices, data/logging queues, and more. As mentioned above, you typically have at least two and often times many more, plugged into a load balancer in order to process user requests.

You should know that app server implementations require choosing a specific language (Node.js, Ruby, PHP, Scala, Java, C# .NET, etc.) and a web MVC framework for that language (Express for Node.js, Ruby on Rails, Play for Scala, Laravel for PHP, etc.).

4. Database Servers

Every modern web application leverages one or more databases to store information. Databases provide ways of defining your data structures, inserting new data, finding existing data, updating or deleting existing data, performing computations across the data, and more. In most cases the web app servers talk directly to one, as will the job servers. Additionally, each backend service may have it's own database that's isolated from the rest of the application.

5. Caching Service

A caching service provides a simple key/value data store that makes it possible to save and lookup information in close to O(1) time. Applications typically leverage caching services to save the results of expensive computations so that it's possible to retrieve the results from the cache instead of recomputing them the next time they're needed. An application might cache results from a database query, calls to external services, HTML for a given URL, and many more. Here are some examples from real world applications:

- **Google** caches search results for common search queries like "dog" or "Taylor Swift" rather than re-computing them each time

- **Facebook** caches much of the data you see when you log in, such as post data, friends,

The two most widespread caching server technologies are Redis and Memcache.

6. Job Queue & Servers

Most web applications need to do some work asynchronously behind the scenes that's not directly associated with responding to a user's request. **For instance, Google needs to crawl and index the entire internet in order to return search results.** It does not do this every time you search. Instead, it crawls the web asynchronously, updating the search indexes along the way.

While there are different architectures that enable asynchronous work to be done, the most ubiquitous is what I'll call the "job queue" architecture. It consists of two components: a queue of "jobs" that need to be run and one or more job servers (often called "workers") that run the jobs in the queue.

Job queues store a list of jobs that need to be run asynchronously. The simplest are first-in-first-out (FIFO) queues though most applications end up needing some sort of priority queuing system. Whenever the app needs a job to be run, either on some sort of regular schedule or as determined by user actions, it simply adds the appropriate job to the queue.

We run jobs to encode videos and photos, process CSVs for metadata tagging, aggregate user statistics, send password reset emails, and more. Job servers process jobs. They poll the job queue to determine if there's work to do and if there is, they pop a job off the queue and execute it. The underlying languages and frameworks choices are as numerous as for web servers.

7. Full-text Search Service

Many if not most web apps support some sort of search feature where a user provides a text input (often called a "query") and the app returns the most "relevant" results. The technology powering this functionality is typically referred to as "full-text search", which leverages an inverted index to quickly look up documents that contain the query keywords.

Documents (Photo titles)	
id	title
1	Man running in the mountains
2	Mountains with snow
3	Man running marathon

Inverted Index	
keyword	photo_ids
man	1, 3
running	1, 3
mountains	1, 2
snow	2
marathon	3

Example showing how three document titles are converted into an inverted index to facilitate fast lookup from a specific keyword to the documents with that keyword in the title. Note, common words such as “in”, “the”, “with”, etc. (called stop words), are typically not included in an inverted index.

While it’s possible to do full-text search directly from some databases (e.g., MySQL supports full-text search), it’s typical to run a separate “search service” that computes and stores the inverted index and provides a query interface. The most popular full-text search platform today is Elasticsearch though there are other options such as Sphinx or Apache Solr.

8. Services

Once an app reaches a certain scale, there will likely be certain “services” that are carved out to run as separate applications. They’re not exposed to the external world but the app and other services interact with them.

- **Account service**
- **Content service**
- **Payment service**
- **HTML to PDF service**

9. Data

Today, companies live and die based on how well they harness data. Almost every app these days, once it reaches a certain scale, leverages a data pipeline to ensure that data can be collected, stored, and analyzed. A typical pipeline has three main stages:

1. The app sends data, typically events about user interactions, to the data "firehose" which provides a streaming interface to ingest and process the data. Often times the raw data is transformed or augmented and passed to another firehose. AWS Kinesis and Kafka are the two most common technologies for this purpose.

2. The raw data as well as the final transformed/augmented data are saved to cloud storage. AWS Kinesis provides a setting called "firehose" that makes saving the raw data to it's cloud storage (S3) extremely easy to configure.

3. The transformed/augmented data is often loaded into a data warehouse for analysis. AWS Redshift, as does a large and growing portion of the startup world, though larger companies will often use Oracle or other proprietary warehouse technologies. If the data sets are large enough, a Hadoop-like NoSQL MapReduce technology may be required for analysis.

10. Cloud storage

"Cloud storage is a simple and scalable way to store, access, and share data over the Internet" according to AWS. You can use it to store and access more or less anything you'd store on a local file system with the

benefits of being able to interact with it via a RESTful API over HTTP. Amazon's S3 offering is by far the most popular cloud storage available today.

11. CDN

CDN stands for "Content Delivery Network" and the technology provides a way of serving assets such as static HTML, CSS, Javascript, and images over the web much faster than serving them from a single origin server. It works by distributing the content across many "edge" servers around the world so that users end up downloading assets from the "edge" servers instead of the origin server. For instance in the image below, a user in Spain requests a web page from a site with origin servers in NYC, but the static assets for the page are loaded from a CDN "edge" server in England, preventing many slow cross-Atlantic HTTP requests.

In general now days a web app should always use a CDN to serve CSS, Javascript, images, videos and any other assets. Some apps might also be able to leverage a CDN to serve static HTML pages.