A Web-Based Mapping Service Like Google Maps

In this lesson, we will discuss a case study for a web-based mapping service like Google Maps.

We'll cover the following

Before I begin talking about the service's architecture, I would like to state that this is not a system design lesson because it doesn't contain any of the database design, traffic estimations, or code of any sort.

I will just discuss the basic architectural aspects of the service and how the concepts we've learned in the course apply here.

Let's get on with it.

A little background on Google Maps#

Google Maps (https://cloud.google.com/maps-platform/) is a web-based mapping service by *Google*. It offers satellite imagery, route planning features, real-time traffic conditions, an API for writing map-based games like *Pokémon Go*, and several other features.

First up, these massive and successful services are a result of years of evolution and iterative development. Online services are built feature by feature and take years to perfect. *Google Maps* started as a desktop-based software written in *C++* and evolved over the years to become what it is today, a beautiful mapping service used by over a billion users.

Read-heavy application#

Let's get down to the technicalities of it. An application like this is *read-heavy* and not *write-heavy*. Since the end-users aren't generating new content in the application over time, users do perform some write operations though it is negligible in comparison to a write-heavy application like *Twitter* or *Instagram*. This means the data can be largely cached and there will be significantly less load on the database.

Data Type: Spatial#

Speaking of data, a mapping application like this has *spatial data*. *Spatial data* is the data with objects representing geometric information like points, lines, polygons. The data also contains alphanumeric information,

like *Geohash (https://en.wikipedia.org/wiki/Geohash)*, latitudes, longitudes, *GIS Geographical Information System* data, etc.

There are dedicated *spatial databases* available for persisting this kind of data. Popular databases like *MySQL*, *MongoDB (https://docs.mongodb.com/manual/core/geospatial-indexes/)*, *CouchDB (https://github.com/couchbase/geocouch/)*, *Neo4J*, *Redis (https://github.com/EverythingMe/geodis)*, and *Google Big Query GIS (https://cloud.google.com/bigquery/docs/gis-intro)* also support persistence of *spatial data*. They have additional plugins built for it.

If you want to read more about spatial databases, this is a good read (https://en.wikipedia.org/wiki/Spatial_database).

Database#

The coordinates of the places are persisted in the database. When the user runs a search for a specific location, the coordinates are fetched from the database, and the numbers are converted into a map image.

We can expect a surge in traffic on the service during peak office hours or festivals or major events in the city. We need dynamic *horizontal scalability*, to manage these traffic spikes. The app needs to be *elastic* to scale up and down on the fly.

As I mentioned earlier, we have the option of picking from multiple databases as both *relational* and *non-relational* support persistence of *spatial data*. I am more inclined to pick a *non-relational NoSQL* one because the map data doesn't contain many relationships. It directly fetches the coordinates and processes them based on the user request. Also, a *NoSQL* database is inherently *horizontally scalable*.

Though, we can also scale well with a *relational database* with *caching* because the application is read-heavy. However, in *real-time* use cases with a lot of updates, it will be a bit of a challenge.

Real-time features like *LIVE* traffic patterns, information on congested routes, and the alternative routes suggestions as we drive in real-time, etc. are pretty popular with the *Google Maps* users.

Architecture#

Naturally, to set up a service like this we will pick a *client-server* architecture as we need control over the service. Otherwise, we could have thought about the *P2P* architecture, but *P2P* won't do us any good here.

Backend technology#

Speaking of the server-side language we can pick *Java*, *Scala*, *Python*, and *Go*. Any of the mature backend technology stacks will do. My personal pick will be *Java*, because it is performant and heavily used for writing scalable distributed systems, and for the enterprise development.

Monolith vs microservice#

Speaking of *monolithic architecture* vs *microservice*, which one do you think we should pick to write the app?

Let's figure this out by going through the features of the service. The core feature is the map search. The service also enables us to plan our routes based on different modes of travel, including cars, walking, cycling, etc.

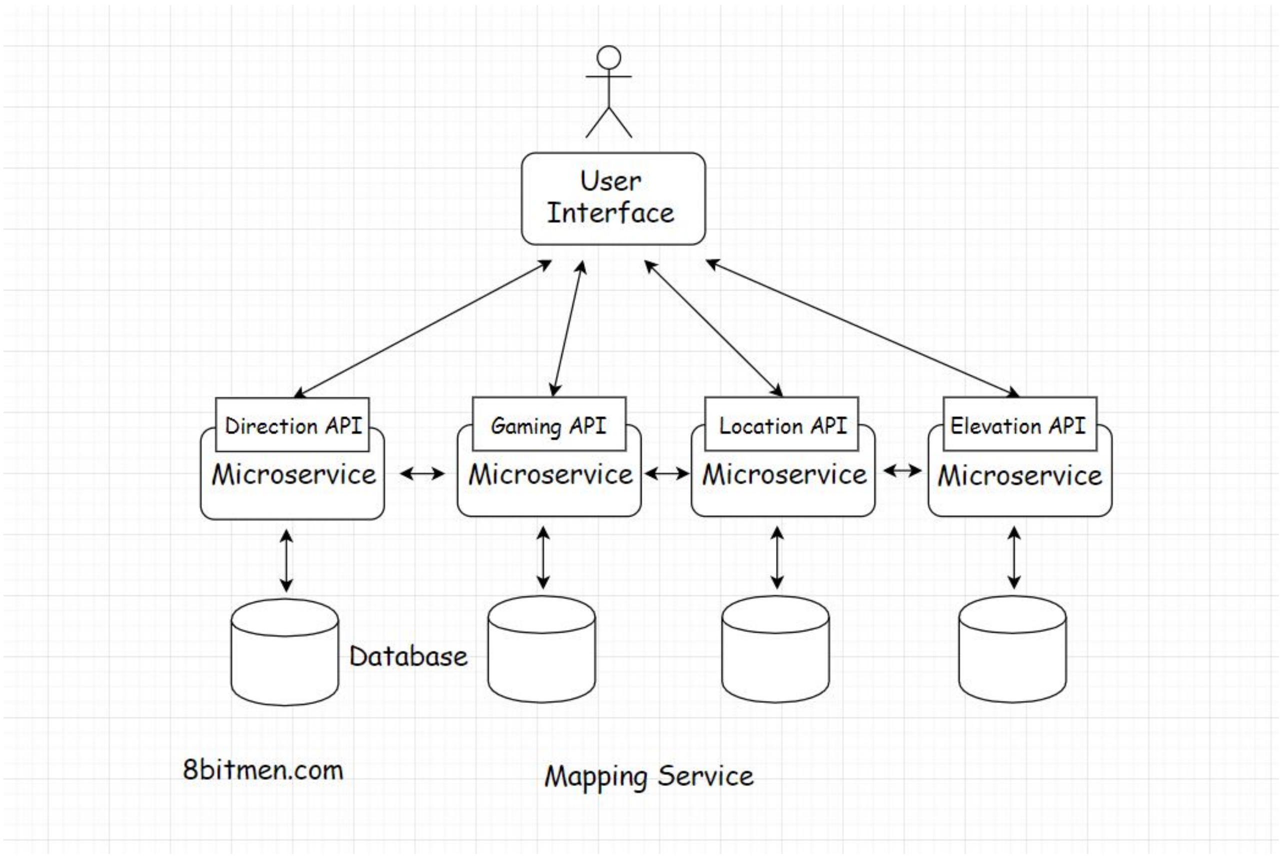
Once our trip starts, the map offers alternative route locations in real-time. The service adjusts the map based on the user's real-time location and the destinations.

APIs#

For the third-party developers, *Google* offers different *APIs* such as the *Direction API*, *Distance Matrix*, *Geocoding*, *Places*, *Roads*, *Elevation*, *Time zone*, and *Custom search API*.

The *Distance Matrix API* tells us how much time it will take to reach a destination depending on the mode of travel: walking, flying, or driving. *Real-time* alternative routes are displayed with the help of predictive modelling based on machine learning algorithms. The *Geocoding API* is about converting numbers into actual places and vice versa.

Google Maps also has a *Gaming API* for building map-based games.



We may not have to implement everything in the first release, but this gives us a clue that *monolithic architecture* is totally out of the picture.

We need *microservices* to implement so many different functionalities. Let's write a separate service for every feature. This is a cleaner approach, and it helps the service scale and stay highly available. If a few services

like real-time traffic, elevation API, etc. go down, the core search remains unaffected.

Server-side rendering of map tiles#

Speaking of the core location search service, when the user searches for a specific location, the service has to match the search text with the name of the location in the database and pull up the place's coordinates.

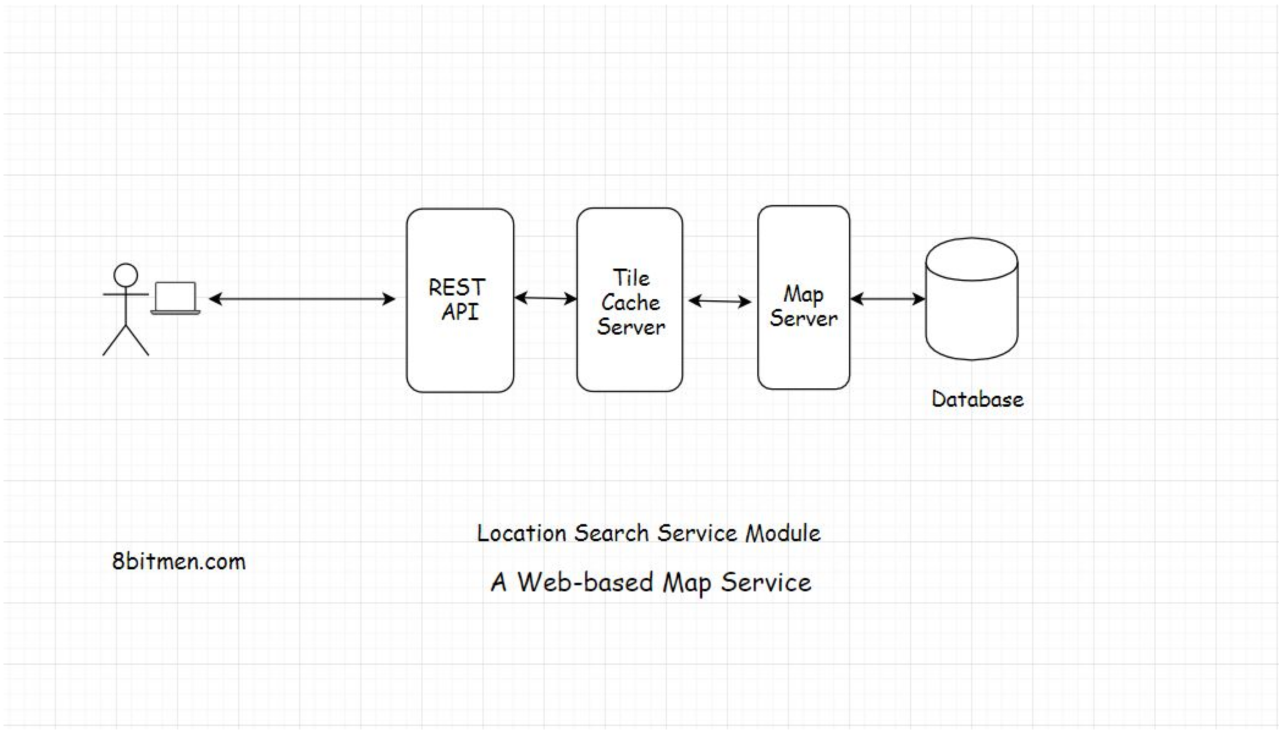
Once the service has the coordinates how do we convert those into an image? Also, should we render the image on the client or the server?

Server-side rendering is a preferable option in this scenario because we can cache the rendered image for future requests. The image is a kind of static content and will be the same for all the users.

Also, as opposed to generating a single map image for the full web page, the entire map is broken down into tiles that enable the system to generate only the part of the map that the user engages with.

Smaller tiles help with the zoom in and out operations. You might have noticed this when using *Google Maps*. Instead of the entire web page being refreshed, the map is refreshed in sections or tiles. Rendering the entire map instead of tiles every time would be very resource intensive.

We can create the map in advance by rendering it on the server and caching the tiles. Also, we need a dedicated map server to render the tiles on the backend.



User Interface#

Speaking of the *UI*, we can write this using *JavaScript*, *Html5*. Simple *JavaScript*, *Jquery* serves me well for simple requirements. However, if you want to leverage a framework, you can look into *React*, *Angular*, etc.

The *UI* having *JavaScript* events enables the user to interact with the map, pin locations, search for places, draw markers and other vectors on the map, etc.

OpenLayers (https://openlayers.org/) is a popular open-source *UI* library for making maps work with web browsers. You can leverage it if you do not want to write everything from the ground up.

Okay!! So, the user runs a search for a location, on the backend, and the request is routed to the tile cache, which has all the pre-generated tiles. It sits between the UI and the map server. If the requested tile is present in the cache it is sent to the UI. If not, the map server hits the database, fetches the coordinates and related data, and generates the tile.

Real-time features#

Let's move onto to the *real-time* features. To implement real-time features, we have to establish a *persistent connection* with the server. We've gone through the *persistent connections* in detail in the course.

Although *real-time* features are cool, they are very resource intensive. There is a limit to the number of *concurrent connections* servers can handle. So, I'll advise implementing real-time features only when it's really required.

This is a good read on the topic. How Hotstar a video streaming service scaled with over 10 million concurrent users (https://www.8bitmen.com/how-hotstar-scaled-with-10-3-million-concurrent-users-an-architectural-insight/)

Well, this is pretty much it for web-based mapping services. We've covered the backend, database, caching, and the UI, and you should have a fundamental understanding of how a service like *Google Maps* works.

I'll see you in the next lesson, where we will discuss a baseball game online ticket booking service.

← Back

Key Things to Remember When Picki...

Next →

A Baseball Game Ticket Booking Web...

✓ Mark as Completed

Report an Issue