A Web-based Mapping Service Like Google Maps

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

We'll cover the following

Before I begin talking about the architecture of the service, I would like to state that this is not a system design lesson, as 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 do the concepts we've learned in the course apply here.

Let's get on with it.

A Little Background On Google Maps

Google Maps is a web-based mapping service by *Google*. If offers satellite imagery, route planning features, real-time traffic conditions, an API for writing map-based games like *Pokemon Go* & several other features.

First up, these massive 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++* & 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* & not *write-heavy*. As 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 the 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 stuff like *Geohash*, latitudes, longitudes, *GIS Geographical Information System* data etc.

There are dedicated *spatial databases* available for persisting this kind of data. Also, popular databases like *MySQL*, *MongoDB*, *CouchDB*, *Neo4J*, *Redis*, *Google Big Query GIS* 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.

Database

The coordinates of the places are persisted in the database and when the user runs a search for a specific location the co-ordinates are fetched from the database & the numbers are converted into a map image.

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

As I brought this up earlier, we have the option of picking from multiple databases as both *relational* and *non-relational* support persistence of *spatial data*. I will be more inclined to pick a *non-relational NoSQL* one as the map data doesn't contain many relationships. It's a direct fetch of the co-ordinates & the processing of them based on the user request. Also, a *NoSQL* database is inherently *horizontally scalable*.

Though, we can scale well with a *relational database* too with *caching* as the application is read-heavy. But 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, the suggestion of alternative routes as we drive in real-time etc. are pretty popular with the users of *Google Maps*.

Architecture

Naturally, to set up a service like this we will pick a *client-server* architecture as we need control over the service. Else 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*, *Go*. Any of the mature backend technology stack will do. My personal pick will be *Java*, since it is performant, heavily used for writing scalable distributed systems, as well as for the enterprise development.

Monolith Vs Microservice

Speaking of *monolithic architecture* vs *microservice*, which one do you think we should pick for writing 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, by car, walking, cycling etc.

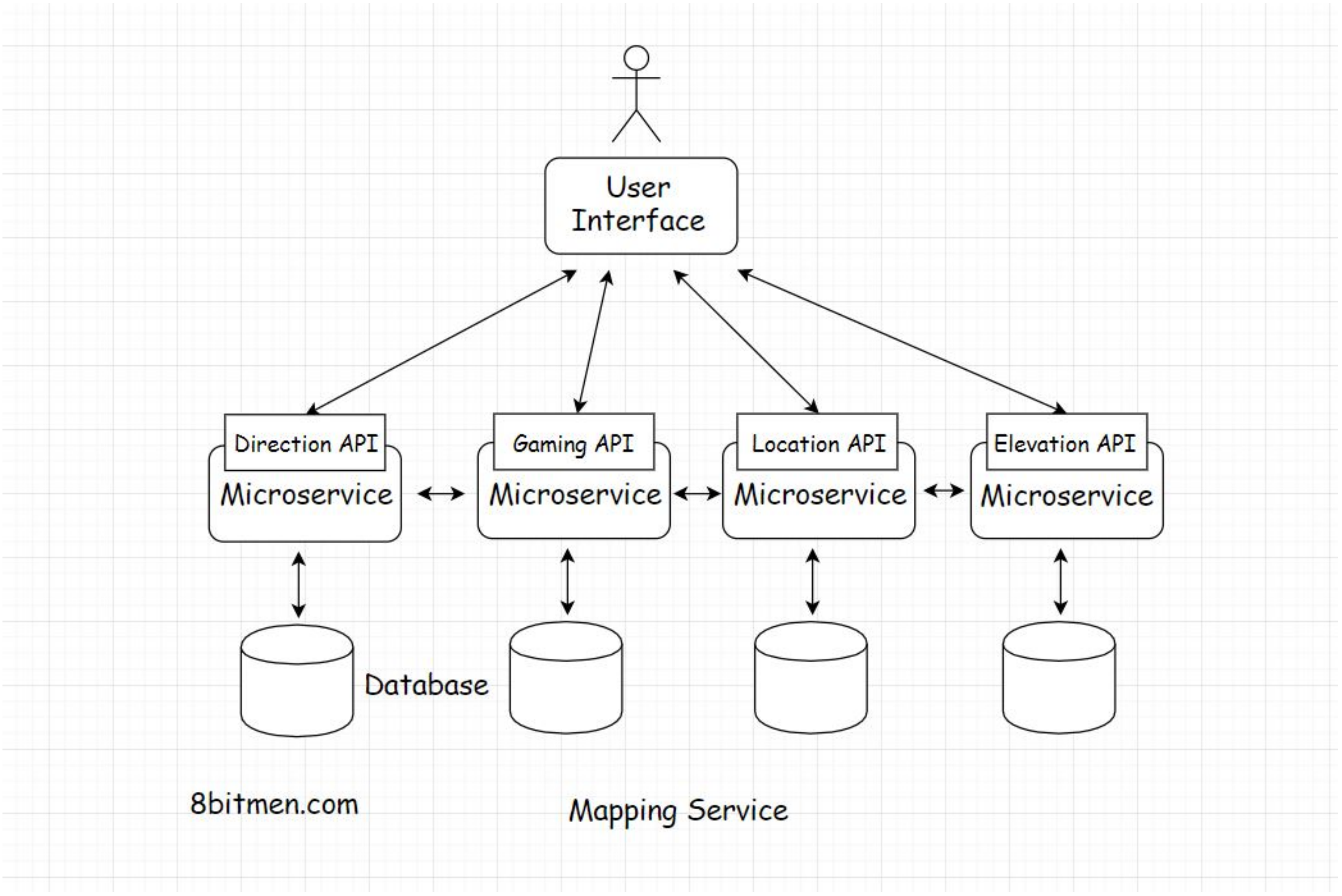
Once our trip starts, the map offers alternative route locations in real-time. The service adjusts the map based on the user's real-time location & the destination.

APIs

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

The *Distance matrix API* tells us how much time will it take to reach a destination depending on the mode of travel walking, flying, 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 & vice versa.

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



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

We need *microservices* to implement so many different functionalities. Write a separate service for every feature. This is a cleaner approach, 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 coordinates of the place.

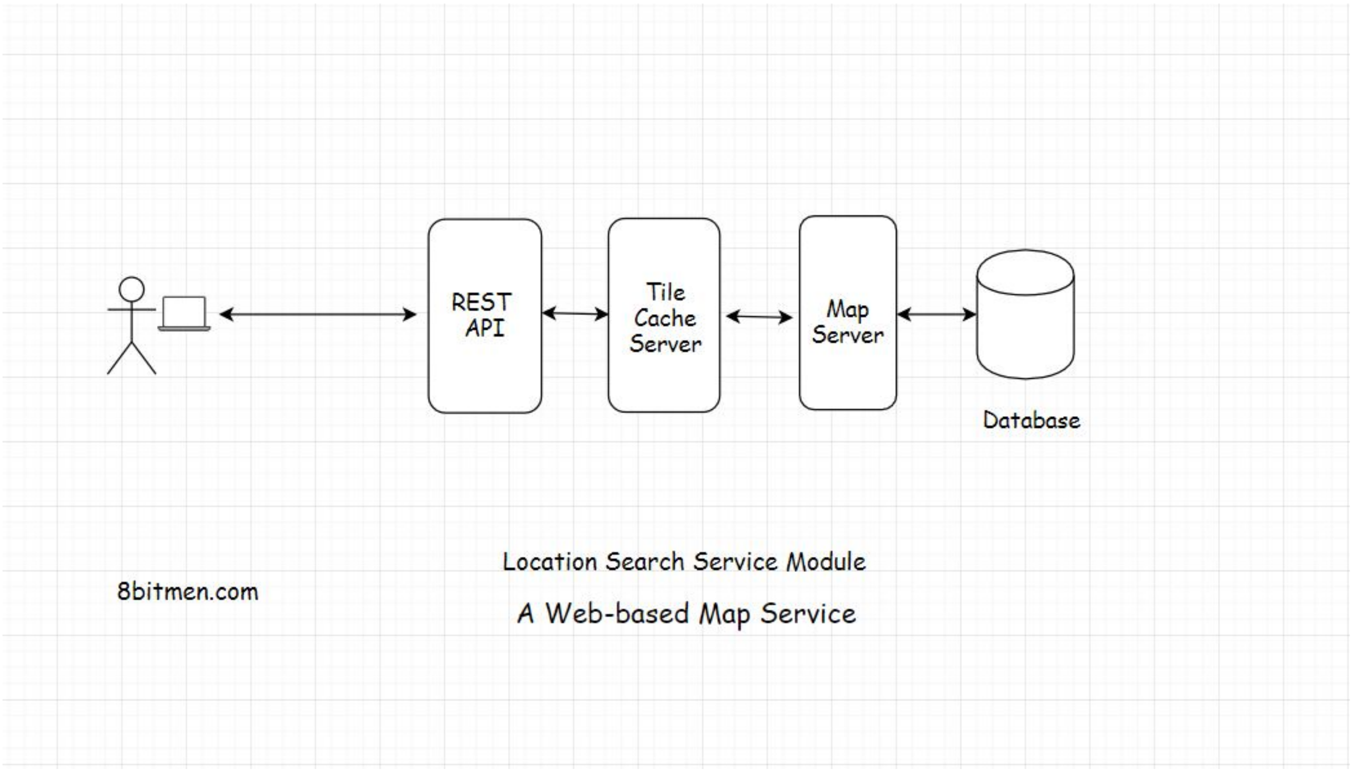
Once the service has the co-ordinates how do we convert those into an image? Also, should we render the image on the client or the server?

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

Also, as opposed to generating a single image of the map 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 user engages with.

Smaller tiles help with the zoom in & out operations. You might have noticed this when using *Google Maps*, that 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 & 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 that using *JavaScript*, *Html5*. Simple *JavaScript*, *Jquery* serves me well for simple requirements. But if you want to leverage a framework, you can look into *React*, *Angular* etc.

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

OpenLayers 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, the request is routed to the tile cache. The 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 co-ordinates and related data & generates the tile.

Real-time Features

Coming to the *real-time* features. To implement it we have to establish a *persistent connection* with the server. We've been through the *persistent connections* in detail in the course.

Though *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

Well, this is pretty much it on a web-based mapping service. We've covered the backend, database, caching and the UI & 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.