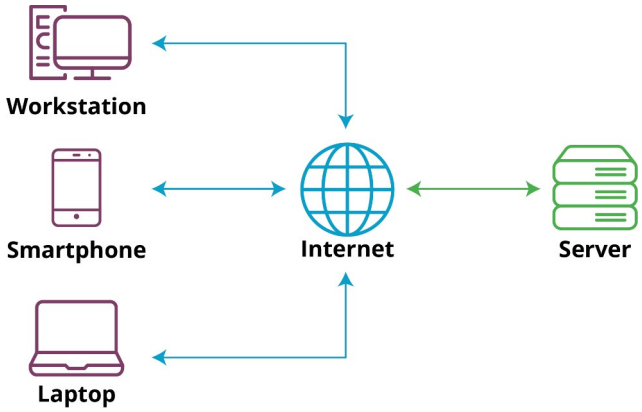
The architecture of a slack-like system

Slack is basically a messenger that focuses on business communications. It has the ability to create a designated space for a given company and supports many group chats aka channels. As it must have many clients, it should be implemented as a client-server system.



First, let's look at the client side. We definitely should have a multi-platform mobile app client, as this is how most people exchange messages these days. Then, a web client is another must, as many people would probably use such an app for work and it's more convenient to have everything work-related on their laptop. Finally, we could have a desktop app, as some people still prefer it to using a web site, but it's optional. With modern technologies though, the desktop app might be not so different from a web application.

Now, let's have a closer look at a server side. A "server" doesn't obviously mean a single server, as but rather a cluster of servers in the cloud. For companies that do not want their employees to store sensitive communication on the web, we can provide an on-premise solution – a server application that can be deployed in a customer's environment and to which all supported clients can address the same way as to main server in the cloud.

For the server side, we will use a popular three-tier architecture, in which the presentation layer will be a mobile app or a web/desktop UI module, a business layer will be the main application, and a data layer will be a database that will store users' messages and some configuration. For business logic level, we will apply a modern microservices architecture that allow for independent development of different functional modules and easier scaling.

First, we will need a microservice for user management. It will allow users to sign up and sign in. It also have a role system, and administrators will be able to create user accounts for their companies.

Then, we will need some kind of administration microservice. It will allow users to create a company, to create a channel, and to make other configurations.

Now, let's move on to the microservices that will provide the main functions of the application. We will need a microservice for processing new messages from users (Post microservice) and a microservice that will form the users' message feed (Feed microservice). Post microservice should support posting a message directly to another user, or to channel that given user is a member of. The message should also support different attachments, like images, documents, videos etc. Feed microservice should return the latest messages addressed to this user, or published into the channels this user is a member of.

The services may exchange information via a message broker, enabling event-driven architecture, e.g. when a new message is posted, we may raise an event “message posted”, so that Feed service can update the feed for everyone who is supposed to see this message.

All main services must provided an API, so the developers can make different integrations useful for them (e.g. they could make a Gitlab plugin that will push notifications of build executions to a specific channel). All API communication to our microservices will be done through API Gateway that applies security, service resolution etc.

Finally, let’s look at the data tier. As we mostly need to store messages with rather simple relations to other entities, we will benefit from NoSql databases, like MongoDb, rather than typical relational databases.

Below there is a final architecture schema of the system.