slack engineering (https://slack.engineering/)

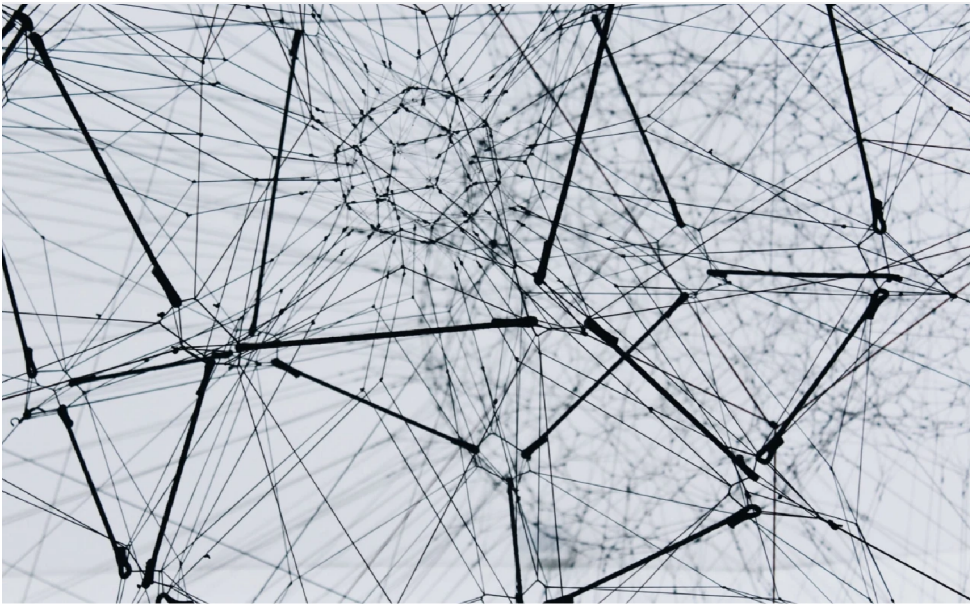


Photo by Alina Grubnyak

How Slack Built Shared Channels

Building shared channels challenged Slack's fundamental assumption that the workspace is the atomic unit of partitioning customer data.

Yingyu Sun backend on network

Mike Demmer Principal Engineer

11 minutes • Written 1 year ago

Written with contributions from the Shared Channels Team.

Slack was originally built to be the collaboration hub for the work *within* your company. As the network of companies using Slack for internal work grew, we saw the value of allowing different companies to collaborate *together* in one channel.

We're now making shared channels (https://slackhq.com/slack-shared-channels) available to all customers! A shared channel is one that connects two separate organizations. You no longer need to go back and forth between external emails and internal Slack channels, or provision an endless number of guests into your Slack workspace: A shared channel creates one productive space for people from both companies to send messages, share files, and work together in Slack.

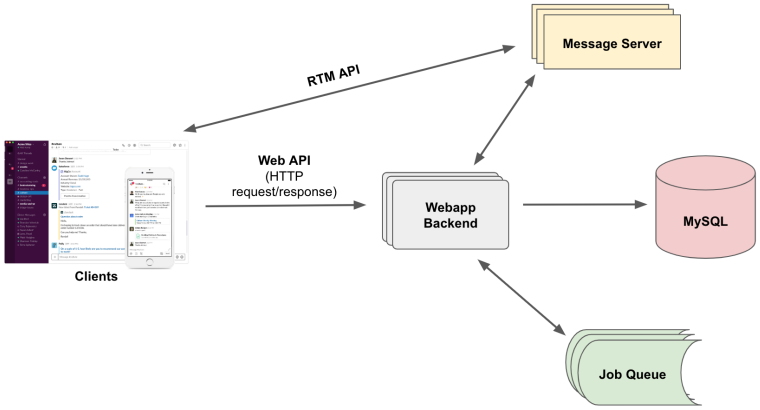
The idea of shared channels challenged Slack's fundamental assumption that the workspace is the atomic unit of partitioning customer data, however. For this post, we'll discuss a few of the most interesting technical challenges from this change. We'll talk about our initial architecture, the decisions that went into reframing how Slack worked, the implications of those decisions, and how we're working on scaling for the future.

Slack before shared channels

Before we dive into shared channels, let's take a look at how Slack works at a high level. Slack implements a client-server architecture where clients (mobile, desktop, web and apps) talk to two backend systems:

1. *webapp servers*, which handle HTTP request/response cycles and communicate with other systems like the main databases and job queues
2. *real-time message servers*, which fan out messages, profile changes, user presence updates, and a host of other events to clients

For example, a new message posted in a channel is sent via an API call to the webapp servers, where it is sent to the message servers and is persisted in the database. The real-time message servers receive these new messages and send them to connected clients over web sockets, based on who is in the channel.



Basic Slack architecture

From the time Slack was launched in 2014, this core system architecture centered around the concept of a “workspace.” Logically, workspaces enveloped all other objects in the system, including users, channels, messages, files, emoji, apps and more. They also provided the administrative and security boundary to implement access controls, as well as policy and visibility preferences.

The backend systems used the boundaries of the workspace as a convenient way to scale the service, by spreading out load among sharded systems. Specifically, when a workspace was created, it was assigned to a specific database shard, messaging server shard, and search service shard. This design allowed Slack to scale horizontally and onboard more customers by adding more server capacity and putting new workspaces onto new servers. The design also meant that application code interacting

with a workspace's content needed to first determine which workspace the request was for, and then use that context to route to a particular database or other server shard. **In essence, the workspace was the unit of tenancy in the multi-tenant service, used for data partitioning and segmentation.**

This design was very simple and effective: To find data or send messages, our code only needed to look up the appropriate workspace's shard and route the request there to verify that the requesting user had access to the given channel. Over the years, more and more application code and services were developed around the core assumption that data lived within a workspace, further solidifying this walled boundary.

Of course, introducing shared channels meant that we needed to rethink these assumptions, since shared channels enable users to access some channels, messages and files *across workspace boundaries*. This required us to re-examine the basic access control, visibility, and data sharding capabilities on which Slack had been built, and we discovered a number of interesting design challenges and trade-offs.

Architecture of shared channels

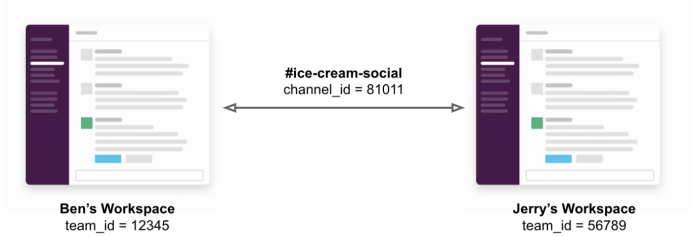
Shared channels required us to look at how messages should flow across workspaces. We needed to decide how to distribute and store messages so that members of both workspaces could join the channel, send messages, and use Slack normally.

We considered a number of different approaches. The first was to continue down the path we had started by assuming that everything that users interact with on Slack belongs to the user's workspace. This design would mean that both workspaces would have a copy of the shared channel in their respective database shard, and messages would be written to both the sending user's shard and the other side's shard. As we had done before, we wouldn't need to query multiple databases or worry about routing requests to the other workspace's shard.

Though simple, this option had a lot of downsides. Considering more than a billion messages (https://www.sec.gov/Archives/edgar/data/1764925/000162828019007428/slacks-1a3.htm#sC9C346D78943772D97FB567D8BF6BBDD) are sent on Slack weekly, duplicating data for both workspaces would restrict our potential to scale shared channels. This wasn't limited to just messages—it also included pins, reactions and other channel-specific information. We also wanted writing data to be as real-time and consistent as possible, but writing to

multiple databases and multiple real-time message servers risked having inconsistent write times, leading to inconsistent channel histories.

Instead, we went with an approach that was more consistent and performant and had more potential to scale. We decided to have one copy of the shared channel data and instead route read and write requests to the single shard that hosts a given channel. We used a new database table called shared_channels as a bridge to connect workspaces in a shared channel.



channels

id	workspace_id	name	topic	is_private
81011	12345	ice-cream-social	Friends are welcome to attend!	0

shared_channels

channel_id	workspace_id	target_workspace_id	originating_workspace_id	name	topic	is_private
81011	12345	56789	12345			
81011	56789	12345	12345	ice-cream-social	Remember to RSVP!	1

Every channel in Slack (including shared and non-shared channels) has a single channels row that lives on the originating workspace's shard. In our example, Ben's workspace originated the shared channel, so

the channel would be written to the channels table on its workspace shard. All content in the given channel (messages, reactions, pins, and so on) is also written to that workspace shard alongside the channels table.

For a shared channel, there are two additional shared_channels rows: one row for each workspace, pointing to the other as the target workspace. These rows include the channel ID, target workspace ID, originating workspace ID, and overrides for certain channel properties. We can look at either workspace's database shard to know who it has shared channels with and determine which side initiated each channel. Additionally, each workspace can have different views of the channel. The originating workspace fetches channel details, like name, topic and privacy setting, from its row in the channels table as it does with any other channel. For the target workspace (in this example, Jerry's workspace), these fields are overridden from the properties in its shared_channels row, which allows each workspace to set its own name, topic and purpose for the channel.

Breaking the pattern by not storing channel data on a workspace's shard required a lot of work—the messaging, database and API layers of our application needed to be made aware of the fact that channel data would sometimes need to be fetched from other shards, and that channel properties might

need to be overridden from the shared_channels rows.

Implications and challenges

Channel privacy

Since we went ahead with having one copy of a shared channel, we had to reconsider our implementation of channel privacy. Before shared channels, we used ID prefixes to distinguish public versus private channels. We stored private channels in the groups table with a G-encoded ID and public channels in the channels table with a C-encoded ID. This ensured that we had no mix of data between public and private channels and that there was no confusion about visibility based on its encoding. **With shared channels, we wanted to allow the individual workspaces to decide the privacy of a shared channel on their own side.** Given our design, in which we would have a single channel with a single ID shared on either side, we needed to decouple channel privacy from the table on which the channel lived and the prefix with which the channel ID was encoded.

To accomplish this goal, we decided that all shared channels would be created as C-encoded channels and stored on the channels table. This presented a consistent channel ID to the message server and all clients. Since an encoded channel ID no longer indicated whether the channel was public or private, we added an explicit boolean is_private flag that made it quick and easy to check if any channel was public or private.

This flag was the first step in a large effort to consolidate logic around different channel types. Before shared channels, the application code largely followed the original database pattern, with distinct (but often duplicative) logic for channels, groups and direct messages. These parallel code paths added greater complexity to each feature or performance improvement that touched any channel-like object. Enabled by the decision to decouple privacy from the database and ID logic, we were able to consolidate the database structure and application logic around a single channel-object model. This decision has continued to pay dividends as the product becomes more complex, channels become more powerful, and our architecture continues to scale.

To take advantage of these improvements, however, clients, who could once assume that all C-encoded channels would be public, now needed to correctly process this new channel object model when determining channel privacy. To make these changes

compatible with pre-existing clients and third-party app developers, we introduced the conversations.* API to replace the old channels.* and groups.* endpoints. The new API can take as input any channel-like objects and is able to appropriately switch between them.

User visibility

Since shared channels allow users to interact with members of other workspaces, one of the most significant problems was determining when and how different users could interact.

Prior to shared channels, a call to the users.info API method was straightforward: Given a user ID, we queried if that user belonged on the same workspace as the caller. If so, the API returned their profile, and if not, it returned an error indicating that the user could not be accessed. With shared channels, user visibility needed to expand so that profiles could be visible across the workspace boundaries. Furthermore, when a user on one workspace wanted to view an external user, both the content of the resulting profile and whether the two users could interact would depend on whether or not the users were in a shared channel together.

This required a few changes in Flannel, our edge cache service that we use to allow clients to load

minimal data on startup. (You can read more about Flannel in this blog post (/flannel-an-application-level-edge-cache-to-make-slack-scale).) At the time, Flannel did not support the concept of external users, since it assumed that users would interact only with members of the same workspace. This meant that when an external user was mentioned or sent a message in a shared channel, Flannel would be unable to find the user and fall back to our web API to fetch the user. Depending on the size of the channel, simultaneous client actions ran the risk of overwhelming the back end and bringing down Slack.

The Flannel team worked to extend its workspace model to support user objects and visibility rules for shared channels. When a new user loaded Slack, Flannel would load information about all of the shared channels that the user was a member of, including external user profiles. This meant that Flannel also needed to subscribe to any presence changes for external users as well as channel membership events, such as channel_member_join and channel_member_leave, in order to keep its model updated.

Building for large organizations

Our customers continue to reshape the way Slack is used in the workplace, and shared channels are no

exception. It's a race to scale shared channels to fit the needs of our largest customers, some of which have upward of 160,000 active users and more than 5,000 shared channels.

The scalability of shared channels depends on (among other things) Flannel, to avoid overload. In many cases also related to user visibility, however, clients can't completely rely on Flannel to efficiently cache updates. Falling back to our web API unveils several O(n) operations in which our APIs iterate over a group of entities (users, workspaces, channels, etc.) to determine things like the number of users in a channel and the number of external workspaces a user can see based on their shared channel membership. Since shared channels continue to become the place where work happens across organizations, we've worked to make sure that these O(n) operations are either efficiently cached or eliminated altogether.

We've also been working to close the gaps in shared channels for our large enterprise customers, specifically around data governance. These include policies like data retention, in which messages and files are retained for only a selected period of time and then automatically deleted. Another challenge is to make our Enterprise Key Management (/engineering-dive-into-slack-enterprise-key-management) feature, in which each workspace in

the channel can bring their own encryption keys to secure their own data, work for shared channels.

What's next for shared channels?

We've gone through a few of the challenges we faced when building this strategically important feature in Slack. Shared channels will gradually redefine what Slack is and how people choose to use it. Over time, we will continue to build upon shared channels and explore ways to make them a place where work happens for cross-organization collaboration of all shapes and sizes.

Expanding beyond the single-workspace model was just the first step. We imagine that someday, our customers will want to share a channel with as many external workspaces as they need. The value of shared channels will grow as we collaborate with more and more workspaces to get work done.

While we've made significant changes to our existing architecture, there is still plenty of work to do to evolve our data model as we continue to build the network. If you found these technical challenges interesting, you can also join our network of employees (https://slack.com/careers) at Slack!

#database #performance #php #software-architecture