WHITE PAPER

Making Session Stores more Intelligent with Microservices

Kyle Davis, Technical Marketing Manager, Redis

CONTENTS

What is a session store? 2

What is an intelligent session store? 2

Why does any service need a session store? 2

Why a microservice? 4

Transport Mechanisms 5

From submitted to calculated data 6

Bloom Filters 7

HyperLogLog 7

Bit Counting 7

Session Store Patterns 8

Content Surfacing 8

Activity Pattern Monitoring & Personalization 9

Group Notifications 10

Putting it all together 11

Conclusion 13

What is a session store?

Simply put, a session store is a “chunk” of data that is connected to a user of a service, stored separate from the primary database in order to provide stickiness without direct, constant access to the database. “User,” in this case, is loosely defined. A user could be as simple as a mere visitor to a webpage, a user with an account in a phone app or even another service that accesses data via an API.

A session is often persisted between requests through a cookie. The server gives the client the cookie and the client stores it, then sends subsequent request back with this cookie. The server then uses the cookie string as a token for which data can be associated with the user.

Often times, the session data is the most frequently used by a single user (and that user alone). The session data is also often a critical requirement for rendering a page or view.

Many times, session data is ephemeral and duplicated in some other data store. However, as we will explore in this document, this doesn’t always have to be true.

What is an intelligent session store?

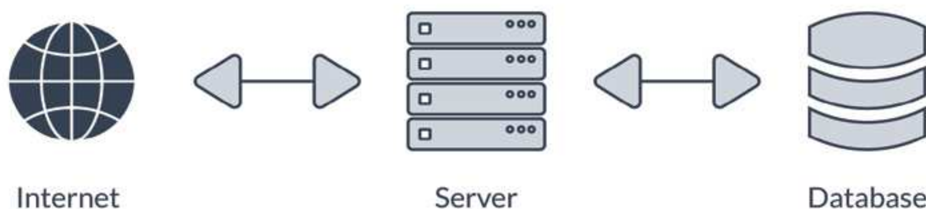
In this document we will be exploring session stores that move beyond “dumb” stores of data. This intelligent data in a session store might be calculated, inferred or otherwise not directly supplied by the user of the service. In this way, you can store the traditional session data (username, preferences or other common stateful data) alongside intelligent data.

Some examples of the intelligent data we’ll be using are:

- Group notifications - providing a single notification to a specific slice of users
- Content surfacing data - a dataset that can be leveraged for users to be pointed to additional content
- Activity data - information about the users’ behavior and usage of the service
- Personalization data - data that can be used to make the service more specific and relevant to each user

Why does any service need a session store?

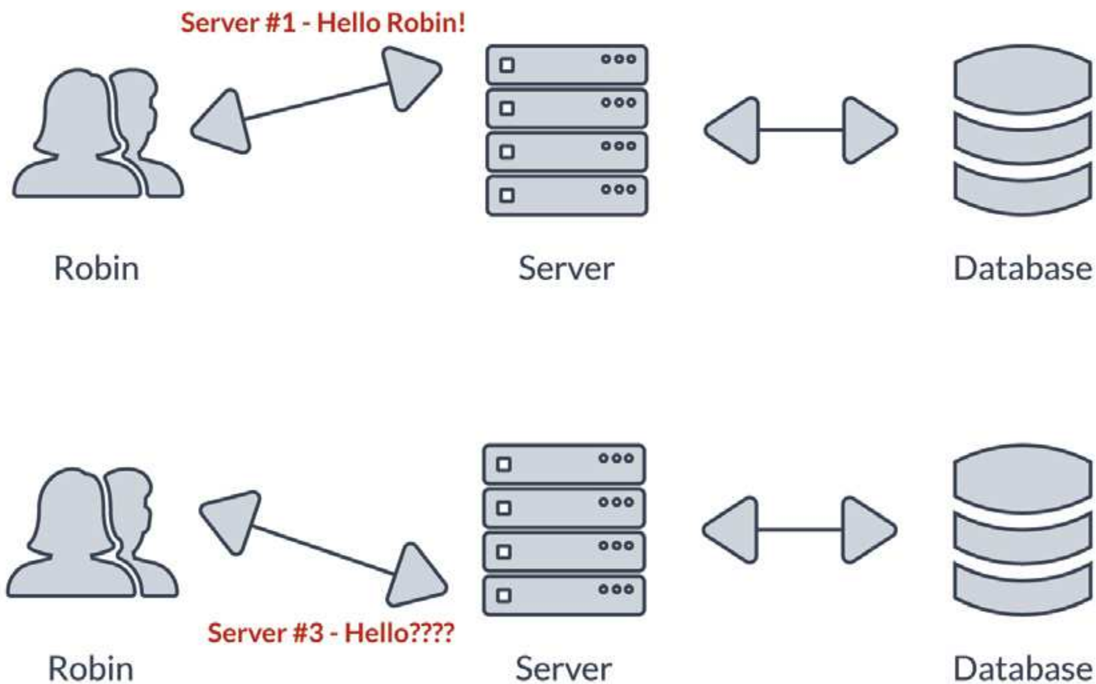
In a very simple world, one does not need a session store. In this very simple example, you have the entire internet connected to your single server, that is then backed by a single database. On every page view, the web server connects to the database and grabs the requisite information.



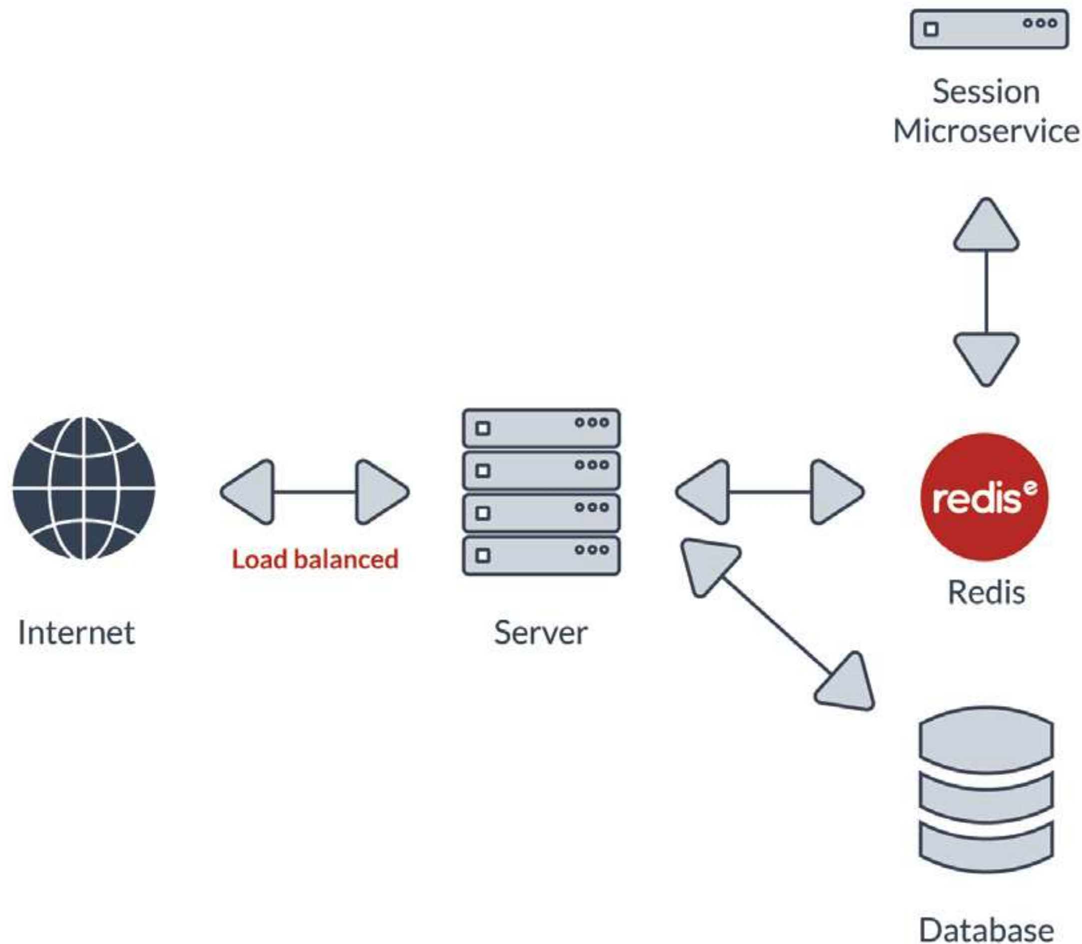
This might work great for small use cases, but when your website becomes busy, you’re likely to start having problems with the database becoming slow. After all, querying—or worse,writing to most databases is very resource intensive as compared to the duties of a web server. To remedy this situation, you alter your website code to start using a file-based session store. This is the simplest of the session store strategies—effectively, individual sessions are stored in text files that reside on your webserver. The web server software would then read or manipulate the session data file directly.

This helps a great deal because the database is much less frequently accessed, and thus is much quicker. However, when your website grows even more, another problem arises. At this point, you've limited the number of requests being made to the database, but with all the file I/O, in addition to the normal duties of a web server, the web server itself starts to strain under the pressure of your popular site. The common solution to this conundrum is to add more web servers and a load balancer. The load balancer distributes traffic evenly to all web server boxes.

This solution might work well during testing, but actual users will probably have complaints. Let's examine what happens when a user with a session that stores their name gets directed to two different web servers with a file-based session store.



As the graphic above illustrates, when the web server array is storing session data on a file and the load balancer just distributes to the next web server, the session data may be stored on a different server in a text file. As a result, the service doesn't know that the user is called Robin. Let's explore a better solution.



In this example, we have a web server that stores the session data in Redis rather than an on-server text file. Also included is a session microservice that can manage our session's intelligent session data. Of note in this architecture is that the web server is never directly connected to the microservice and everything is directed through and in the Redis database. Since Redis Enterprise is known for its ability to scale easily in order to accommodate gigantic amounts of traffic, this connection allows for near infinite growth of users, servers and microservice instances. Additionally, Redis Enterprise has extensive high availability features, which is essential for the critical components of any architecture.

Why a microservice?

Microservices have some very well known advantages over a monolithic approach to building a complex application. With microservices, you can more easily scale your application's capacity, development and reliability. In this case, "development" refers to the developers/teams needed to implement complex constellations of functions that your application will need to cover.

Developers can be a picky bunch. One team may want to use an entirely different set of tools, languages and methodologies than another team. Merging these teams together can be difficult (or impossible) to efficiently accomplish. In addition, the larger a codebase grows, the harder it is to:

- Understand
- Test
- Add features

By approaching a session store as a microservice, you can abstract the complexity of the store from the web serving layer and even use entirely different languages and tools. Indeed, you can precisely test a session store for appropriate behavior.

Finally, you can add features independently from any other parts of the serving layer.

For capacity, microservices allow you to assign more infrastructure as needed, without having to scale out other parts of the architecture. In contrast to a monolithic approach, microservices allow infrastructure to be sized appropriately for discrete services. With our session microservice example, we can add significant extra functionality without worrying about it becoming a chokepoint later as we can scale out. The opposite is also true: if other parts of the architecture are resource intensive, we can keep the session management layer more lean.

Finally, microservices can add reliability to our application. Building microservices intrinsically requires attention to failure situations. Indeed, you can even build in functionality to allow for hot restarts of microservices or even complete failures of a particular microservice, causing your application to degrade rather than hard fail.

Transport Mechanisms

For our session store, we'll be using Redis in two ways. Firstly, we'll be using Redis as you might expect: as a database. Secondly, we'll be using Redis as a transport. Redis has some very interesting and lightweight mechanisms to manage data flow.

The mostly commonly known Redis functionality of this type is pub/sub. The publish and subscribe pattern in Redis is classified as "fire and forget," which means that once you publish a message, it's gone—there is no automatic and durable record of that message being published. Much like the classic thought experiment:

> "If a tree falls in a forest and no one is around to hear it, does it make a sound?"

The Redis equivalent would be:

> "If a message is published to a channel and no one is subscribed, does it exist?"

Unlike the thought experiment, in Redis we have a straightforward answer: No.

You can subscribe to a single channel `(SUBSCRIBE)` or, more interestingly, you can subscribe to a pattern `(PSUBSCRIBE)` which monitors channels based on glob wildcards.

The other pattern we'll be using is blocking lists. Lists are ordered items that can be easily pushed to and pop'ed from. A blocking list is the same underlying data type but with a twist: in the case of an empty list the server blocks any other commands from being executed until either an item is pushed by another client into the list OR until a timeout clock has elapsed.

With these two patterns, we can create a flexible microservice transport that supports both critical and non-critical pathways. Let's examine the non-critical pathway first:

Step	Serving Layer	Session Microservices
0		Pattern subscribed to anything with "pageview:?*"
1	Gets HTTP request	
2	Publishes Redis message to a channel "pageview:[session id]:[request id]"	
3	Renders route	Gets message from serving layer
4	Closes connection	Starts processing
5		Ends processing
6		Adds message to a list "async:pageview:[session id]:[request id]"

Step 0 occurs at some point prior to the request being received.

In this sequence, the serving layer is not waiting for the session microservice to respond. Indeed, the serving layer only sends out the message but it doesn't have any knowledge of whether or not this message was received by the intended recipient. This type of sequence is great for situations where the data being stored in the session is completely trivial and not critical to the rest of the rendering. The advantage of using this pattern is that the only performance impact on the serving layer is a simple O(1) command: `PUBLISH.` The session microservice processing can occur during or even after the response has been sent.

Now, for the critical path, we'll see how the microservice layer plays a larger role.

Step	Serving Layer	Session Microservices
0		Pattern subscribed to anything with “pageview:?*”
1	Gets HTTP request	
2	Publishes Redis message to a channel “pageview:[session id]:[request id]”	
3	Blocks with BLPOP on key “async:pageview:[session id]:[request id]”	Gets message from serving layer
4		Starts processing
5		Ends processing
6		Adds message to a list "async:pageview:[session id]:[request id]"
7	Response from blocking command; gets data from list	
8	Renders route	
9	Closes connection	

Step 0 occurs at some point prior to the request being received.

The two sequences are the same up until step 3. Here, in the critical pathway, at step 3 we now block the serving layer's client (note: you either have to create a Redis client connection per request or, better, use a connection pool). This means that the client will sit idle until the microservice layer adds an item to the list or timeout is met (the timeout can be a very low value). A very important point is that both session and request have unique IDs. There should only ever be one subscriber to any given “async:*” list because of the unique IDs. On the microservice layer, the microservice does its processing, then posts a message to this unique list. This message can either be simply something that indicates to the serving layer that the microservice layer is complete or the message can, itself, contain information. This sequence is useful when you need to confirm that the microservice has completed or the rest of your steps depend on something the microservice has performed. As you might imagine, in a relative sense, this sequence has more overhead and potential latency, but keep in mind that these are very simple, lightweight operations that are most likely in the sub-millisecond timescale (obviously, minus whatever processing time the microservice requires).

From submitted to calculated data

Most session stores simply store data that is at some point submitted by the user: account details, preferences, or maybe even shopping carts. In this way, the data is rather simple—effectively, the database is merely holding a value and serving it back to the user. Session stores can, however, be used for more sophisticated data. Let's take a look at some data structures available in Redis and/or Redis modules (later, we'll illustrate how to use them in a session store situation). With all of the structures described below, we'll describe their properties as they relate to use and storage characteristics, without diving too deeply into the algorithm implementation or theory (which is beyond the scope of this document).

Bloom Filters

Bloom filters are likely the most well-known probabilistic data structure. Bloom filters test for presence and can tell if an item has possibly been added to the filter previously or definitely not been added to the filter previously. In other words, false positives are possible, but false negatives are not. In exchange for accepting false positives you gain immense savings in storage: you need a handful of bits per item, regardless of the sizes or number of items in the filter. So, you can do two operations with Bloom filters:

- Add an item
- Check if an item has previously been added to the filter

Bloom filters in Redis are available through the module ReBloom, which can be either used in the open source version of Redis by downloading, compiling and running the `MODULE LOAD` command or directly in Redis Enterprise by downloading the package and enabling it in the administrative interface.

The most simplistic implementations of Bloom filters can "fill up," i.e. as more items are added, the higher the error rate becomes. The ReBloom implementation of Bloom filters are scalable: you specify an error rate, and if you try to add items that would result in an error rate larger than the specified rate, ReBloom will seamlessly create an additional Bloom filter inside the same key. When you check for existence of an item, it will then check all the internal Bloom filters. This trade-off does result in a heavier existence check operation for Bloom filters that have been scaled multiple times, but it can yield a virtually consistent error rate. Luckily, you can specify an initial size to fit your item cardinality requirements, lessening unneeded scaling.

HyperLogLog

HyperLogLog is a structure that can determine the cardinality of unique items probabilistically. Like Bloom filters, you need mere bits per item and the size of the items are unrelated to the size of the structure. The cardinality of the unique items is an estimate. As an example of size required and error rate, you can hold 264 unique items (of any size) in ~12kb of storage with an error rate of 0.81%. HyperLogLog has three operations:

- Add an item
- Count the unique items
- Merge to HyperLogLogs together

The merge operation will yield a set count of unique items between the two HyperLogLogs (union). As an example, if you merge two HyperLogLogs, each holding the lone item "dog," and you then ran a count on the newly merged HyperLogLog, the count would still only be 1.

HyperLogLog is one of Redis' built-in data types.

Bit Counting

Bit counting is not a stand-alone structure, but rather an alternate method of using the Redis String data structure. Redis can manipulate a string value as an array of bits by index at an O(1) complexity on pre-allocated values. There are many uses for this technique, but we can use it directly to create a time series-like structure.

If you start with a known, fixed point in time and keep count of a specified period of time since this fixed point, you can use set individual bits at correlated indexes. Let's say your fixed point is Jan 1, 2018 at midnight (2018/1/1 00:00:00), you want to record that something happened at 1am (1/1/2018 01:00:00) and you're keeping track of minute periods. In this case you would flip the 60th bit from 0 to 1 to indicate some activity at 1am.

Redis has a number of commands that enable:

- Counting the first bit set to 1 in an array
- Querying individual bits
- Setting individual bits

- Querying multiple bits
- Accessing multiple bits as numbers of arbitrary bit-width

Bit counting and the bit counting commands are built-in capabilities of Redis.

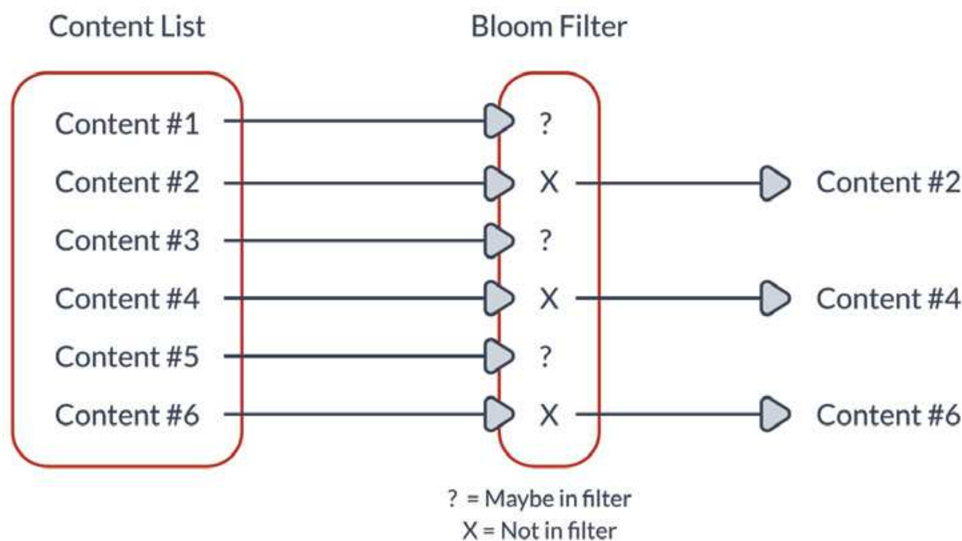
Session Store Patterns

Let's explore a few patterns for our session store microservice to implement.

Content Surfacing

Given a content-heavy platform such as a blog, e-commerce or news site, a common goal is to keep a user engaged by showing them new (and personalized) content. This presents challenges as it would involve storing every content piece that the user has interacted with, then finding the difference to a set list. If you insert the content read by the user in the primary database for the site, this would quickly turn into a scaling problem as, for most databases, writing is an expensive operation—with Redis being a notable exception. Then you would need to do another expensive operation to find the difference between the list of content and the list of viewed content. We can do better with a session store and a Bloom filter.

First, any time a user views a piece of content we can add it to a Bloom filter housed in that user's session store. Second, when we want to surface new content, we can take a list of content pieces and then check for the existence of these pieces in the filter. Checking for presence in a Bloom filter is the O(log n) operation, where n is the number of times the filter has had to be scaled (n will likely be very small, so we can check for presence of multiple items without worrying too much about performance). Since Bloom filters cannot return false negatives, we know with certainty that the content that is not in the Bloom filter is 'fresh' for this particular user.



The interesting thing about this pattern is that it can be scaled very easily. The list of content will change relatively infrequently and can be cached from the primary database easily. Each user has their own Bloom filter and the storage requirements are modest, which means that the needs are more related to the size of the user base rather than the amount of content or number of interactions. Finally, in a sharded environment, like a Redis Enterprise cluster, the data and load can be easily shared across multiple shards since it's made up of small, individual keys per user.

Activity Pattern Monitoring & Personalization

Today's most engaging and dynamic web experiences are powered by data that users provide to the site, both consciously and subconsciously. We can utilize our microservice to collect users' behavioral data and then process and utilize that data to create a customized experience for the user.

Firstly, this process is often seen as difficult to accomplish. Analytical data is often collected in another service (Google Analytics, for example) and this data is anonymized. Doing self-collection of this type of data is often infeasible because of the write and storage requirements that are difficult to meet when recording all the data directly in disk-based databases.

Secondly, making decisions on this type of data is deferred to complicated and expensive graph or ML-based solutions. While these solutions often yield interesting results, the data is inferential and locked into black boxes that are difficult to combine with other insights. For example, you might have market research that demonstrates some concrete correlations, but if your ML-based model doesn't provide output with the specific data that lines up with the market research, you can't fully utilize your data. Take this example:

	ML Solution	Direct
Input(s)	User behavior	User behavior (Added television and wall mount to cart)
Output	User would like item #123, ...	User would like an HDMI cable
How do we know this?	? (black box)	Market research tells us that 9 of 10 people who purchase a TV and wall mount also purchase a new cable

To accomplish this type of monitoring, we need to utilize many of our structures. With a microservice we have limited amounts of information, but it will be sufficient to gain significant insights.

First, let's record site activity. We can accomplish this with bit counting, which will tell us not only when activity occurred, but also any discernible patterns in that activity. Initially, we'll pick our fixed point in time and a time period. Let's say the starting point is January 1st and our time period will be minutes. Any time a user loads a page during any particular minute, we'll flip the bit corresponding to the number of minutes since our fixed point; 1500 minutes after our fixed point means we'll flip the 1500th bit in our activity monitoring key.

Next, we'll record the unique pages consumed by the visitor in a HyperLogLog, simply adding the URI to the HyperLogLog. Likewise, for every page consumed by the visitor, the URI is added to a page visit Bloom filter. We'll also increment a simple Redis counter on every page visit.

All of these operations are done on each page visit—this might sound like quite a bit, however all of these operations are lightweight.

With this information, we can start combining them into specific insights. For example:

- Is this the first time a user has visited a specific page? Check the URI in the page visit Bloom filter.
- New to the site? The HyperLogLog count is 1 and the page is not in the Bloom filter.
- Inactive for the last five minutes? Sum the last five bits of the activity bit counter. If 0, the user hasn't been on the site in the last five minutes.
- Visited a cluster of pages? Check the Page Visit bloom filter for this cluster.

Given our previous example, let's say the Cart Add confirmation page has a URI that looks like this:

```
/added/[primary-item-category]
```

So, we have two URIs that could look like this:

```
/added/tv

/added/tv-mount
```

When the user visits the Purchase Confirmation page, we can check the page visit bloom filter for various combinations, including our TV & TV Mount combination. If these match up, then we can suggest an HDMI cable for the user to complete the purchase. This can be accomplished in real time and without having to record each interaction verbatim. It should be noted that we're relying on a positive presence test in a Bloom filter; consequently, it's possible that false positives will occur. Recall that the ReBloom implementation can grow while keeping the error rate at an acceptable level, preventing an ever increasing false-positive rate as the filter fills. Additionally, since we are checking multiple items and the error probability is independent on each check, the error rate on the combination multiplies for each check. Assuming that we have set up a Bloom filter in ReBloom with the default error rate (0.01 or 1/100), the probability of both items being false positives is 1/10,000 for a given Bloom filter state. Finally, in this use case, it would be acceptable to suggest an item that isn't perfect: who hasn't had that happen while doing online shopping? It is a tradeoff between accuracy and speed /storage efficiency that must be weighed for each individual use case.

Privacy is obviously a concern when it comes to monitoring behavior. Users often feel a sense of unease when they realize that a site may be tracking the pages they visit. The thought of a nefarious 3rd party gaining access to this information is disconcerting. An interesting aspect of probabilistic data structures such as Bloom filters or HyperLogLogs is that they are intrinsically one way—it is impossible to extract the information put into them directly. If, through some means, a malicious 3rd party gained access to the Bloom filter or HyperLogLog data in the session store we're describing, they would only be able to query items (URIs, in this case), not get a list of items. While this shouldn't be considered a form of encryption, this configuration makes you a much less appealing target when the malicious actor has to approach it as a needle-in-a-haystack exercise.

Group Notifications

Let's say your website has groups of users and a notification system. Sending notifications to a single user is simple enough: you have a list of notifications and a status (read/unread). Each users has his/her own list. Group notifications, however, present some challenges. When sending a notification to a group, a new notification is inserted to the personal notification list of each user. This is problematic. Pushing notifications to a large group of users requires a write operation for each user which might be fine for small groups but imagine groups with millions of users. When a user has seen the notification, you need to set the status from unread to read, which would involve touching the primary database.

To make this more efficient, let's rework this pattern. First, group notifications are stored in a simple List data structure for the entire group. When you want to add a notification, simply add one item to this structure. This becomes an easily cacheable piece of data since it will change infrequently and is the same for all users. Next, we have a Bloom filter for each user stored in their Redis-backed session. When the user checks the group notifications, you take the first n items from the list and check for their presence in the Bloom filter—seen items are present in the filter while unseen items are not. In this case, we assume everything has been seen unless the notification is not in the filter.

This process is very lightweight. Adding notifications to a group is merely a single update that affects every user in the group, no matter the size of the group. Marking "read" occurs fully in the session store and is on a data structure that is only linked to a single user, which means it can be scaled effortlessly. The storage requirements are minimal and checking for the read/unread status consists of some simple hashing functions followed by querying only a very few bits.

While some of these patterns may fall into the ephemeral category, notifications fall into a class of data that should be session based (unique to a single user) and always persisted. Redis Enterprise is uniquely positioned to fill this role as it allows for persistence through in-memory replication. Pairing this with one-minute snapshotting persistence can create a layered durability where no single event can take out your in-memory database and, in the event that all nodes fail, you've got a copy that is no more than one minute old.

Putting it all together

Let's start by building our serving layer. It will be a simple web server based in Node.js with Express. You can certainly use any language you'd like to develop this server; there is nothing special about the abilities of Javascript. The only real requirements are (1) the ability to know the route that is being served, (2) the ability to generate relatively unique numbers and (3) the ability to make Redis calls.

The server itself has a few key routes:

/ The homepage will direct you to some of the examples

/added/[anything] This route is a wildcard—it will show a simple page for any value after *"/page/"*. We have a couple of primary categories "tv" and "tv-mount" to illustrate the personalization use case

/notifications This displays the notifications as read or unread

On each page you'll see a footer with the stats (that won't be currently populated—more on that later).

To run the server, first install it with NPM:

```
$ npm install
```

Then run the server:

```
$ node index.js --connection ./path-to-your-redis/connection-info-as.json
```

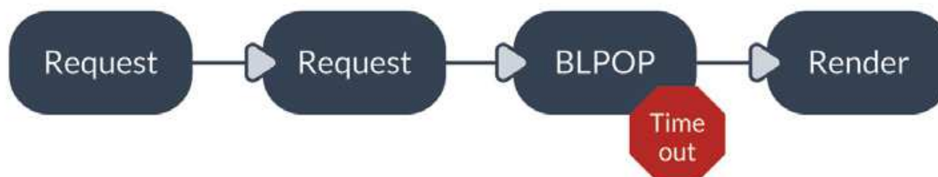
Then you can connect at *http://localhost:3379/*

At this point the server will work, but you may notice that it's semi-functional and some pages are slow. That's because we're trying to interact with a microservice that isn't running: we're in degraded mode.

To run the microservice, open a new terminal window and run:

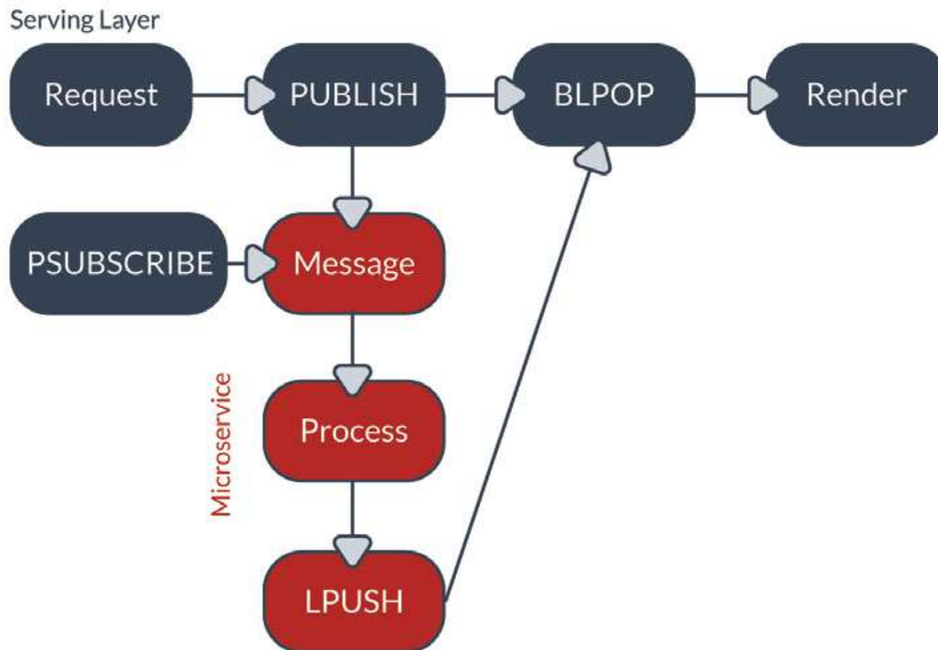
```
$ node microservice.js --connection ./path-to-your-redis/connection-info-as.json
```

Once this is running, you'll notice that the footer will start to be populated and the critical path pages are running much more quickly.



Let's examine how this works visually, first in degraded mode for a critical path:

Now on a critical path route the `PUBLISH` on the serving layer is triggering the `PSUBSCRIBE` on the microservice. The serving layer is setting idle because of the blocking command. When the microservice is done processing, it then pushes an item into the list, unblocking the serving layer.



The interesting thing about this architecture is that the serving layer and the microservice never touch each other, they are both connected independently through Redis. The serving layer only uses two commands:

```
PUBLISH

BLPOP
```

`PUBLISH` acts as a requestor to the microservice. In the case of the critical path, the `BLPOP` awaits any responses at the "async:…" key as the published channel. We place a one second timeout on the `BLPOP`, which is too long for even degraded performance of a live site. On the application layer, we place a timeout at 100ms, at which point the service stops waiting and ignores any subsequent response from the `BLPOP`. For clarity's sake, the demo code doesn't illustrate this, but in production it would be wise to also issue an `EXPIRE` on this key in order to cover situations where the serving layer dies after issuing a `PUBLISH` and before the `BLPOP`. Finally, when an item is pushed from the microservice, the value is encoded in JSON as the list item. The serving layer parses the JSON and uses it to render the pages.

On the microservice side, `PSUBSCRIBE` is utilized to subscribe to multiple channels at one time. These channels, along with both the unique session ID and request ID, act as the request topic. The session store parses the channel name and starts performing the actions for that particular request topic. Once the session store completes the actions, it encodes the values in JSON and pushes them, as an item, to the complete original message topic.

Finally, it's important to note that this architecture allows for multiple instances of the microservice to be running. It can be split by changing the glob pattern matches in the session IDs. As an example, if your session IDs are in base 36 (a-z0-9), you could assign one microservice instance to handle session IDs that start with "a" to "r" and another instance to handle "s" through "9." This can be simulated by running multiple versions of the microservice. Additionally, any number of serving layers can be assigned to your microservice(s).

In the demo code, we're doing a number of different subscriptions:

`ss:mark-notification-seen:[session ID]:[request ID]` marks a notification as "seen." The microservice uses `BF.ADD` (add item to Bloom filter).

`ss:pageview:[session ID]:[request ID]` does the analytics for each page. The microservice uses a `MULTI/EXEC` block with `PFADD, INCR, BITFIELD, PFCOUNT` and `BF.ADD` to record unique page views, total page views and activity; get activity; and get the current unique page count.

`ss:combo:[session ID]:[request ID]` checks the status of the pre-defined combo (e.g. added a "tv-mount" and "tv") using `BF.MEXISTS.`

`ss:notifications:[session ID]:[request ID]:[JSON Array of notifications]` checks to see if notifications have been seen or not. The microservice takes the JSON encoded array of notifications and checks them against the bloom filter with `BF.MEXISTS.`

`ss:featuredpages:[session ID]:[request ID]:[JSON Array of notifications]` checks to see if any of the featured pages have been viewed or not. The microservice takes the JSON encoded array of notifications and checks them against the bloom filter with `BF.MEXISTS.` The demo uses the same data for featured pages as the combo pages.

Each subscription is abstracted as a `command` in the code. This command is noted by the second colon-delimited item in each subscription (mark-notification-seen, pageview, etc.). We iterate through the commands and issue a `PSUBSCRIBE` to match anything at the end of these patterns (`?*`).

Conclusion

Creating a session store microservice backed by the unique features of Redis Enterprise allows for scalable and data-rich patterns that would be otherwise impractical to implement in monolithic architectures. The data can be stored and retrieved from the session store and include probabilistic and analytical data alongside more traditional session data.

This architecture allows for multiple teams to collaborate in a polyglot environment. While the examples in this document are in Javascript, the ubiquity of Redis clients across programming languages allow for teams to create services that use the same pattern in practically any language or runtime.

The session store microservice makes possible more advanced data storage than what would be available in other databases, thanks to the rich data types available in Redis and their extensibility through the Redis module system. By using the pub/sub and block list patterns together, computational infrastructure can be optimized to fit the particular needs and patterns of your particular use case.

The demo code for the techniques outlined in this white paper can be found at: https://github.com/stockholmux/intelligent-session-store-microservice

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com