Remote Settings Documentation

Mozilla

Feb 04, 2021

Contents

Remote Settings is a Mozilla service that makes it easy to manage evergreen settings data in Firefox. A simple API is available in Firefox for accessing the synchronized data.

CHAPTER 1

What is Remote Settings?

Basically, Remote Settings consists of two components: a remote server (REST API) powered by Kinto and a client (Gecko API).

Everything is done via a collection of records that is kept in sync between the client local database and the remote data.

1.1 Why is it better than building my own?

Out of the box you get for free:

- Multi-signoff
- Syncing of data - real time push based updates
- Content signing - your data is signed server side and verified on the client side transparently
- File attachment support
- Target filtering (JEXL a-la Normandy)
- Telemetry

1.2 What does the workflow look like?

Once your collection is setup, a typical workflow would be:

1. Connect to the UI on the VPN
2. Make some changes
3. Request a review (with an optional comment)

The people designated as reviewers will receive a notification email.

4. As a reviewer, you can preview the changes in a real browser

5. Once you verified that the changes have the expected effects, you can approve (or reject) the changes from the Admin UI
6. Changes will then be pulled by every Firefox clients on their next synchronization

1.3 What does the client API look like?

On the client side, listen for changes via an event listener:

```
const { RemoteSettings } = ChromeUtils.import("resource://services-settings/remote-
↪settings.js", {});

RemoteSettings("my-collection")
  .on("sync", (e) => {
    const { created, updated, deleted } = e.data;
    /*
      updated == [

        {
          old: {label: "Yahoo",  enabled: true,  weight: 10, id: "d0782d8d", last_
↪modified: 1522764475905},
          new: {label: "Google", enabled: true,  weight: 20, id: "8883955f", last_
↪modified: 1521539068414},
        },
      ]
     */
  });
```

Or get the current list of local records:

```
const records = await RemoteSettings("my-collection").get();
/*
  records == [
    {label: "Yahoo",  enabled: true,  weight: 10, id: "d0782d8d", last_modified:␣
↪1522764475905},
    {label: "Google", enabled: true,  weight: 20, id: "8883955f", last_modified:␣
↪1521539068414},
    {label: "Ecosia", enabled: false, weight: 5,  id: "337c865d", last_modified:␣
↪1520527480321},
  ]
*/
```

Note:

- Client API full reference

1.4 What does the server side API look like?

If you want, like our Web UI, to rely on the REST API for your integration, the *multi-signoff tutorial* gives a good overview.

Basically, creating a record would look like this:

```
curl -X POST ${SERVER}/buckets/main-workspace/collections/${COLLECTION}/records \
     -H 'Content-Type:application/json' \
     -d "{\"data\": {\"property\": $i}}" \
     -u us3r:p455w0rd
```

Requesting review:

```
curl -X PATCH ${SERVER}/buckets/main-workspace/collections/${COLLECTION} \
     -H 'Content-Type:application/json' \
     -d '{"data": {"status": "to-review"}}' \
     -u us3r:p455w0rd
```

Approving changes:

```
curl -X PATCH ${SERVER}/buckets/main-workspace/collections/${COLLECTION} \
     -H 'Content-Type:application/json' \
     -d '{"data": {"status": "to-sign"}}' \
     -u us3r:p455w0rd
```

And the record is now published:

```
curl ${SERVER}/buckets/main-workspace/collections/${COLLECTION}/records
```

Note:

- Kinto REST API reference
- Python client
- JavaScript client

1.5 Awesome! How do I get started?

You'll find out *in the next chapter*!

CHAPTER 2

Getting Started

We will help you to use Remote Settings in your application!

2.1 Create a new official type of Remote Settings

Basically, you will have to go through these 3 steps:

1. Setup the Mozilla VPN and request access to the server using this Bugzilla template
2. Design your data model (see below) and prepare the list of colleagues that will be allowed to review your data
3. Request the creation of your collection using this Bugzilla ticket template

Once done, you will be able to login and edit your records on the Admin UIs:

- https://settings-writer.prod.mozaws.net/v1/admin/

The records will then be publicly visible at https://firefox.settings.services.mozilla.com/v1/bucket/main/collections/\protect\T1\textbraceleftcollection-id\protect\T1\textbraceright/records

Don't hesitate to contact us (`@delivery` on Slack) if you're stuck or have questions about the process!

Check out the *screencast to create, request review and approve changes*, or *our FAQ*!

Note: In order to **try out changes in a real environment**, you can use the **STAGE** instance:

- https://settings-writer.stage.mozaws.net/v1/admin/ (*Admin UI*)

In order to switch Firefox from PROD to STAGE, use the Remote Settings DevTools!

The records will be publicly visible at https://settings.stage.mozaws.net/v1/bucket/main/collections/\protect\T1\textbraceleftcollection-id\protect\T1\textbraceright/records

---

Note: If you simply **want to play** with the stack or the API, the best way to get started is probably to use our *DEV server*, since everyone is allowed to manipulate data on the server and the multi-signoff workflow is not enabled. Check out the *dedicated tutorial*!

2.2 About your data

Name your collection in lowercase with dashes (eg. `public-list-suffix`, examples).

The Admin UI automatically builds forms based on some metadata for your collection, namely:

- the list of fields to be displayed as the list columns (eg. `title`, `comment.author`)
- a JSON schema that will be render as a form to create and edit your records (see example)
- whether you want to control the ID field or let the server assign it automatically
- whether you want to be able to attach files on records

Note: If your client code expects to find 0 or 1 record by looking up on a specific field, you should probably use that field as the record ID. `RemoteSettings("cid").get({filters: {id: "a-value"}})` will be instantaneous.

It is recommended to keep your Remote Settings records small, especially if you update them often. If you have big amounts of data to publish via Remote Settings, use our *file attachments feature* instead.

By default, all records are made available to all users. If you want to control which users should have a particular entry, you can add a `filter_expression` field (see *target filters*).

2.3 Collection manifests

Both STAGE and PROD collections attributes and permissions are managed via YAML files in the remote-settings-permissions Github repository.

If you want to accelerate the process of getting your collection deployed or adjust its schema, in STAGE or PROD, you can open a pull-request with the collection, and the definition of `{collection}-editors` and `{collection}-reviewers` groups. Check out the existing ones that were merged.

CHAPTER 3

Case Studies

This page (*under construction*) contains some pointers to existing use cases and implementations.

3.1 Search configuration

The list of search engines is managed via the `search-config` collection.

3.1.1 Configuration

- A complex JSON schema validates entries (source)
- On the server, the group of editors is different from the group of users allowed to approve changes (permissions)

3.1.2 Implementation

- Client is initialized from the Search service (source)
- Initial data is packaged in release, and is loaded on first startup from a call to `.get()` (source)
- A hijack blocklist is managed separately, and the integrity/signature of the local records is verified on each read (source). A certificate might have to be downloaded if missing/outdated.

3.1.3 Misc

- searchengine devtools

3.2 Normandy Recipes

The list of ongoing experimentations is managed via the `normandy-recipes-capabilities` collection.

3.2.1 Implementation

- Synchronization happens on first startup because no initial data is packaged:
  - `Normandy.init()` in BrowserGlue (source)
  - Calling `.get()` with implicit `syncIfEmpty: true` option will initialize the local DB by synchronizing the collection for Normandy (source)
- In order to guarantee that the records are published from Normandy, each record is signed individually on the server side (source). Records are published from Django using `kinto-http.py`.
- Signature verification for the whole collection is done as usual, and the per-record one is verified on read when recipe eligibility is checked (source)

3.2.2 Configuration

- Multi signoff is disabled (config)
- A scheduled task backports certain recipes into a legacy collection for old clients (source, config)

3.2.3 Misc

- Poucave checks for Normandy

3.3 HIBP Monitor Breaches

The list of websites whose credentials database was leaked is managed via the `fxmonitor-breaches` collection.

3.3.1 Automation

- A script pulls from Have I Been Powned API, and creates the missing records using a Kinto Account, and then requests review (source)
- This script is ran by OPs as a cron job (source, request ticket)
- A human approves the changes manually

3.4 Blocklist

The list of blocked addons is managed via the `blocklists/addons-bloomfilters`.

3.4.1 Implementation

Addons blocklist implemented using a bloomfilter (docs)

- Bloomfilters are published from a Cron job on the addons-server, implementated using raw Python requests (source)
- Incremental updates of bloomfilters are downloaded as binary attachments, full or base + stashes (source)
- Attachments are stored in IndexedDB thanks to the `useCache: true` option (source)

- When using the attachment IndexedDB cache, attachments can be packaged in release in order to avoid downloading on new profiles initialization. The bloomfilter base attachment is shipped in release along with its record metadata (source)
- Attachments are updated in tree regularly using custom code in `periodic_file_updates.sh` (source)

3.4.2 Misc

- Remote Settings authentication from CLI in Javascript (See Bug 1630651)

3.5 User Journey

Contextual features recommandations is managed via the `cfr` collection.

3.5.1 Localization

- Contextual recommandations are published using translatable placeholders or string IDs

```
"content": {
  "icon": "chrome://browser/skin/notification-icons/block-fingerprinter.svg",
  "text": {
    "string_id": "cfr-doorhanger-fingerprinters-description"
  },
  "layout": "icon_and_message",
  "buttons": {
    "primary": {
      "event": "PROTECTION",
      "label": {
        "string_id": "cfr-doorhanger-socialtracking-ok-button"
      },
      "action": {
        "type": "OPEN_PROTECTION_PANEL"
      }
    },
    ...
```

- In parallel, localizations are published in a separate collection
- Each locale has its own record, with its ID in the following format ‘‘ *cfr-v1-${locale}* ‘‘ and a Fluent file attached.
- A specificly instantiated downloader fetches the relevant one and reloads l10n (source)
- This specific record is checked on each load, attachment is downloaded only if updated/missing/corrupted (built-in feature of attachment downloader)

3.6 Security State

Several security related collections are managed in the dedicated `security-state` bucket.

3.6.1 Configuration

- Dedicated bucket in order to have specific content signature certificates

```
const OneCRLBlocklistClient = RemoteSettings(
  Services.prefs.getCharPref(ONECRL_COLLECTION_PREF),
  {
    bucketNamePref: ONECRL_BUCKET_PREF,
    lastCheckTimePref: ONECRL_CHECKED_PREF,
    signerName: Services.prefs.getCharPref(ONECRL_SIGNER_PREF),
  }
);
```

source

3.6.2 Cert Revocations (CRLite)

Certificates revocation list using a bloomfilter.

- Sysops run a scheduled job that pulls data from a Git repo, authenticates using a Kinto account to publish (`account:crlite_publisher`), and approves changes with another one (`account:crlite_reviewer`) (source)
- Download of attachments happens sequentially at the end of first sync (*caution*)
- Incremental updates of bloomfilters are downloaded as binary attachments in profile folder (source)
- Poucave check for age of revocations (source).

3.6.3 Intermediates

- Download of attachments sequentially at the end of first sync (*caution*)

CHAPTER 4

Support

4.1 Troubleshooting

- Open a Server Side ticket (Admin, permissions etc.)
- Open a Client Side ticket (Gecko API related)

4.1.1 I cannot access my collection

- Check that you can ping the server on the VPN - Make sure you were added in the appropriate VPN group (see *Getting Started*) - Join `#engops` on Slack to troubleshoot.
- Check that you can login on the Admin UI
- In the `main-workspace` bucket, check that you can create records in your collection (eg. main-workspace/tippytop)

4.1.2 I approved the changes, but still don't see them

- A CDN serves as a cache, only push notifications bust the cache efficiently
- Check that your data is visible on the source server: eg. https://settings.prod.mozaws.net/v1/buckets/main/collections/cfr/changeset?_expected=something-random-42

4.2 Frequently Asked Questions

4.2.1 How do I setup Firefox to pull data from STAGE?

The recommended way to setup Firefox to pull data from STAGE is to use the Remote Settings DevTools extension: switch the environment in the configuration section and click the *Sync* button.

Alternatively, you can change the appropriate preferences, restart and trigger a synchronization manually.

4.2.2 How do I preview the changes before approving?

The recommended way to setup Firefox to pull data from the preview collection is to use the Remote Settings DevTools extension: switch the environment to *Preview* and click the *Sync* button.

Alternatively, you can change the `services.settings.default_bucket` preference to `main-preview`, and trigger a synchronization manually.

4.2.3 How do I preview the changes before requesting review?

Currently, this is not possible.

Possible workarounds:

- use a *local server* or the *DEV server*
- request review, preview changes, fix up, request review again

4.2.4 How do I trigger a synchronization manually?

See developer docs.

4.2.5 How do I define default data for new profiles?

See developer docs about initial data.

4.2.6 How do I automate the publication of records? (one shot)

The Remote Settings server is a REST API (namely a Kinto instance). Records can be created in batches, and as seen in the *multi signoff tutorial* reviews can be requested and approved using `PATCH` requests.

If it is a one time run, then you can run the script as if it was you:

1. Authenticate on the Admin UI
2. On the top right corner, use the icon to copy the authentication string (eg. `Bearer r43yt0956u0yj1`)
3. Use this header in your `cURL` commands (or Python/JS/Rust clients etc.)

```
curl 'https://settings-writer.stage.mozaws.net/v1/' \
  -H 'Authorization: Bearer r43yt0956u0yj1'
```

4.2.7 How do I automate the publication of records? (forever)

If the automation is meant to last (eg. cronjob, lambda, server to server) then the procedure would look like this:

1. Request a dedicated Kinto internal account to be created for you (eg. `password-rules-publisher`). Secret password should remain in a vault and managed by OPs.
2. Write a script that:

1. takes the server and credentials as ENV variables (eg. `SERVER=prod AUTH=password-rules-publisher:s3cr3t`);
2. compares your source of truth with the collection records. Exit early if no change;
3. performs all deletions/updates/creations;
4. patches the collection metadata in order to request review (see *multi-signoff tutorial*);
3. Request the OPs team to setup a cronjob in order to run your script (request example)

We recommend the use of kinto-http.py (script exanple), but Node JS is also possible (HIBP example).

Note: Even if publication of records is done by a script, a human will have to approve the changes manually. Generally speaking, disabling dual sign-off is possible, but only in **very** specific cases.

If you want to skip manual approval, request a review of your design by the cloud operations security team.

4.2.8 How often the synchronization happens?

Synchronizations can be within 10 minutes of the change or in 24 hours.

There are two triggers for synchronization: a push notification and a polling check. Every five minutes a server side process checks for changes. If any changes are found a push notification will be sent and online clients will check in for updates. Clients that are offline or did not receive the push notification will either catch-up on next startup or automatically poll for changes every 24 hours.

4.2.9 Once data is ready in STAGE, how do we go live in PROD?

Stage and prod are aligned in terms of setup, features and versions.

Hence, once done in STAGE there is nothing specific / additional to do: you should be able to redo the same in PROD!

If you have a lot of data that you want to duplicate from one instance to another, you can use kinto-wizard to dump and load records!

```
pip install --user kinto-wizard
```

Dump the main records:

```
kinto-wizard dump --records --server https://settings.stage.mozaws.net/v1 --
↪bucket=main --collection=top-sites > top-sites.yaml
```

Open the `.yaml` file and rename the bucket name on top to `main-workspace`.

Login in the Remote Settings Admin and copy the authentication header (icon in the top bar), in order to use it in the `--auth` parameter of the `kinto-wizard load` command.

```
kinto-wizard load --server https://settings.prod.mozaws.net/v1 --auth="Bearer uLdb-
↪Yafefe....2Hyl5_w" top-sites.yaml
```

Requesting review can be done via the UI, *or the command-line*.

4.2.10 How many records does it support?

We already have use-cases that contain several hundreds of records, and it's totally fine.

Nevertheless, if you have thousands of records that change very often, we should talk! Mostly in order to investigate the impact in terms of payload, bandwidth, signature verification etc.

4.2.11 Are there any size restrictions for a single record, or all records in a collection?

Quotas were not enabled on the server. Therefore, technically you can create records with any size, and have as many as you want in the collection.

However, beyond some reasonable size for the whole collection serialized as JSON, it is recommended using our *attachments feature*.

Using attachments on records, you can publish data of any size (as JSON, gzipped, etc.). It gets published on S3 and the records only contain metadata about the remote file (including hash, useful for signature verification).

4.2.12 Also does remote settings do any sort of compression for the records?

We are working on improving the handling of Gzip encoding for the attachments files (see Bug 1339114).

But by default, Remote Settings does not try to be smart regarding compression.

4.2.13 Is it possible to deliver remote settings to some users only?

By default, settings are delivered to every user.

You can add *JEXL filters on records* to define targets. Every record will be downloaded but the list obtained with `.get()` will only contain entries that match.

In order to limit the users that will download the records, you can check out our *dedicated tutorial*.

4.2.14 How does the client choose the collections to synchronize?

First, the client fetches the list of published collections.

Then, it synchronizes the collections that match one of the following:

- it has an instantiated client — ie. a call to `RemoteSettings("cid")` was done earlier
- some local data exists in the internal IndexedDB
- a JSON dump was shipped in mozilla-central for this collection in `services/settings/dumps/`

CHAPTER 5

Target filters

By default all records in a collection are made available. However, there are some use cases where it is more practical to use a single server side collection and filter record visibility in the browser. Target filters allows for these use cases.

Filters are conditional expressions evaluated in the client's browser and, if they pass, the corresponding record is available. Filters have access to information about the user, such as their locale, addons, and Firefox version.

Important: All records are downloaded to Firefox. Filters only control what is returned by `.get()` or the `sync` event.

Important: The ideal use case for target filters is to use a single collection to service a small number (dozens) of different groups of users. Filtering by channel, browser version or locale are good use cases.

5.1 How?

When a record has a string in the `filter_expression` field, it is evaluated and the record is potentially filtered for some users.

In order to restrict a setting to a particular audience, just write the proper filter string in the Admin UI and save the record.

```
{
    id: "68b19efa-1067-401b-b1c1-8d7b4263bb86",
    last_modified: 1531762863373,
    title: "Only for users of 57",
    filter_expression: "env.version == 57"
}
```

When calling `RemoteSettings("key").get()` or listening to `sync` events, you will only see the settings entries whose `filter_expression` resolved to a truthy value (and those who don't have any as by default).

See *the dedicated section about testing and debugging*.

5.2 Filter Expressions

Filter expressions are written using a language called JEXL. JEXL is an open-source expression language that is given a context (in this case, information about the user's browser) and evaluates a statement using that context. JEXL stands for "JavaScript Expression Language" and uses JavaScript syntax for several (but not all) of its features.

Note: The rest of this document includes examples of JEXL syntax that has comments inline with the expressions. JEXL does **not** have any support for comments in statements, but we're using them to make understanding our examples easier.

5.2.1 JEXL Basics

The JEXL Readme describes the syntax of the language in detail; the following section covers the basics of writing valid JEXL expressions.

Note: Normally, JEXL doesn't allow newlines or other whitespace besides spaces in expressions, but filter expressions in Remote Settings allow arbitrary whitespace.

A JEXL expression evaluates down to a single value. JEXL supports several basic types, such as numbers, strings (single or double quoted), and booleans. JEXL also supports several operators for combining values, such as arithmetic, boolean operators, comparisons, and string concatenation.

```
// Arithmetic
2 + 2 - 3 // == 1

// Numerical comparisons
5 > 7 // == false

// Boolean operators
false || 5 > 4 // == true

// String concatenation
"Mozilla" + " " + "Firefox" // == "Mozilla Firefox"
```

Expressions can be grouped using parenthesis:

```
((2 + 3) * 3) - 3 // == 7
```

JEXL also supports lists and objects (known as dictionaries in other languages) as well as attribute access:

```
[1, 2, 1].length // == 3
{foo: 1, bar: 2}.foo // == 1
```

Unlike JavaScript, JEXL supports an `in` operator for checking if a substring is in a string or if an element is in an array:

```
"bar" in "foobarbaz" // == true
3 in [1, 2, 3, 4] // == true
```

The context passed to JEXL can be expressed using identifiers, which also support attribute access:

```
env.locale == 'en-US' // == true if the client's locale is en-US
```

Another unique feature of JEXL is transforms, which modify the value given to them. Transforms are applied to a value using the | operator, and may take additional arguments passed in the expression:

```
'1980-01-07'|date // == a date object
```

5.2.2 Context

This section defines the context passed to filter expressions when they are evaluated. In other words, this is the client information available within filter expressions.

env
The `env` object contains general information about the client.

`env.`**version**
Example: `'47.0.1'`

String containing the user's Firefox version.

`env.`**channel**
String containing the update channel. Valid values include, but are not limited to:

- `'release'`
- `'aurora'`
- `'beta'`
- `'nightly'`
- `'default'` (self-built or automated testing builds)

`env.`**isDefaultBrowser**
Boolean specifying whether Firefox is set as the user's default browser.

`env.`**appinfo**
Object containing application details:

- `ID`: String containing the XUL application ID, eg. Firefox is `"{ec8030f7-c20a-464f-9b0e-13a3a9e97384}"`.
- `platformVersion`: The version of the XULRunner platform
- `platformBuildID`: The build ID/date of gecko and the XULRunner platform
- `version`: The version of the XUL application. It is different than the version of the XULRunner platform. Be careful about which one you want.
- more...

`env.`**searchEngine**
Example: `'google'`

String containing the user's default search engine identifier. Identifiers are lowercase, and may by locale-specific (Wikipedia, for examnple, often has locale-specific codes like `'wikipedia-es'`).

The default identifiers included in Firefox are:

- `'google'`
- `'yahoo'`

- 'amazondotcom'
- 'bing'
- 'ddg'
- 'twitter'
- 'wikipedia'

env.**syncSetup**
Boolean containing whether the user has set up Firefox Sync.

env.**syncDesktopDevices**
Integer specifying the number of desktop clients the user has added to their Firefox Sync account.

env.**syncMobileDevices**
Integer specifying the number of mobile clients the user has added to their Firefox Sync account.

env.**syncTotalDevices**
Integer specifying the total number of clients the user has added to their Firefox Sync account.

env.**plugins**
An object mapping of plugin names to plugin objects describing the plugins installed on the client.

env.**locale**
Example: 'en-US'

String containing the user's locale.

env.**distribution**
String set to the user's distribution ID. This is commonly used to target funnelcake builds of Firefox.

On Firefox versions prior to 48.0, this value is set to undefined.

env.**telemetry**
Object containing data for the most recent Telemetry packet of each type. This allows you to target recipes at users based on their Telemetry data.

The object is keyed off the ping type, as documented in the Telemetry data documentation (see the type field in the packet example). The value is the contents of the ping.

```
// Target clients that are running Firefox on a tablet
env.telemetry.main.env.system.device.isTablet

// Target clients whose last crash had a BuildID of "201403021422"
env.telemetry.crash.payload.metadata.BuildID == '201403021422'
```

env.**doNotTrack**
Boolean specifying whether the user has enabled Do Not Track.

env.**addons**
Object containing information about installed add-ons. The keys on this object are add-on IDs. The values contain the following attributes:

env.addon.**id**
String ID of the add-on.

addon.**installDate**
Date object indicating when the add-on was installed.

addon.**isActive**
Boolean indicating whether the add-on is active (disabling an add-on but not uninstalling it will set this to false).

`addon.`**`name`**
String containing the user-visible name of the add-on.

`addon.`**`type`**
String indicating the add-on type. Common values are `extension`, `theme`, and `plugin`.

`addon.`**`version`**
String containing the add-on's version number.

```
// Target users with a specific add-on installed
env.addons["shield-recipe-client@mozilla.org"]

// Target users who have at least one of a group of add-ons installed
env.addons|keys intersect [
   "shield-recipe-client@mozilla.org",
   "some-other-addon@example.com"
]
```

5.2.3 Operators

This section describes the special operators available to filter expressions on top of the standard operators in JEXL. They're documented as functions, and the parameters correspond to the operands.

`intersect`(*list1*, *list2*)
Returns an array of all values in `list1` that are also present in `list2`. Values are compared using strict equality. If `list1` or `list2` are not arrays, the returned value is `undefined`.

Arguments

- **`list1`** – The array to the left of the operator.
- **`list2`** – The array to the right of the operator

```
// Evaluates to [2, 3]
[1, 2, 3, 4] intersect [5, 6, 2, 7, 3]
```

5.2.4 Transforms

This section describes the transforms available to filter expressions, and what they do. They're documented as functions, and the first parameter to each function is the value being transformed.

`stableSample`(*input*, *rate*)
Randomly returns `true` or `false` based on the given sample rate. Used to sample over the set of matched users.

Sampling with this transform is stable over the input, meaning that the same input and sample rate will always result in the same return value.

Arguments

- **`input`** – A value for the sample to be stable over.
- **`rate`** (*number*) – A number between `0` and `1` with the sample rate. For example, `0.5` would be a 50% sample rate.

```
// True 50% of the time, stable per-version per-locale.
[env.locale, env.version]|stableSample(0.5)
```

bucketSample (*input*, *start*, *count*, *total*)

Returns `true` or `false` if the current user falls within a "bucket" in the given range.

Bucket sampling randomly groups users into a list of "buckets", in this case based on the input parameter. Then, you specify which range of available buckets you want your sampling to match, and users who fall into a bucket in that range will be matched by this transform. Buckets are stable over the input, meaning that the same input will always result in the same bucket assignment.

Importantly, this means that you can use an independent input across several settings to ensure they do not get delivered to the same users. For example, if you have two settings that are variants of each other, you can ensure they are not shown to the same cohort:

```
// Half of users will match the first filter and not the
// second one, while the other half will match the second and not
// the first, even across multiple settings.
[env.locale]|bucketSample(0, 5000, 10000)
[env.locale]|bucketSample(5000, 5000, 10000)
```

The range to check wraps around the total bucket range. This means that if you have 100 buckets, and specify a range starting at bucket 70 that is 50 buckets long, this function will check buckets 70-99, and buckets 0-19.

Arguments

- **input** – A value for the bucket sampling to be stable over.
- **start** (*integer*) – The bucket at the start of the range to check. Bucket indexes larger than the total bucket count wrap to the start of the range, e.g. bucket 110 and bucket 10 are the same bucket if the total bucket count is 100.
- **count** (*integer*) – The number of buckets to check, starting at the start bucket. If this is large enough to cause the range to exceed the total number of buckets, the search will wrap to the start of the range again.
- **total** (*integer*) – The number of buckets you want to group users into.

date (*dateString*)

Parses a string as a date and returns a Date object. Date strings should be in ISO 8601 format.

Arguments

- **dateString** (*string*) – String to parse as a date.

```
'2011-10-10T14:48:00'|date // == Date object matching the given date
```

keys (*obj*)

Return an array of the given object's own keys (specifically, its enumerable properties). Similar to Object.keys, except that if given a non-object, `keys` will return `undefined`.

Arguments

- **obj** – Object to get the keys for.

```
// Evaluates to ['foo', 'bar']
{foo: 1, bar:2}|keys
```

Preference Filters

preferenceValue (*prefKey*, *defaultValue*)

Arguments

- **prefKey** (*string*) – Full dotted-path name of the preference to read.
- **defaultValue** – The value to return if the preference does not have a value. Defaults to `undefined`.

Returns The value of the preference.

```
// Match users with more than 2 content processes
'dom.ipc.processCount'|preferenceValue > 2
```

preferenceIsUserSet (*prefKey*)

Arguments

- **prefKey** (*string*) – Full dotted-path name of the preference to read.

Returns `true` if the preference has a value that is different than its default value, or `false` if it does not.

```
// Match users who have modified add-on signature checks
'xpinstall.signatures.required'|preferenceIsUserSet
```

preferenceExists (*prefKey*)

Arguments

- **prefKey** (*string*) – Full dotted-path name of the preference to read.

Returns `true` if the preference has *any* value (whether it is the default value or a user-set value), or `false` if it does not.

```
// Match users with an HTTP proxy
'network.proxy.http'|preferenceExists
```

5.2.5 Examples

This section lists some examples of commonly-used filter expressions.

```
// Match users using the en-US locale
env.locale == 'en-US'

// Match users in any English locale using Firefox Beta
(
   env.locale in ['en-US', 'en-AU', 'en-CA', 'en-GB', 'en-NZ', 'en-ZA']
   && env.channel == 'beta'
)

// Match users located in the US who have Firefox as their default browser
env.country == 'US' && env.isDefaultBrowser

// Match users with the Flash plugin installed. If Flash is missing, the
// plugin list returns `undefined`, which is a falsy value in JavaScript and
// fails the match. Otherwise, it returns a plugin object, which is truthy.
env.plugins['Shockwave Flash']
```

5.3 Advanced: Testing Filter Expressions in the Browser Console

1. Open the browser console
   - Tools > Web Developer > Browser Console
   - `Cmd + Shift + J`
2. Run the following in the console:

```
const { RemoteSettings } = ChromeUtils.import("resource://services-settings/
↪remote-settings.js", {});
const client = RemoteSettings("a-key");
```

   The following lines create a local record with a filter expression field and fetch the current settings list.

```
let FILTER_TO_TEST = `
    env.locale == "fr-FR"
`;

(
  async function () {
    const collection = await client.openCollection();
    await collection.clear();
    await collection.db.saveLastModified(42);

    const record = await collection.create({
      id: "68b19efa-1067-401b-b1c1-8d7b4263bb86",  // random uuidgen
      filter_expression: FILTER_TO_TEST
    }, { synced: true });

    const filtered = await client.get();
    console.log(filtered.length == 1);
  }
)();
```

3. The console will log `true` or `false` depending on whether the expression passed for your client or not.

CHAPTER 6

Screencasts

6.1 Modify records and request review

1. Login as an editor
2. Make some changes in the collection
3. Request review

6.2 Approve changes

1. Login as a review
2. Review the changes
3. Approve the review

6.3 Fetch settings from local server

1. Change the necessary settings
2. Register a callback for the `sync` event
3. Trigger a synchronization manually

CHAPTER 7

Use the Dev Server

7.1 Goals

- Create remote records
- Pull them from the server

7.2 Prerequisites

This guide assumes you have already installed and set up the following:

- cURL
- jq (*optional*)

7.3 Introduction

Remote Settings is built on top of a project called Kinto. As a service to the Kinto community, Mozilla hosts a public instance of Kinto at https://kinto.dev.mozaws.net/v1. Although this server is not officially maintained and has no official connection with the Remote Settings project, it can be convenient to use it when exploring Remote Settings.

Several authentication options are available. We will use local accounts and Basic Auth for the sake of simplicity.

> **Warning:** Since the server is publicly accessible, we flush its content every day. We thus **recommend you to keep the initialization commands in a small shell script**.

7.4 Create your account

Let's create a user `alice` with password `w0nd3rl4nd`.

```
SERVER=https://kinto.dev.mozaws.net/v1

curl -X PUT ${SERVER}/accounts/alice \
     -d '{"data": {"password": "w0nd3rl4nd"}}' \
     -H 'Content-Type:application/json'
```

When reaching out the server root URL with these credentials you should see a `user` entry whose `id` field is `account:alice`.

```
BASIC_AUTH=alice:w0nd3rl4nd

curl -s ${SERVER}/ -u $BASIC_AUTH | jq .user
```

7.5 Create a collection

Choose a name for your settings that makes sense for your use-case and is specific enough (eg. `focus-search-engines`, not `search`).

```
CID=focus-search-engines
```

Using the REST API, we create a collection:

```
curl -X PUT ${SERVER}/buckets/main/collections/${CID} \
     -H 'Content-Type:application/json' \
     -u ${BASIC_AUTH}
```

We create a simple record for testing purposes:

```
curl -X POST ${SERVER}/buckets/main/collections/${CID}/records \
     -d '{"data": {"title": "example"}}' \
     -H 'Content-Type:application/json' \
     -u ${BASIC_AUTH}
```

At this point, the server part is ready: it contains a public collection with one record. You can fetch its records with:

```
curl ${SERVER}/buckets/main/collections/${CID}/records
```

And it should be listed in the monitor/changes endpoint:

```
curl ${SERVER}/buckets/monitor/collections/changes/records
```

7.6 Prepare the client

There is no officially "blessed" way to point the client at the dev server. Unlike other environments, the Remote Settings dev tools cannot be used for this purpose. You can change the `services.settings.server` preference if you like, but because the data in the dev server is not signed, you will get signature verification errors.

Important: This is a critical preference, you should use a dedicated Firefox profile for development.

From your code, or the browser console, register the new collection by listening to the `sync` event:

```
const { RemoteSettings } = ChromeUtils.import("resource://services-settings/remote-
↪settings.js", {});

const client = RemoteSettings("focus-search-engines");

// No signature on Dev Server
client.verifySignature = false;

client.on("sync", ({ data }) => {
  // Dump records titles to stdout
  data.current.forEach(r => dump(`${r.title}\n`));
});
```

7.7 Synchronize manually

Then force a synchronization manually with:

```
await RemoteSettings.pollChanges();
```

Note: Since the developement server is flushed every day, if the client was previously synchronized with data that is not there anymore, the synchronization might fail. You can start from a new profile (`./mach run --temp-profile`) or clear the local state manually (using Remote Settings DevTools or development docs about local data).

See also:

Check out *the dedicated screencast* for this operation!

7.8 Going further

Now that your client can pull data from the server, you can proceed with more advanced stuff like:

- Login on the Admin UI and browse your data
- Create, modify, delete remote records on the server and check out the different `sync` event data attributes
- Define a JSON schema on your collection to validate records and have forms in the Admin UI
- Attach files to your records (see *tutorial*)
- If you feel ready, try out the STAGE environment with VPN access, multi signoff (see *tutorial*), running a *local server* etc.

CHAPTER 8

Setup a Local Server

8.1 Goals

- Run a local server
- Pull data from it
- Setup advanced features like multi signoff and signatures

8.2 Prerequisites

This guide assumes you have already installed and set up the following:

- cURL
- Docker

8.3 Introduction

There are several ways to run a local instance of Kinto, the underlying software of Remote Settings.

We will use Docker for the sake of simplicity, but you may find more convenient to install the Python package for example.

8.4 Quick start

We will run a local container with the minimal configuration. It should be enough to hack on your Remote Settings integration in the plane. However if your goal is to setup a local server that has the same signoff features as STAGE and PROD, you can continue into the configuration of the next section.

Pull the Docker container:

```
docker pull mozilla/kinto-dist
```

Create a configuration file `server.ini` with the following content:

```
[app:main]
use = egg:kinto
kinto.includes = kinto.plugins.admin
                 kinto.plugins.accounts
                 kinto.plugins.history
                 kinto_changes
                 kinto_attachment
                 kinto_signer

kinto.storage_backend = kinto.core.storage.memory
kinto.storage_url =
kinto.cache_backend = kinto.core.cache.memory
kinto.cache_url =
kinto.permission_backend = kinto.core.permission.memory
kinto.permission_url =

multiauth.policies = account
multiauth.policy.account.use = kinto.plugins.accounts.authentication.
↪AccountsAuthenticationPolicy
kinto.userid_hmac_secret =␣
↪284461170acd78f0be0827ef514754937474d7c922191e4f78be5c1d232b38c4

kinto.bucket_create_principals = system.Authenticated
kinto.account_create_principals = system.Everyone
kinto.account_write_principals = account:admin

kinto.experimental_permissions_endpoint = true
kinto.experimental_collection_schema_validation = true
kinto.changes.resources = /buckets/main
kinto.attachment.base_path = /tmp/attachments
kinto.attachment.base_url =
kinto.attachment.extra.base_url = http://localhost:8888/attachments
kinto.attachment.folder = {bucket_id}/{collection_id}
kinto.signer.resources = /buckets/main-workspace -> /buckets/main-preview -> /buckets/
↪main
kinto.signer.group_check_enabled = true
kinto.signer.to_review_enabled = true
kinto.signer.signer_backend = kinto_signer.signer.autograph
kinto.signer.main-workspace.editors_group = {collection_id}-editors
kinto.signer.main-workspace.reviewers_group = {collection_id}-reviewers
kinto.signer.autograph.server_url = http://autograph-server:8000
# Use credentials from https://github.com/mozilla-services/autograph/blob/5b4a473/
↪autograph.yaml
kinto.signer.autograph.hawk_id = kintodev
kinto.signer.autograph.hawk_secret =␣
↪3isey64n25fim18chqgewirm6z2gwva1mas0eu71e9jtisdwv6bd

[uwsgi]
wsgi-file = app.wsgi
enable-threads = true
http-socket = 0.0.0.0:8888
processes =  1
```

(continues on next page)

(continued from previous page)

```
master = true
module = kinto
harakiri = 120
uid = kinto
gid = kinto
lazy = true
lazy-apps = true
single-interpreter = true
buffer-size = 65535
post-buffering = 65535
static-map = /attachments=/tmp/attachments

[loggers]
keys = root, kinto

[handlers]
keys = console

[formatters]
keys = color

[logger_root]
level = INFO
handlers = console

[logger_kinto]
level = DEBUG
handlers = console
qualname = kinto

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = color

[formatter_color]
class = logging_color_formatter.ColorFormatter
```

Create a local folder to receive the potential records attachments, Docker should have the permissions to write it:

```
mkdir --mode=777 attachments  # world writable
```

Now, we will run the container with the local configuration file and attachments folder mounted:

```
docker run -v `pwd`/server.ini:/etc/kinto.ini \
           -v `pwd`/attachments:/tmp/attachments \
           -e KINTO_INI=/etc/kinto.ini \
           -p 8888:8888 \
           mozilla/kinto-dist
```

Your local instance should now be running at http://localhost:8888/v1 and the Admin UI available at http://localhost:8888/v1/admin/

8.4.1 Create basic objects

Let's create an `admin` user:

```
SERVER=http://localhost:8888/v1

curl -X PUT ${SERVER}/accounts/admin \
     -d '{"data": {"password": "s3cr3t"}}' \
     -H 'Content-Type:application/json'
```

And a `main` bucket, that is publicly readable and where authenticated users can create collections:

```
BASIC_AUTH=admin:s3cr3t

curl -X PUT ${SERVER}/buckets/main \
     -d '{"permissions": {"read": ["system.Everyone"], "collection:create": ["system.
↪Authenticated"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH
```

Now your local server will roughly behave like the dev server, you can jump to *the other tutorial* in order to create remote records and synchronize locally.

8.5 Configure multi-signoff

In this section, we will have a local setup that enables multi-signoff and interacts with an Autograph instance in order to sign the data.

First, run the Autograph container in a separate terminal:

```
docker run --rm --name autograph-server mozilla/autograph
```

Autograph generates the `x5u` certificate chains on startup. In order to have them available to download from Firefox, let's copy them out of the container.

First, look up the certificate filename using `ls` from within the container:

```
docker exec -i -t autograph-server '/bin/sh'
$ ls /tmp/autograph/chains/remotesettingsdev/
remote-settings.content-signature.mozilla.org-20190503.chain
$ ^C
```

Then, copy the file from the container into the host:

```
mkdir -p /tmp/autograph/chains/remotesettingsdev/
docker cp autograph-server:/tmp/autograph/chains/remotesettingsdev/remote-settings.
↪content-signature.mozilla.org-20190503.chain /tmp/autograph/chains/
↪remotesettingsdev/
```

And run the Remote Settings server with a link to `autograph-server` container:

```
docker run -v `pwd`/server.ini:/etc/kinto.ini \
           --link autograph-server:autograph-server \
           -e KINTO_INI=/etc/kinto.ini \
           -p 8888:8888 \
           mozilla/kinto-dist
```

Both containers should be connected, and the heartbeat endpoint should only return positive checks:

```
curl http://localhost:8888/v1/__heartbeat__

{"attachments":true, "cache":true, "permission":true, "signer": true, "storage":true}
```

In the previous section we were using the `main` bucket directly, but in this setup, we will create the collections in the `main-workspace` bucket. Data will be automatically copied to the `main-preview` and `main` when requesting review and approving changes during the multi-signoff workflow.

We'll use the same `admin` user:

```
curl -X PUT ${SERVER}/accounts/admin \
     -d '{"data": {"password": "s3cr3t"}}' \
     -H 'Content-Type:application/json'
```

The `main-workspace` bucket allows any authenticated user to create collections (like on STAGE):

```
BASIC_AUTH=admin:s3cr3t

curl -X PUT ${SERVER}/buckets/main-workspace \
     -d '{"permissions": {"collection:create": ["system.Authenticated"], "group:create
↪": ["system.Authenticated"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH
```

The `main-preview` and `main` buckets are (re)initialized with read-only permissions:

```
curl -X PUT ${SERVER}/buckets/main-preview \
     -d '{"permissions": {"read": ["system.Everyone"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH

curl -X PUT ${SERVER}/buckets/main \
     -d '{"permissions": {"read": ["system.Everyone"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH
```

8.6 Prepare the client

The official way to point the client at another server is using the Remote Settings dev tools. This tool can set the constellation of preferences necessary to operate correctly with your local server.

See also:

Check out *the dedicated screencast* for this operation!

8.7 What's next?

- Create a collection in the `main-workspace` bucket
- Assign users to editors and reviewers groups
- Create records, request review, preview changes in the browser, approve the changes

We cover that in *the dedicated multi-signoff tutorial*.

CHAPTER 9

Multi Signoff Workflow

9.1 Goals

- Create some records
- Request review
- Preview changes in the browser
- Approve/Decline the changes

9.2 Prerequisites

This guide assumes you have already installed and set up the following:

- cURL
- jq (*optional*)
- *a local instance* with multi signoff enabled or access/contact with two users that have permissions on STAGE/PROD

We'll refer the running instance as `$SERVER` (eg. `http://localhost:8888/v1` or `https://settings-writer.prod.mozaws.net/v1` via the VPN).

Note: If you need to give additional users access to your collection on STAGE/PROD you must edit the *collection manifest*.

9.3 Introduction

Multi signoff basically consists in 3 steps:

1. Editors create/update/delete records on the `main-workspace` bucket
2. Editors request review. The changes are automatically published in the `main-preview` bucket.
3. Reviewers can configure their browser to preview the changes, and will approve (or decline) the review request. If approved, the changes are published in the `main` bucket.

See also:

If you're interested by workflows in the Admin UI, check out *the screencasts* instead!

9.3.1 Create some users

If you're not using STAGE or PROD, we'll need to create some `reviewer` and `editor` accounts on the server. We'll reuse the `admin` superuser seen in previous tutorials.

```
curl -X PUT ${SERVER}/accounts/editor \
     -d '{"data": {"password": "3d1t0r"}}' \
     -H 'Content-Type:application/json'

curl -X PUT ${SERVER}/accounts/reviewer \
     -d '{"data": {"password": "r3v13w3r"}}' \
     -H 'Content-Type:application/json'
```

Note: In STAGE or PROD, humans authenticate via LDAP/OpenID Connect. But scripted/scheduled tasks can also have their dedicated account like above. Ask us!

9.4 Create a collection

The `main-workspace` bucket is where every edit happens.

We first have to create a new collection (eg. `password-recipes`). We'll use the `editor` account:

```
curl -X PUT ${SERVER}/buckets/main-workspace/collections/password-recipes \
     -H 'Content-Type:application/json' \
     -u editor:3d1t0r
```

Note: In PROD, only administrators are allowed to create collections, and the *request is made via Bugzilla*.

Now that we created this collection, two groups should have been created automatically. Check their presence and content with:

```
curl -s ${SERVER}/buckets/main-workspace/groups/password-recipes-editors | jq
curl -s ${SERVER}/buckets/main-workspace/groups/password-recipes-reviewers | jq
```

9.5 Manage reviewers

Only the members of the `password-recipes-editors` group are allowed to request reviews for the records changes.

Only the members of the `password-recipes-reviewers` group are allowed to approve/decline them.

We will add our `reviewer` user above to the `password-recipes-reviewers` group with this JSON PATCH request:

```
curl -X PATCH $SERVER/buckets/main-workspace/groups/password-recipes-reviewers \
     -H 'Content-Type:application/json-patch+json' \
     -d '[{ "op": "add", "path": "/data/members/0", "value": "account:reviewer" }]' \
     -u editor:3d1t0r
```

Note: When using internal accounts the, user IDs are prefixed with `account:`. In STAGE/PROD, most user IDs look like this: `ldap:jdoe@mozilla.com`.

9.6 Change records and request review

See also:

Check out *the dedicated screencast* for the equivalent with the Admin UI!

Create (or update or delete) some records:

```
for i in `seq 1 10`; do
    curl -X POST ${SERVER}/buckets/main-workspace/collections/password-recipes/
↪records \
         -H 'Content-Type:application/json' \
         -d "{\"data\": {\"property\": $i}}" \
         -u editor:3d1t0r
done
```

And request review:

```
curl -X PATCH ${SERVER}/buckets/main-workspace/collections/password-recipes \
        -H 'Content-Type:application/json' \
        -d '{"data": {"status": "to-review"}}' \
        -u editor:3d1t0r
```

At this point the changes were published to the `main-preview` bucket, which is publicly readable:

```
curl -s ${SERVER}/buckets/main-preview/collections/password-recipes/records | jq
```

The collection metadata now contain some signature information:

```
curl -s ${SERVER}/buckets/main-preview/collections/password-recipes | jq .data.
↪signature
```

The monitor/changes endpoint mentions the new collection `password-recipes`:

```
curl -s ${SERVER}/buckets/monitor/collections/changes/records | jq
```

9.7 Preview changes in the browser

Important: It is recommended to use the Remote Settings DevTools instead of changing preferences manually.

The following preferences must be changed to the following values in `about:config`:

- `services.settings.server` : `http://localhost:8888/v1`
- `services.settings.default_bucket` : `main-preview`

From your code, or the browser console, register the new collection by listening to the `sync` event and trigger synchronization:

```
const { RemoteSettings } = ChromeUtils.import("resource://services-settings/remote-
↪settings.js", {});

RemoteSettings("password-recipes").on("sync", ({ data }) => {
  data.current.forEach(r => dump(`${r.property}\n`));
});
```

Then force a synchronization manually with:

```
RemoteSettings.pollChanges();
```

9.8 Approve/Decline changes

See also:

Check out *the dedicated screencast* for the equivalent with the Admin UI!

Using the `reviewer` authentication, change the collection status to either `to-sign` (approve) or `work-in-progress` (decline).

```
curl -X PATCH ${SERVER}/buckets/main-workspace/collections/password-recipes \
     -H 'Content-Type:application/json' \
     -d '{"data": {"status": "to-sign"}}' \
     -u reviewer:r3v13w3r
```

At this point the changes were published to the `main` bucket, which is publicly readable:

```
curl -s ${SERVER}/buckets/main/collections/password-recipes/records | jq
```

The main collection metadata now contain some signature information:

```
curl -s ${SERVER}/buckets/main/collections/password-recipes | jq .data.signature
```

In the browser, the following preferences must be reset to their default value:

- `services.settings.default_bucket` : `main`

CHAPTER 10

Work with Attachments

10.1 Goals

- Publish large or binary content

10.2 Prerequisites

This guide assumes that you have already installed the following commands:

- cURL
- uuidgen
- jq (*optional*)

And that you are familiar with the Remote Settings API, at least on the dev server.

We'll refer the running instance as `$SERVER` (eg. `https://kinto.dev.mozaws.net/v1`).

10.3 Introduction

Files can be attached to records. When a record has a file attached to it, it has an `attachment` attribute, which contains the file related information (url, hash, size, mimetype, etc.).

The Remote Settings client API is **not in charge** of downloading the remote files during synchronization.

However, a helper is available on the client instance.

During synchronization, only the records that changed are fetched. Depending on your implementation, attachments may have to be redownloaded completely even if only a few bytes were changed.

10.4 Publish records with attachments

Files can be attached to existing records or records can be created when uploading the attachment.

Suppose that we want to attach a file (`/home/mathieu/DAFSA.bin`) to the existing record `bec3e95c-4d28-40c1-b486-76682962861f`:

```
BUCKET=main-workspace # (or just ``main`` in Dev)
COLLECTION=public-suffix-list
RECORD=bec3e95c-4d28-40c1-b486-76682962861f
FILEPATH=/home/mathieu/DAFSA.bin

curl -X POST ${SERVER}/buckets/${BUCKET}/collections/${COLLECTION}/records/${RECORD}/
↪attachment \
     -H 'Content-Type:multipart/form-data' \
     -F attachment=@$FILEPATH \
     -u user:pass
```

And in order to create a record with both attributes and attachment, you'll have a generate a record id yourself.

```
RECORD=`uuidgen`

curl -X POST ${SERVER}/buckets/${BUCKET}/collections/${COLLECTION}/records/${RECORD}/
↪attachment \
     -H 'Content-Type:multipart/form-data' \
     -F attachment=@$FILEPATH \
     -F 'data={"name": "Mac Fly", "age": 42}' \
     -u user:pass
```

Note: Since the dev server is open to anyone and runs on `.mozaws.net`, we only allow certain types of files (images, audio, video, archives, `.bin`, `.json`, `.gz`).

If you need to uploadd files with specific extension, let us know and we add it to the whitelist (except `.html`, `.js`).

10.5 Synchronize attachments

Attachments can be downloaded when the `"sync"` event is received.

```
const client = RemoteSettings("a-key");

client.on("sync", async ({ data: { created, updated, deleted } }) => {
  const toDownload = created
    .concat(updated.map(u => u.new))
    .filter(d => d.attachment);

  // Download attachments
  const fileURLs = await Promise.all(
    toDownload.map(entry => client.attachments.download(entry, { retries: 2 }))
  );

  // Open downloaded files...
  const fileContents = await Promise.all(
    fileURLs.map(async url => {
```

(continues on next page)

(continued from previous page)

```
      const r = await fetch(url);
      return r.blob();
    })
  );
});
```

See more details in client documentation.

10.6 About compression

The server does not compress the files.

We plan to enable compression at the HTTP level (Bug 1339114) for when clients fetch the attachment using the `Accept-Encoding: gzip` request header.

10.7 In the admin tool

The Remote Settings administration tool supports attachments as well. If a collection has a record schema and attachments are "enabled" for that collection, then editors will be able to upload attachments as part of editing records.

The controls for attachments in a given collection are in the `attachment` field in the collection metadata (probably located in the remote-settings-permissions repo). The `attachment` attribute should be an object and it can have the following properties:

- `enabled`: boolean, true to enable attachments for this collection
- `required`: boolean, true if records in this collection must have an attachment

CHAPTER 11

Normandy Integration

11.1 Goals

- Synchronize settings on certain clients only
- Settings available only temporarily

Note: This differs from *JEXL filters*, with which all records are synchronized but listing them locally returns a filtered set.

11.2 Introduction

When a collection is published on the server, it get pulled during synchronization if at least one the following conditions is met:

- There is an instantiated client — ie. a call to `RemoteSettings("cid")` was done earlier
- Some local data exists in the internal IndexedDB — ie. it was pulled once already
- A JSON dump was shipped in mozilla-central for this collection — in `services/settings/dumps/`

Basically, here we will leverage the fact that **if the client is never instantiated, then it will never get synchronized**, and thus will never have any local data.

11.3 Disabled by default

Instantiating a client conditionnaly using a preference whose default value is `false` does the trick! By default, users won't synchronize this collection data.

```
if (Services.prefs.getBoolPref("my-feature-pref", false)) {
    const client = RemoteSettings("cid");
    const records = await client.get();
}
```

11.4 Pref Flip

Using Normandy preference experiments, you can flip the above preference to `true` for a sub-population of users, or temporarily etc. (using JEXL filters BTW).

When the experiment will be *enabled* on the targeted users, the client will be instantiated and the synchronization of the collection data will take place.

11.5 Clean-up

Once the experiment is switched back to *disabled*, the local data should be deleted. We will use a preference observer to detect that the preference is switched back to `false`:

```
Services.prefs.addObserver("my-feature-pref", {
  async observe(aSubject, aTopic, aData) {
    if (!Services.prefs.getBoolPref(aData)) {
      // Pref was switched to false, clean-up local IndexedDB data.
      const collection = await RemoteSetttings("cid").openCollection();
      await collection.clear();
    }
  }
});
```

You can also open a ticket to request the deletion of the collection from the server.

CHAPTER 12

Kinto Admin Development

12.1 Goals

- Development environment for Kinto Admin
- Connect to a local Remote Settings server
- Contribute patches

12.2 Prerequisites

This guide assumes you have already installed and set up the following:

- *Setup a Local Server* with multi-signoff

12.3 Kinto Admin

This part is very classic. We recommend using NVM in order to have recent versions of Node and NPM.

The UI should be accessible at http://0.0.0.0:3000

12.4 Initialization script

Since the container is not configured with a real database by default, the content is flushed on each restart.

This means you will have to populate data regularly.

We'll create a small bash script `init.sh` with the following commands:

```
set -e
set -o pipefail

SERVER=http://localhost:8888/v1
```

In the *prerequisite tutorial*, an `admin` user was created, as well as the basic buckets. Let's add that to our script:

```
curl -X PUT --fail --verbose ${SERVER}/accounts/admin \
     -d '{"data": {"password": "s3cr3t"}}' \
     -H 'Content-Type:application/json'

BASIC_AUTH=admin:s3cr3t

curl -X PUT --fail --verbose ${SERVER}/buckets/main-workspace \
     -d '{"permissions": {"collection:create": ["system.Authenticated"], "group:create
↪": ["system.Authenticated"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH

curl -X PUT --fail --verbose ${SERVER}/buckets/main-preview \
     -d '{"permissions": {"read": ["system.Everyone"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH

curl -X PUT --fail --verbose ${SERVER}/buckets/main \
     -d '{"permissions": {"read": ["system.Everyone"]}}' \
     -H 'Content-Type:application/json' \
     -u $BASIC_AUTH
```

In order to play with multi-signoff, we'll create an `editor` and a `reviewer` accounts, put these lines in the `init.sh` script.

```
curl -X PUT --fail --verbose ${SERVER}/accounts/editor \
     -d '{"data": {"password": "3d1t0r"}}' \
     -H 'Content-Type:application/json'

curl -X PUT --fail --verbose ${SERVER}/accounts/reviewer \
     -d '{"data": {"password": "r3v13w3r"}}' \
     -H 'Content-Type:application/json'
```

Now create a collection, with a dedicated reviewer group:

```
curl -X PUT --fail --verbose ${SERVER}/buckets/main-workspace/collections/password-
↪recipes \
     -H 'Content-Type:application/json' \
     -u editor:3d1t0r


curl -X PATCH --fail --verbose $SERVER/buckets/main-workspace/groups/password-recipes-
↪reviewers \
     -H 'Content-Type:application/json-patch+json' \
     -d '[{ "op": "add", "path": "/data/members/0", "value": "account:reviewer" }]' \
     -u $BASIC_AUTH
```

And at last, create some records, request review and approve changes.

```
for i in `seq 1 10`; do
    curl -X POST --fail --verbose ${SERVER}/buckets/main-workspace/collections/
↪password-recipes/records \
         -H 'Content-Type:application/json' \
         -d "{\"data\": {\"property\": $i}}" \
         -u editor:3d1t0r
done

curl -X PATCH --fail --verbose ${SERVER}/buckets/main-workspace/collections/password-
↪recipes \
        -H 'Content-Type:application/json' \
        -d '{"data": {"status": "to-review"}}' \
        -u editor:3d1t0r

curl -X PATCH --fail --verbose ${SERVER}/buckets/main-workspace/collections/password-
↪recipes \
     -H 'Content-Type:application/json' \
     -d '{"data": {"status": "to-sign"}}' \
     -u reviewer:r3v13w3r

echo ""
echo "Done."
```

With the service running locally, populating it should now just consist in running:

```
bash init.sh
```

12.5 Connect Admin UI

On http://0.0.0.0:3000, when specifying http://0.0.0.0:8888/v1 in the *Server URL*, the option to login with *Kinto Account* should be shown.

Using Container Tabs in Firefox, you can have one tab logged as `editor` with password `3d1t0r` and another one with `reviewer` and `r3v13w3r`.

12.6 Submit Patches

Development happens on Github.

The process for a patch to reach Remote Settings is the following:

- Get the patch merged on Kinto/kinto-admin
- Release a new version of kinto-admin on NPM
- Upgrade the kinto-admin plugin in Kinto (example PR)
- Release a new version of Kinto
- Upgrade Kinto in kinto-dist
- Release a new version of kinto-dist
- STAGE is deployed automatically when a new tag of kinto-dist is published
- Request a deployment in PROD on Bugzilla (example)

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Index