```
/* vim: set ts=2 sw=2 sts=2 et tw=80: */
/* This Source Code Form is subject to the terms of the Mozilla Public
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/. */
"use strict";

const { XPCOMUtils } = ChromeUtils.importESModule(
  "resource://gre/modules/XPCOMUtils.sys.mjs"
);
const { AppConstants } = ChromeUtils.importESModule(
  "resource://gre/modules/AppConstants.sys.mjs"
);
const lazy = {};

ChromeUtils.defineESModuleGetters(lazy, {
  Downloader: "resource://services-settings/Attachments.sys.mjs",
  ExperimentAPI: "resource://nimbus/ExperimentAPI.sys.mjs",
  KintoHttpClient: "resource://services-common/kinto-http-client.sys.mjs",
  MacAttribution: "resource:///modules/MacAttribution.sys.mjs",
  NimbusFeatures: "resource://nimbus/ExperimentAPI.sys.mjs",
  PanelTestProvider: "resource://activity-stream/lib/PanelTestProvider.sys.mjs",
  RemoteL10n: "resource://activity-stream/lib/RemoteL10n.sys.mjs",
  SnippetsTestMessageProvider:
    "resource://activity-stream/lib/SnippetsTestMessageProvider.sys.mjs",
  SpecialMessageActions:
    "resource://messaging-system/lib/SpecialMessageActions.sys.mjs",
  TargetingContext: "resource://messaging-system/targeting/Targeting.sys.mjs",
  Utils: "resource://services-settings/Utils.sys.mjs",
  setTimeout: "resource://gre/modules/Timer.sys.mjs",
});

XPCOMUtils.defineLazyModuleGetters(lazy, {
  Spotlight: "resource://activity-stream/lib/Spotlight.jsm",
  ToastNotification: "resource://activity-stream/lib/ToastNotification.jsm",
  ToolbarBadgeHub: "resource://activity-stream/lib/ToolbarBadgeHub.jsm",
  ToolbarPanelHub: "resource://activity-stream/lib/ToolbarPanelHub.jsm",
  MomentsPageHub: "resource://activity-stream/lib/MomentsPageHub.jsm",
  InfoBar: "resource://activity-stream/lib/InfoBar.jsm",
  ASRouterTargeting: "resource://activity-stream/lib/ASRouterTargeting.jsm",
  ASRouterPreferences: "resource://activity-stream/lib/ASRouterPreferences.jsm",
  TARGETING_PREFERENCES:
    "resource://activity-stream/lib/ASRouterPreferences.jsm",
  ASRouterTriggerListeners:
    "resource://activity-stream/lib/ASRouterTriggerListeners.jsm",
});

XPCOMUtils.defineLazyServiceGetters(lazy, {
  BrowserHandler: ["@mozilla.org/browser/clh;1", "nsIBrowserHandler"],
});
const { actionCreators: ac } = ChromeUtils.importESModule(
  "resource://activity-stream/common/Actions.sys.mjs"
);

const { CFRMessageProvider } = ChromeUtils.importESModule(
  "resource://activity-stream/lib/CFRMessageProvider.sys.mjs"
);
const { OnboardingMessageProvider } = ChromeUtils.import(
  "resource://activity-stream/lib/OnboardingMessageProvider.jsm"
);
const { RemoteSettings } = ChromeUtils.importESModule(
  "resource://services-settings/remote-settings.sys.mjs"
);
const { CFRPageActions } = ChromeUtils.import(
  "resource://activity-stream/lib/CFRPageActions.jsm"
);
const { AttributionCode } = ChromeUtils.importESModule(
  "resource:///modules/AttributionCode.sys.mjs"
);

// List of hosts for endpoints that serve router messages.
// Key is allowed host, value is a name for the endpoint host.
const DEFAULT_ALLOWLIST_HOSTS = {
  "activity-stream-icons.services.mozilla.com": "production",
  "snippets-admin.mozilla.org": "preview",
};
const SNIPPETS_ENDPOINT_ALLOWLIST =
  "browser.newtab.activity-stream.asrouter.allowHosts";
// Max possible impressions cap for any message
const MAX_MESSAGE_LIFETIME_CAP = 100;

const LOCAL_MESSAGE_PROVIDERS = {
  OnboardingMessageProvider,
  CFRMessageProvider,
};
const STARTPAGE_VERSION = "6";
```

```
// Remote Settings
const RS_MAIN_BUCKET = "main";
const RS_COLLECTION_L10N = "ms-language-packs"; // "ms" stands for Messaging System
const RS_PROVIDERS_WITH_L10N = ["cfr"];
const RS_FLUENT_VERSION = "v1";
const RS_FLUENT_RECORD_PREFIX = `cfr-${RS_FLUENT_VERSION}`;
const RS_DOWNLOAD_MAX_RETRIES = 2;
// This is the list of providers for which we want to cache the targeting
// expression result and reuse between calls. Cache duration is defined in
// ASRouterTargeting where evaluation takes place.
const JEXL_PROVIDER_CACHE = new Set(["snippets"]);

// To observe the app locale change notification.
const TOPIC_INTL_LOCALE_CHANGED = "intl:app-locales-changed";
const TOPIC_EXPERIMENT_ENROLLMENT_CHANGED = "nimbus:enrollments-updated";
// To observe the pref that controls if ASRouter should use the remote Fluent files for l10n.
const USE_REMOTE_L10N_PREF =
  "browser.newtabpage.activity-stream.asrouter.useRemoteL10n";

const MESSAGING_EXPERIMENTS_DEFAULT_FEATURES = [
  "cfr",
  "fxms-message-1",
  "fxms-message-2",
  "fxms-message-3",
  "fxms-message-4",
  "fxms-message-5",
  "fxms-message-6",
  "fxms-message-7",
  "fxms-message-8",
  "fxms-message-9",
  "fxms-message-10",
  "fxms-message-11",
  "infobar",
  "moments-page",
  "pbNewtab",
  "spotlight",
];

// Experiment groups that need to report the reach event in Messaging-Experiments.
// If you're adding new groups to it, make sure they're also added in the
// `messaging_experiments.reach.objects` defined in "toolkit/components/telemetry/Events.yaml"
const REACH_EVENT_GROUPS = ["cfr", "moments-page", "infobar", "spotlight"];
const REACH_EVENT_CATEGORY = "messaging_experiments";
const REACH_EVENT_METHOD = "reach";

const MessageLoaderUtils = {
  STARTPAGE_VERSION,
  REMOTE_LOADER_CACHE_KEY: "RemoteLoaderCache",
  _errors: [],

  reportError(e) {
    console.error(e);
    this._errors.push({
      timestamp: new Date(),
      error: { message: e.toString(), stack: e.stack },
    });
  },

  get errors() {
    const errors = this._errors;
    this._errors = [];
    return errors;
  },

  /**
   * _localLoader - Loads messages for a local provider (i.e. one that lives in mozilla central)
   *
   * @param {obj} provider An AS router provider
   * @param {Array} provider.messages An array of messages
   * @returns {Array} the array of messages
   */
  _localLoader(provider) {
    return provider.messages;
  },

  async _remoteLoaderCache(storage) {
    let allCached;
    try {
      allCached =
        (await storage.get(MessageLoaderUtils.REMOTE_LOADER_CACHE_KEY)) || {};
    } catch (e) {
      // istanbul ignore next
      MessageLoaderUtils.reportError(e);
      // istanbul ignore next
```

```
    allCached = {};
  }
  return allCached;
},

/**
 * _remoteLoader - Loads messages for a remote provider
 *
 * @param {obj} provider An AS router provider
 * @param {string} provider.url An endpoint that returns an array of messages as JSON
 * @param {obj} options.storage A storage object with get() and set() methods for caching.
 * @returns {Promise} resolves with an array of messages, or an empty array if none could be fetched
 */
async _remoteLoader(provider, options) {
  let remoteMessages = [];
  if (provider.url) {
    const allCached = await MessageLoaderUtils._remoteLoaderCache(
      options.storage
    );
    const cached = allCached[provider.id];
    let etag;

    if (
      cached &&
      cached.url === provider.url &&
      cached.version === STARTPAGE_VERSION
    ) {
      const { lastFetched, messages } = cached;
      if (
        !MessageLoaderUtils.shouldProviderUpdate({
          ...provider,
          lastUpdated: lastFetched,
        })
      ) {
        // Cached messages haven't expired, return early.
        return messages;
      }
      etag = cached.etag;
      remoteMessages = messages;
    }

    let headers = new Headers();
    if (etag) {
      headers.set("If-None-Match", etag);
    }

    let response;
    try {
      response = await fetch(provider.url, {
        headers,
        credentials: "omit",
      });
    } catch (e) {
      MessageLoaderUtils.reportError(e);
    }
    if (
      response &&
      response.ok &&
      response.status >= 200 &&
      response.status < 400
    ) {
      let jsonResponse;
      try {
        jsonResponse = await response.json();
      } catch (e) {
        MessageLoaderUtils.reportError(e);
        return remoteMessages;
      }
      if (jsonResponse && jsonResponse.messages) {
        remoteMessages = jsonResponse.messages.map(msg => ({
          ...msg,
          provider_url: provider.url,
        }));

        // Cache the results if this isn't a preview URL.
        if (provider.updateCycleInMs > 0) {
          etag = response.headers.get("ETag");
          const cacheInfo = {
            messages: remoteMessages,
            etag,
            lastFetched: Date.now(),
            version: STARTPAGE_VERSION,
          };

          options.storage.set(MessageLoaderUtils.REMOTE_LOADER_CACHE_KEY, {
```

```
            ...allCached,
            [provider.id]: cacheInfo,
          });
        }
      } else {
        MessageLoaderUtils.reportError(
          `No messages returned from ${provider.url}.`
        );
      }
    } else if (response) {
      MessageLoaderUtils.reportError(
        `Invalid response status ${response.status} from ${provider.url}.`
      );
    }
  }
  return remoteMessages;
},

/**
 * _remoteSettingsLoader - Loads messages for a RemoteSettings provider
 *
 * Note:
 * 1). The "cfr" provider requires the Fluent file for l10n, so there is
 * another file downloading phase for those two providers after their messages
 * are successfully fetched from Remote Settings. Currently, they share the same
 * attachment of the record "${RS_FLUENT_RECORD_PREFIX}-${locale}" in the
 * "ms-language-packs" collection. E.g. for "en-US" with version "v1",
 * the Fluent file is attched to the record with ID "cfr-v1-en-US".
 *
 * 2). The Remote Settings downloader is able to detect the duplicate download
 * requests for the same attachment and ignore the redundent requests automatically.
 *
 * @param {object} provider An AS router provider
 * @param {string} provider.id The id of the provider
 * @param {string} provider.collection Remote Settings collection name
 * @param {object} options
 * @param {function} options.dispatchCFRAction Action handler function
 * @returns {Promise<object[]>} Resolves with an array of messages, or an
 *                              empty array if none could be fetched
 */
async _remoteSettingsLoader(provider, options) {
  let messages = [];
  if (provider.collection) {
    try {
      messages = await MessageLoaderUtils._getRemoteSettingsMessages(
        provider.collection
      );
      if (!messages.length) {
        MessageLoaderUtils._handleRemoteSettingsUndesiredEvent(
          "ASR_RS_NO_MESSAGES",
          provider.id,
          options.dispatchCFRAction
        );
      } else if (
        RS_PROVIDERS_WITH_L10N.includes(provider.id) &&
        lazy.RemoteL10n.isLocaleSupported(MessageLoaderUtils.locale)
      ) {
        const recordId = `${RS_FLUENT_RECORD_PREFIX}-${MessageLoaderUtils.locale}`;
        const kinto = new lazy.KintoHttpClient(lazy.Utils.SERVER_URL);
        const record = await kinto
          .bucket(RS_MAIN_BUCKET)
          .collection(RS_COLLECTION_L10N)
          .getRecord(recordId);
        if (record && record.data) {
          const downloader = new lazy.Downloader(
            RS_MAIN_BUCKET,
            RS_COLLECTION_L10N,
            "browser",
            "newtab"
          );
          // Await here in order to capture the exceptions for reporting.
          await downloader.downloadToDisk(record.data, {
            retries: RS_DOWNLOAD_MAX_RETRIES,
          });
          lazy.RemoteL10n.reloadL10n();
        } else {
          MessageLoaderUtils._handleRemoteSettingsUndesiredEvent(
            "ASR_RS_NO_MESSAGES",
            RS_COLLECTION_L10N,
            options.dispatchCFRAction
          );
        }
      }
    } catch (e) {
      MessageLoaderUtils._handleRemoteSettingsUndesiredEvent(
```

```
        "ASR_RS_ERROR",
        provider.id,
        options.dispatchCFRAction
      );
      MessageLoaderUtils.reportError(e);
    }
  }
  return messages;
},

/**
 * Fetch messages from a given collection in Remote Settings.
 *
 * @param {string} collection The remote settings collection identifier
 * @returns {Promise<object[]>} Resolves with an array of messages
 */
_getRemoteSettingsMessages(collection) {
  return RemoteSettings(collection).get();
},

/**
 * Return messages from active Nimbus experiments and rollouts.
 *
 * @param {object} provider A messaging experiments provider.
 * @param {string[]?} provider.featureIds
 *                    An optional array of Nimbus feature IDs to check for
 *                    enrollments. If not provided, we will fall back to the
 *                    set of default features. Otherwise, if provided and
 *                    empty, we will not ingest messages from any features.
 *
 * @return {object[]} The list of messages from active enrollments, as well as
 *                    the messages defined in unenrolled branches so that they
 *                    reach events can be recorded (if we record reach events
 *                    for that feature).
 */
async _experimentsAPILoader(provider) {
  // Allow tests to override the set of featureIds
  const featureIds = Array.isArray(provider.featureIds)
    ? provider.featureIds
    : MESSAGING_EXPERIMENTS_DEFAULT_FEATURES;
  let experiments = [];
  for (const featureId of featureIds) {
    const featureAPI = lazy.NimbusFeatures[featureId];
    const experimentData = lazy.ExperimentAPI.getExperimentMetaData({
      featureId,
    });

    // We are not enrolled in any experiment or rollout for this feature, so
    // we can skip the feature.
    if (
      !experimentData &&
      !lazy.ExperimentAPI.getRolloutMetaData({ featureId })
    ) {
      continue;
    }

    const featureValue = featureAPI.getAllVariables();

    // If the value is a multi-message config, add each message in the
    // messages array. Cache the Nimbus feature ID on each message, because
    // there is not a 1-1 correspondance between templates and features.
    // This is used when recording expose events (see |sendTriggerMessage|).
    const messages =
      featureValue?.template === "multi" &&
      Array.isArray(featureValue.messages)
        ? featureValue.messages
        : [featureValue];
    for (const message of messages) {
      if (message?.id) {
        message._nimbusFeature = featureId;
        experiments.push(message);
      }
    }

    // Add Reach messages from unenrolled sibling branches, provided we are
    // recording Reach events for this feature. If we are in a rollout, we do
    // not have sibling branches.
    if (!REACH_EVENT_GROUPS.includes(featureId) || !experimentData) {
      continue;
    }

    // Check other sibling branches for triggers, add them to the return array
    // if found any. The `forReachEvent` label is used to identify those
    // branches so that they would only be used to record the Reach event.
    const branches =
```

```
      (await lazy.ExperimentAPI.getAllBranches(experimentData.slug)) || [];
    for (const branch of branches) {
      let branchValue = branch[featureId].value;
      if (!branchValue || branch.slug === experimentData.branch.slug) {
        continue;
      }
      const branchMessages =
        branchValue?.template === "multi" &&
        Array.isArray(branchValue.messages)
          ? branchValue.messages
          : [branchValue];
      for (const message of branchMessages) {
        if (!message?.trigger) {
          continue;
        }
        experiments.push({
          forReachEvent: { sent: false, group: featureId },
          experimentSlug: experimentData.slug,
          branchSlug: branch.slug,
          ...message,
        });
      }
    }
  }

  return experiments;
},

_handleRemoteSettingsUndesiredEvent(event, providerId, dispatchCFRAction) {
  if (dispatchCFRAction) {
    dispatchCFRAction(
      ac.ASRouterUserEvent({
        action: "asrouter_undesired_event",
        event,
        message_id: "n/a",
        event_context: providerId,
      })
    );
  }
},

/**
 * _getMessageLoader - return the right loading function given the provider's type
 *
 * @param {obj} provider An AS Router provider
 * @returns {func} A loading function
 */
_getMessageLoader(provider) {
  switch (provider.type) {
    case "remote":
      return this._remoteLoader;
    case "remote-settings":
      return this._remoteSettingsLoader;
    case "remote-experiments":
      return this._experimentsAPILoader;
    case "local":
    default:
      return this._localLoader;
  }
},

/**
 * shouldProviderUpdate - Given the current time, should a provider update its messages?
 *
 * @param {any} provider An AS Router provider
 * @param {int} provider.updateCycleInMs The number of milliseconds we should wait between updates
 * @param {Date} provider.lastUpdated If the provider has been updated, the time the last update occurred
 * @param {Date} currentTime The time we should check against. (defaults to Date.now())
 * @returns {bool} Should an update happen?
 */
shouldProviderUpdate(provider, currentTime = Date.now()) {
  return (
    !(provider.lastUpdated >= 0) ||
    currentTime - provider.lastUpdated > provider.updateCycleInMs
  );
},

async _loadDataForProvider(provider, options) {
  const loader = this._getMessageLoader(provider);
  let messages = await loader(provider, options);
  // istanbul ignore if
  if (!messages) {
    messages = [];
    MessageLoaderUtils.reportError(
      new Error(
```

```
          `Tried to load messages for ${provider.id} but the result was not an Array.`
        )
      );
    }

    return { messages };
  },

  /**
   * loadMessagesForProvider - Load messages for a provider, given the provider's type.
   *
   * @param {obj} provider An AS Router provider
   * @param {string} provider.type An AS Router provider type (defaults to "local")
   * @param {obj} options.storage A storage object with get() and set() methods for caching.
   * @param {func} options.dispatchCFRAction dispatch an action the main AS Store
   * @returns {obj} Returns an object with .messages (an array of messages) and .lastUpdated (the time the messages were updated)
   */
  async loadMessagesForProvider(provider, options) {
    let { messages } = await this._loadDataForProvider(provider, options);
    // Filter out messages we temporarily want to exclude
    if (provider.exclude && provider.exclude.length) {
      messages = messages.filter(
        message => !provider.exclude.includes(message.id)
      );
    }
    const lastUpdated = Date.now();
    return {
      messages: messages
        .map(messageData => {
          const message = {
            weight: 100,
            ...messageData,
            groups: messageData.groups || [],
            provider: provider.id,
          };

          return message;
        })
        .filter(message => message.weight > 0),
      lastUpdated,
      errors: MessageLoaderUtils.errors,
    };
  },

  /**
   * cleanupCache - Removes cached data of removed providers.
   *
   * @param {Array} providers A list of activer AS Router providers
   */
  async cleanupCache(providers, storage) {
    const ids = providers.filter(p => p.type === "remote").map(p => p.id);
    const cache = await MessageLoaderUtils._remoteLoaderCache(storage);
    let dirty = false;
    for (let id in cache) {
      if (!ids.includes(id)) {
        delete cache[id];
        dirty = true;
      }
    }
    if (dirty) {
      await storage.set(MessageLoaderUtils.REMOTE_LOADER_CACHE_KEY, cache);
    }
  },

  /**
   * The locale to use for RemoteL10n.
   *
   * This may map the app's actual locale into something that RemoteL10n
   * supports.
   */
  get locale() {
    const localeMap = {
      "ja-JP-macos": "ja-JP-mac",

      // While it's not a valid locale, "und" is commonly observed on
      // Linux platforms. Per l10n team, it's reasonable to fallback to
      // "en-US", therefore, we should allow the fetch for it.
      und: "en-US",
    };

    const locale = Services.locale.appLocaleAsBCP47;
    return localeMap[locale] ?? locale;
  },
};
```

```
/**
 * @class _ASRouter - Keeps track of all messages, UI surfaces, and
 * handles blocking, rotation, etc. Inspecting ASRouter.state will
 * tell you what the current displayed message is in all UI surfaces.
 *
 * Note: This is written as a constructor rather than just a plain object
 * so that it can be more easily unit tested.
 */
class _ASRouter {
  constructor(localProviders = LOCAL_MESSAGE_PROVIDERS) {
    this.initialized = false;
    this.clearChildMessages = null;
    this.clearChildProviders = null;
    this.updateAdminState = null;
    this.sendTelemetry = null;
    this.dispatchCFRAction = null;
    this._storage = null;
    this._resetInitialization();
    this._state = {
      providers: [],
      messageBlockList: [],
      messageImpressions: {},
      screenImpressions: {},
      messages: [],
      groups: [],
      errors: [],
      localeInUse: Services.locale.appLocaleAsBCP47,
    };
    this._experimentChangedListeners = new Map();
    this._triggerHandler = this._triggerHandler.bind(this);
    this._localProviders = localProviders;
    this.blockMessageById = this.blockMessageById.bind(this);
    this.unblockMessageById = this.unblockMessageById.bind(this);
    this.handleMessageRequest = this.handleMessageRequest.bind(this);
    this.addImpression = this.addImpression.bind(this);
    this.addScreenImpression = this.addScreenImpression.bind(this);
    this._handleTargetingError = this._handleTargetingError.bind(this);
    this.onPrefChange = this.onPrefChange.bind(this);
    this._onLocaleChanged = this._onLocaleChanged.bind(this);
    this.isUnblockedMessage = this.isUnblockedMessage.bind(this);
    this.unblockAll = this.unblockAll.bind(this);
    this.forceWNPanel = this.forceWNPanel.bind(this);
    this._onExperimentEnrollmentsUpdated =
      this._onExperimentEnrollmentsUpdated.bind(this);
    this.forcePBWindow = this.forcePBWindow.bind(this);
    Services.telemetry.setEventRecordingEnabled(REACH_EVENT_CATEGORY, true);
  }

  async onPrefChange(prefName) {
    if (lazy.TARGETING_PREFERENCES.includes(prefName)) {
      let invalidMessages = [];
      // Notify all tabs of messages that have become invalid after pref change
      const context = this._getMessagesContext();
      const targetingContext = new lazy.TargetingContext(context);

      for (const msg of this.state.messages.filter(this.isUnblockedMessage)) {
        if (!msg.targeting) {
          continue;
        }
        const isMatch = await targetingContext.evalWithDefault(msg.targeting);
        if (!isMatch) {
          invalidMessages.push(msg.id);
        }
      }
      this.clearChildMessages(invalidMessages);
    } else {
      // Update message providers and fetch new messages on pref change
      this._loadLocalProviders();
      let invalidProviders = await this._updateMessageProviders();
      if (invalidProviders.length) {
        this.clearChildProviders(invalidProviders);
      }
      await this.loadMessagesFromAllProviders();
      // Any change in user prefs can disable or enable groups
      await this.setState(state => ({
        groups: state.groups.map(this._checkGroupEnabled),
      }));
    }
  }

  // Fetch and decode the message provider pref JSON, and update the message providers
  async _updateMessageProviders() {
    lazy.ASRouterPreferences.console.debug("entering updateMessageProviders");

    const previousProviders = this.state.providers;
```

```
    const providers = await Promise.all(
      [
        // If we have added a `preview` provider, hold onto it
        ...previousProviders.filter(p => p.id === "preview"),
        // The provider should be enabled and not have a user preference set to false
        ...lazy.ASRouterPreferences.providers.filter(
          p =>
            p.enabled &&
            lazy.ASRouterPreferences.getUserPreference(p.id) !== false
        ),
      ].map(async _provider => {
        // make a copy so we don't modify the source of the pref
        const provider = { ..._provider };

        if (provider.type === "local" && !provider.messages) {
          // Get the messages from the local message provider
          const localProvider = this._localProviders[provider.localProvider];
          provider.messages = [];
          if (localProvider) {
            provider.messages = await localProvider.getMessages();
          }
        }
        if (provider.type === "remote" && provider.url) {
          provider.url = provider.url.replace(
            /%STARTPAGE_VERSION%/g,
            STARTPAGE_VERSION
          );
          provider.url = Services.urlFormatter.formatURL(provider.url);
        }
        if (provider.id === "messaging-experiments") {
          // By default, the messaging-experiments provider lacks a featureIds
          // property, so fall back to the list of default features.
          if (!provider.featureIds) {
            provider.featureIds = MESSAGING_EXPERIMENTS_DEFAULT_FEATURES;
          }
        }
        // Reset provider update timestamp to force message refresh
        provider.lastUpdated = undefined;
        return provider;
      })
    );

    const providerIDs = providers.map(p => p.id);
    let invalidProviders = [];

    // Clear old messages for providers that are no longer enabled
    for (const prevProvider of previousProviders) {
      if (!providerIDs.includes(prevProvider.id)) {
        invalidProviders.push(prevProvider.id);
      }
    }

    return this.setState(prevState => ({
      providers,
      // Clear any messages from removed providers
      messages: [
        ...prevState.messages.filter(message =>
          providerIDs.includes(message.provider)
        ),
      ],
    })).then(() => invalidProviders);
  }

  get state() {
    return this._state;
  }

  set state(value) {
    throw new Error(
      "Do not modify this.state directy. Instead, call this.setState(newState)"
    );
  }

  /**
   * _resetInitialization - adds the following to the instance:
   *  .initialized {bool}            Has AS Router been initialized?
   *  .waitForInitialized {Promise}  A promise that resolves when initializion is complete
   *  ._finishInitializing {func}    A function that, when called, resolves the .waitForInitialized
   *                                 promise and sets .initialized to true.
   * @memberof _ASRouter
   */
  _resetInitialization() {
    this.initialized = false;
    this.initializing = false;
    this.waitForInitialized = new Promise(resolve => {
```

```
    this._finishInitializing = () => {
      this.initialized = true;
      this.initializing = false;
      resolve();
    };
  });
}

/**
 * Check all provided groups are enabled.
 * @param groups Set of groups to verify
 * @returns bool
 */
hasGroupsEnabled(groups = []) {
  return this.state.groups
    .filter(({ id }) => groups.includes(id))
    .every(({ enabled }) => enabled);
}

/**
 * Verify that the provider block the message through the `exclude` field
 * @param message Message to verify
 * @returns bool
 */
isExcludedByProvider(message) {
  // preview snippets are never excluded
  if (message.provider === "preview") {
    return false;
  }
  const provider = this.state.providers.find(p => p.id === message.provider);
  if (!provider) {
    return true;
  }
  if (provider.exclude) {
    return provider.exclude.includes(message.id);
  }
  return false;
}

/**
 * Takes a group and sets the correct `enabled` state based on message config
 * and user preferences
 *
 * @param {GroupConfig} group
 * @returns {GroupConfig}
 */
_checkGroupEnabled(group) {
  return {
    ...group,
    enabled:
      group.enabled &&
      // And if defined user preferences are true. If multiple prefs are
      // defined then at least one has to be enabled.
      (Array.isArray(group.userPreferences)
        ? group.userPreferences.some(pref =>
            lazy.ASRouterPreferences.getUserPreference(pref)
          )
        : true),
  };
}

/**
 * Fetch all message groups and update Router.state.groups.
 * There are two cases to consider:
 * 1. The provider needs to update as determined by the update cycle
 * 2. Some pref change occured which could invalidate one of the existing
 *    groups.
 */
async loadAllMessageGroups() {
  const provider = this.state.providers.find(
    p =>
      p.id === "message-groups" && MessageLoaderUtils.shouldProviderUpdate(p)
  );
  let remoteMessages = null;
  if (provider) {
    const { messages } = await MessageLoaderUtils._loadDataForProvider(
      provider,
      {
        storage: this._storage,
        dispatchCFRAction: this.dispatchCFRAction,
      }
    );
    remoteMessages = messages;
  }
  await this.setState(state => ({
```

```
    // If fetching remote messages fails we default to existing state.groups.
    groups: (remoteMessages || state.groups).map(this._checkGroupEnabled),
  }));
}

/**
 * loadMessagesFromAllProviders - Loads messages from all providers if they require updates.
 *                                Checks the .lastUpdated field on each provider to see if updates are needed
 * @param toUpdate  An optional list of providers to update. This overrides
 *                  the checks to determine which providers to update.
 * @memberof _ASRouter
 */
async loadMessagesFromAllProviders(toUpdate = undefined) {
  const needsUpdate = Array.isArray(toUpdate)
    ? toUpdate
    : this.state.providers.filter(provider =>
        MessageLoaderUtils.shouldProviderUpdate(provider)
      );
  lazy.ASRouterPreferences.console.debug(
    "entering loadMessagesFromAllProviders"
  );

  await this.loadAllMessageGroups();
  // Don't do extra work if we don't need any updates
  if (needsUpdate.length) {
    let newState = { messages: [], providers: [] };
    for (const provider of this.state.providers) {
      if (needsUpdate.includes(provider)) {
        const { messages, lastUpdated, errors } =
          await MessageLoaderUtils.loadMessagesForProvider(provider, {
            storage: this._storage,
            dispatchCFRAction: this.dispatchCFRAction,
          });
        newState.providers.push({ ...provider, lastUpdated, errors });
        newState.messages = [...newState.messages, ...messages];
      } else {
        // Skip updating this provider's messages if no update is required
        let messages = this.state.messages.filter(
          msg => msg.provider === provider.id
        );
        newState.providers.push(provider);
        newState.messages = [...newState.messages, ...messages];
      }
    }

    // Some messages have triggers that require us to initalise trigger listeners
    const unseenListeners = new Set(lazy.ASRouterTriggerListeners.keys());
    for (const { trigger } of newState.messages) {
      if (trigger && lazy.ASRouterTriggerListeners.has(trigger.id)) {
        lazy.ASRouterTriggerListeners.get(trigger.id).init(
          this._triggerHandler,
          trigger.params,
          trigger.patterns
        );
        unseenListeners.delete(trigger.id);
      }
    }
    // We don't need these listeners, but they may have previously been
    // initialised, so uninitialise them
    for (const triggerID of unseenListeners) {
      lazy.ASRouterTriggerListeners.get(triggerID).uninit();
    }

    // We don't want to cache preview endpoints, remove them after messages are fetched
    await this.setState(this._removePreviewEndpoint(newState));
    await this.cleanupImpressions();
  }

  await this._fireMessagesLoadedTrigger();

  return this.state;
}

async _fireMessagesLoadedTrigger() {
  const win = Services.wm.getMostRecentBrowserWindow() ?? null;
  const browser = win?.gBrowser?.selectedBrowser ?? null;
  // pass skipLoadingMessages to avoid infinite recursion. pass browser and
  // window into context so messages that may need a window or browser can
  // target accordingly.
  await this.sendTriggerMessage(
    {
      id: "messagesLoaded",
      browser,
      context: { browser, browserWindow: win },
    },
```

```
    true
  );
}

async _maybeUpdateL10nAttachment() {
  const { localeInUse } = this.state.localeInUse;
  const newLocale = Services.locale.appLocaleAsBCP47;
  if (newLocale !== localeInUse) {
    const providers = [...this.state.providers];
    let needsUpdate = false;
    providers.forEach(provider => {
      if (RS_PROVIDERS_WITH_L10N.includes(provider.id)) {
        // Force to refresh the messages as well as the attachment.
        provider.lastUpdated = undefined;
        needsUpdate = true;
      }
    });
    if (needsUpdate) {
      await this.setState({
        localeInUse: newLocale,
        providers,
      });
      await this.loadMessagesFromAllProviders();
    }
  }
  return this.state;
}

async _onLocaleChanged(subject, topic, data) {
  await this._maybeUpdateL10nAttachment();
}

observe(aSubject, aTopic, aPrefName) {
  switch (aPrefName) {
    case USE_REMOTE_L10N_PREF:
      CFRPageActions.reloadL10n();
      break;
  }
}

toWaitForInitFunc(func) {
  return (...args) => this.waitForInitialized.then(() => func(...args));
}

/**
 * init - Initializes the MessageRouter.
 *
 * @param {obj} parameters parameters to initialize ASRouter
 * @memberof _ASRouter
 */
async init({
  storage,
  sendTelemetry,
  clearChildMessages,
  clearChildProviders,
  updateAdminState,
  dispatchCFRAction,
}) {
  if (this.initializing || this.initialized) {
    return null;
  }
  this.initializing = true;
  this._storage = storage;
  this.ALLOWLIST_HOSTS = this._loadSnippetsAllowHosts();
  this.clearChildMessages = this.toWaitForInitFunc(clearChildMessages);
  this.clearChildProviders = this.toWaitForInitFunc(clearChildProviders);
  // NOTE: This is only necessary to sync devtools and snippets when devtools is active.
  this.updateAdminState = this.toWaitForInitFunc(updateAdminState);
  this.sendTelemetry = sendTelemetry;
  this.dispatchCFRAction = this.toWaitForInitFunc(dispatchCFRAction);

  lazy.ASRouterPreferences.init();
  lazy.ASRouterPreferences.addListener(this.onPrefChange);
  lazy.ToolbarBadgeHub.init(this.waitForInitialized, {
    handleMessageRequest: this.handleMessageRequest,
    addImpression: this.addImpression,
    blockMessageById: this.blockMessageById,
    unblockMessageById: this.unblockMessageById,
    sendTelemetry: this.sendTelemetry,
  });
  lazy.ToolbarPanelHub.init(this.waitForInitialized, {
    getMessages: this.handleMessageRequest,
    sendTelemetry: this.sendTelemetry,
  });
  lazy.MomentsPageHub.init(this.waitForInitialized, {
```

```
    handleMessageRequest: this.handleMessageRequest,
    addImpression: this.addImpression,
    blockMessageById: this.blockMessageById,
    sendTelemetry: this.sendTelemetry,
  });

  this._loadLocalProviders();

  const messageBlockList =
    (await this._storage.get("messageBlockList")) || [];
  const messageImpressions =
    (await this._storage.get("messageImpressions")) || {};
  const groupImpressions =
    (await this._storage.get("groupImpressions")) || {};
  const screenImpressions =
    (await this._storage.get("screenImpressions")) || {};
  const previousSessionEnd =
    (await this._storage.get("previousSessionEnd")) || 0;

  await this.setState({
    messageBlockList,
    groupImpressions,
    messageImpressions,
    screenImpressions,
    previousSessionEnd,
    ...(lazy.ASRouterPreferences.specialConditions || {}),
    initialized: false,
  });
  await this._updateMessageProviders();
  await this.loadMessagesFromAllProviders();
  await MessageLoaderUtils.cleanupCache(this.state.providers, storage);

  lazy.SpecialMessageActions.blockMessageById = this.blockMessageById;
  Services.obs.addObserver(this._onLocaleChanged, TOPIC_INTL_LOCALE_CHANGED);
  Services.obs.addObserver(
    this._onExperimentEnrollmentsUpdated,
    TOPIC_EXPERIMENT_ENROLLMENT_CHANGED
  );
  Services.prefs.addObserver(USE_REMOTE_L10N_PREF, this);
  // sets .initialized to true and resolves .waitForInitialized promise
  this._finishInitializing();
  return this.state;
}

uninit() {
  this._storage.set("previousSessionEnd", Date.now());

  this.clearChildMessages = null;
  this.clearChildProviders = null;
  this.updateAdminState = null;
  this.sendTelemetry = null;
  this.dispatchCFRAction = null;

  lazy.ASRouterPreferences.removeListener(this.onPrefChange);
  lazy.ASRouterPreferences.uninit();
  lazy.ToolbarPanelHub.uninit();
  lazy.ToolbarBadgeHub.uninit();
  lazy.MomentsPageHub.uninit();

  // Uninitialise all trigger listeners
  for (const listener of lazy.ASRouterTriggerListeners.values()) {
    listener.uninit();
  }
  Services.obs.removeObserver(
    this._onLocaleChanged,
    TOPIC_INTL_LOCALE_CHANGED
  );
  Services.obs.removeObserver(
    this._onExperimentEnrollmentsUpdated,
    TOPIC_EXPERIMENT_ENROLLMENT_CHANGED
  );
  Services.prefs.removeObserver(USE_REMOTE_L10N_PREF, this);
  // If we added any CFR recommendations, they need to be removed
  CFRPageActions.clearRecommendations();
  this._resetInitialization();
}

setState(callbackOrObj) {
  lazy.ASRouterPreferences.console.debug(
    "in setState, callbackOrObj = ",
    callbackOrObj
  );
  lazy.ASRouterPreferences.console.trace();
  const newState =
    typeof callbackOrObj === "function"
```

```
      ? callbackOrObj(this.state)
      : callbackOrObj;
  this._state = {
    ...this.state,
    ...newState,
  };
  if (lazy.ASRouterPreferences.devtoolsEnabled) {
    return this.updateTargetingParameters().then(state => {
      this.updateAdminState(state);
      return state;
    });
  }
  return Promise.resolve(this.state);
}

updateTargetingParameters() {
  return this.getTargetingParameters(
    lazy.ASRouterTargeting.Environment,
    this._getMessagesContext()
  ).then(targetingParameters => ({
    ...this.state,
    providerPrefs: lazy.ASRouterPreferences.providers,
    userPrefs: lazy.ASRouterPreferences.getAllUserPreferences(),
    targetingParameters,
    errors: this.errors,
  }));
}

getMessageById(id) {
  return this.state.messages.find(message => message.id === id);
}

_loadLocalProviders() {
  // If we're in ASR debug mode add the local test providers
  if (lazy.ASRouterPreferences.devtoolsEnabled) {
    this._localProviders = {
      ...this._localProviders,
      SnippetsTestMessageProvider: lazy.SnippetsTestMessageProvider,
      PanelTestProvider: lazy.PanelTestProvider,
    };
  }
}

/**
 * Used by ASRouter Admin returns all ASRouterTargeting.Environment
 * and ASRouter._getMessagesContext parameters and values
 */
async getTargetingParameters(environment, localContext) {
  // Resolve objects that may contain promises.
  async function resolve(object) {
    if (typeof object === "object" && object !== null) {
      if (Array.isArray(object)) {
        return Promise.all(object.map(async item => resolve(await item)));
      }

      if (object instanceof Date) {
        return object;
      }

      const target = {};
      const promises = Object.entries(object).map(async ([key, value]) => {
        try {
          let resolvedValue = await resolve(await value);
          return [key, resolvedValue];
        } catch (error) {
          lazy.ASRouterPreferences.console.debug(
            `getTargetingParameters: Error resolving ${key}: `,
            error
          );
          throw error;
        }
      });
      for (const { status, value } of await Promise.allSettled(promises)) {
        if (status === "fulfilled") {
          const [key, resolvedValue] = value;
          target[key] = resolvedValue;
        }
      }
      return target;
    }

    return object;
  }

  const targetingParameters = {
```

```
    ...(await resolve(environment)),
    ...(await resolve(localContext)),
  };

  return targetingParameters;
}

_handleTargetingError(error, message) {
  console.error(error);
  this.dispatchCFRAction(
    ac.ASRouterUserEvent({
      message_id: message.id,
      action: "asrouter_undesired_event",
      event: "TARGETING_EXPRESSION_ERROR",
      event_context: {},
    })
  );
}

// Return an object containing targeting parameters used to select messages
_getMessagesContext() {
  const { messageImpressions, previousSessionEnd, screenImpressions } =
    this.state;

  return {
    get messageImpressions() {
      return messageImpressions;
    },
    get previousSessionEnd() {
      return previousSessionEnd;
    },
    get screenImpressions() {
      return screenImpressions;
    },
  };
}

async evaluateExpression({ expression, context }) {
  const targetingContext = new lazy.TargetingContext(context);
  let evaluationStatus;
  try {
    evaluationStatus = {
      result: await targetingContext.evalWithDefault(expression),
      success: true,
    };
  } catch (e) {
    evaluationStatus = { result: e.message, success: false };
  }
  return Promise.resolve({ evaluationStatus });
}

unblockAll() {
  return this.setState({ messageBlockList: [] });
}

isUnblockedMessage(message) {
  const { state } = this;
  return (
    !state.messageBlockList.includes(message.id) &&
    (!message.campaign ||
      !state.messageBlockList.includes(message.campaign)) &&
    this.hasGroupsEnabled(message.groups) &&
    !this.isExcludedByProvider(message)
  );
}

// Work out if a message can be shown based on its and its provider's frequency caps.
isBelowFrequencyCaps(message) {
  const { messageImpressions, groupImpressions } = this.state;
  const impressionsForMessage = messageImpressions[message.id];

  const _belowItemFrequencyCap = this._isBelowItemFrequencyCap(
    message,
    impressionsForMessage,
    MAX_MESSAGE_LIFETIME_CAP
  );
  if (!_belowItemFrequencyCap) {
    lazy.ASRouterPreferences.console.debug(
      `isBelowFrequencyCaps: capped by item: `,
      message,
      "impressions =",
      impressionsForMessage
    );
  }
```

```
  const _belowGroupFrequencyCaps = message.groups.every(messageGroup => {
    const belowThisGroupCap = this._isBelowItemFrequencyCap(
      this.state.groups.find(({ id }) => id === messageGroup),
      groupImpressions[messageGroup]
    );

    if (!belowThisGroupCap) {
      lazy.ASRouterPreferences.console.debug(
        `isBelowFrequencyCaps: ${message.id} capped by group ${messageGroup}`
      );
    } else {
      lazy.ASRouterPreferences.console.debug(
        `isBelowFrequencyCaps: ${message.id} allowed by group ${messageGroup}, groupImpressions = `,
        groupImpressions
      );
    }

    return belowThisGroupCap;
  });

  return _belowItemFrequencyCap && _belowGroupFrequencyCaps;
}

// Helper for isBelowFrecencyCaps - work out if the frequency cap for the given
//                                  item has been exceeded or not
_isBelowItemFrequencyCap(item, impressions, maxLifetimeCap = Infinity) {
  if (item && item.frequency && impressions && impressions.length) {
    if (
      item.frequency.lifetime &&
      impressions.length >= Math.min(item.frequency.lifetime, maxLifetimeCap)
    ) {
      lazy.ASRouterPreferences.console.debug(
        `${item.id} capped by lifetime (${item.frequency.lifetime})`
      );

      return false;
    }
    if (item.frequency.custom) {
      const now = Date.now();
      for (const setting of item.frequency.custom) {
        let { period } = setting;
        const impressionsInPeriod = impressions.filter(t => now - t < period);
        if (impressionsInPeriod.length >= setting.cap) {
          lazy.ASRouterPreferences.console.debug(
            `${item.id} capped by impressions (${impressionsInPeriod.length}) in period (${period}) >= ${setting.cap}`
          );
          return false;
        }
      }
    }
  }
  return true;
}

async _extraTemplateStrings(originalMessage) {
  let extraTemplateStrings;
  let localProvider = this._findProvider(originalMessage.provider);
  if (localProvider && localProvider.getExtraAttributes) {
    extraTemplateStrings = await localProvider.getExtraAttributes();
  }

  return extraTemplateStrings;
}

_findProvider(providerID) {
  return this._localProviders[
    this.state.providers.find(i => i.id === providerID).localProvider
  ];
}

routeCFRMessage(message, browser, trigger, force = false) {
  if (!message) {
    return { message: {} };
  }

  switch (message.template) {
    case "whatsnew_panel_message":
      if (force) {
        lazy.ToolbarPanelHub.forceShowMessage(browser, message);
      }
      break;
    case "cfr_doorhanger":
    case "milestone_message":
      if (force) {
        CFRPageActions.forceRecommendation(
```

```
          browser,
          message,
          this.dispatchCFRAction
        );
      } else {
        CFRPageActions.addRecommendation(
          browser,
          trigger.param && trigger.param.host,
          message,
          this.dispatchCFRAction
        );
      }
      break;
    case "cfr_urlbar_chiclet":
      if (force) {
        CFRPageActions.forceRecommendation(
          browser,
          message,
          this.dispatchCFRAction
        );
      } else {
        CFRPageActions.addRecommendation(
          browser,
          null,
          message,
          this.dispatchCFRAction
        );
      }
      break;
    case "toolbar_badge":
      lazy.ToolbarBadgeHub.registerBadgeNotificationListener(message, {
        force,
      });
      break;
    case "update_action":
      lazy.MomentsPageHub.executeAction(message);
      break;
    case "infobar":
      lazy.InfoBar.showInfoBarMessage(
        browser,
        message,
        this.dispatchCFRAction
      );
      break;
    case "spotlight":
      lazy.Spotlight.showSpotlightDialog(
        browser,
        message,
        this.dispatchCFRAction
      );
      break;
    case "toast_notification":
      lazy.ToastNotification.showToastNotification(
        message,
        this.dispatchCFRAction
      );
      break;
  }

  return { message };
}

addScreenImpression(screen) {
  lazy.ASRouterPreferences.console.debug(
    `entering addScreenImpression for ${screen.id}`
  );

  const time = Date.now();

  let screenImpressions = { ...this.state.screenImpressions };
  screenImpressions[screen.id] = time;

  this.setState({ screenImpressions });
  lazy.ASRouterPreferences.console.debug(
    screen.id,
    `screen impression added, screenImpressions[screen.id]: `,
    screenImpressions[screen.id]
  );
  this._storage.set("screenImpressions", screenImpressions);
}

addImpression(message) {
  lazy.ASRouterPreferences.console.debug(
    `entering addImpression for ${message.id}`
  );
```

```
  const groupsWithFrequency = this.state.groups.filter(
    ({ frequency, id }) => frequency && message.groups.includes(id)
  );
  // We only need to store impressions for messages that have frequency, or
  // that have providers that have frequency
  if (message.frequency || groupsWithFrequency.length) {
    const time = Date.now();
    return this.setState(state => {
      const messageImpressions = this._addImpressionForItem(
        state.messageImpressions,
        message,
        "messageImpressions",
        time
      );
      // Initialize this with state.groupImpressions, and then assign the
      // newly-updated copy to it during each iteration so that
      // all the changes get captured and either returned or passed into the
      // _addImpressionsForItem call on the next iteration.
      let { groupImpressions } = state;
      for (const group of groupsWithFrequency) {
        groupImpressions = this._addImpressionForItem(
          groupImpressions,
          group,
          "groupImpressions",
          time
        );
      }

      return { messageImpressions, groupImpressions };
    });
  }
  return Promise.resolve();
}

// Helper for addImpression - calculate the updated impressions object for the given
//                           item, then store it and return it
_addImpressionForItem(currentImpressions, item, impressionsString, time) {
  // The destructuring here is to avoid mutating passed parameters
  // (see https://redux.js.org/recipes/structuring-reducers/prerequisite-concepts#immutable-data-management)
  const impressions = { ...currentImpressions };
  if (item.frequency) {
    impressions[item.id] = [...(impressions[item.id] ?? []), time];

    lazy.ASRouterPreferences.console.debug(
      item.id,
      "impression added, impressions[item.id]: ",
      impressions[item.id]
    );

    this._storage.set(impressionsString, impressions);
  }
  return impressions;
}

/**
 * getLongestPeriod
 *
 * @param {obj} item Either an ASRouter message or an ASRouter provider
 * @returns {int|null} if the item has custom frequency caps, the longest period found in the list of caps.
                       if the item has no custom frequency caps, null
 * @memberof _ASRouter
 */
getLongestPeriod(item) {
  if (!item.frequency || !item.frequency.custom) {
    return null;
  }
  return item.frequency.custom.sort((a, b) => b.period - a.period)[0].period;
}

/**
 * cleanupImpressions - this function cleans up obsolete impressions whenever
 * messages are refreshed or fetched. It will likely need to be more sophisticated in the future,
 * but the current behaviour for when both message impressions and provider impressions are
 * cleared is as follows (where `item` is either `message` or `provider`):
 *
 * 1. If the item id for a list of item impressions no longer exists in the ASRouter state, it
 *    will be cleared.
 * 2. If the item has time-bound frequency caps but no lifetime cap, any item impressions older
 *    than the longest time period will be cleared.
 */
cleanupImpressions() {
  return this.setState(state => {
    const messageImpressions = this._cleanupImpressionsForItems(
      state,
```

```
      state.messages,
      "messageImpressions"
    );
    const groupImpressions = this._cleanupImpressionsForItems(
      state,
      state.groups,
      "groupImpressions"
    );
    return { messageImpressions, groupImpressions };
  });
}

/** _cleanupImpressionsForItems - Helper for cleanupImpressions - calculate the updated
/*                               impressions object for the given items, then store it and return it
 *
 * @param {obj} state Reference to ASRouter internal state
 * @param {array} items Can be messages, providers or groups that we count impressions for
 * @param {string} impressionsString Key name for entry in state where impressions are stored
 */
_cleanupImpressionsForItems(state, items, impressionsString) {
  const impressions = { ...state[impressionsString] };
  let needsUpdate = false;
  Object.keys(impressions).forEach(id => {
    const [item] = items.filter(x => x.id === id);
    // Don't keep impressions for items that no longer exist
    if (!item || !item.frequency || !Array.isArray(impressions[id])) {
      lazy.ASRouterPreferences.console.debug(
        "_cleanupImpressionsForItem: removing impressions for deleted or changed item: ",
        item
      );
      lazy.ASRouterPreferences.console.trace();
      delete impressions[id];
      needsUpdate = true;
      return;
    }
    if (!impressions[id].length) {
      return;
    }
    // If we don't want to store impressions older than the longest period
    if (item.frequency.custom && !item.frequency.lifetime) {
      lazy.ASRouterPreferences.console.debug(
        "_cleanupImpressionsForItem: removing impressions older than longest period for item: ",
        item
      );
      const now = Date.now();
      impressions[id] = impressions[id].filter(
        t => now - t < this.getLongestPeriod(item)
      );
      needsUpdate = true;
    }
  });
  if (needsUpdate) {
    this._storage.set(impressionsString, impressions);
  }
  return impressions;
}

handleMessageRequest({
  messages: candidates,
  triggerId,
  triggerParam,
  triggerContext,
  template,
  provider,
  ordered = false,
  returnAll = false,
}) {
  let shouldCache;
  lazy.ASRouterPreferences.console.debug(
    "in handleMessageRequest, arguments = ",
    Array.from(arguments) // eslint-disable-line prefer-rest-params
  );
  lazy.ASRouterPreferences.console.trace();
  const messages =
    candidates ||
    this.state.messages.filter(m => {
      if (provider && m.provider !== provider) {
        lazy.ASRouterPreferences.console.debug(m.id, " filtered by provider");
        return false;
      }
      if (template && m.template !== template) {
        lazy.ASRouterPreferences.console.debug(m.id, " filtered by template");
        return false;
      }
      if (triggerId && !m.trigger) {
```

```
        lazy.ASRouterPreferences.console.debug(m.id, " filtered by trigger");
        return false;
      }
      if (triggerId && m.trigger.id !== triggerId) {
        lazy.ASRouterPreferences.console.debug(
          m.id,
          " filtered by triggerId"
        );
        return false;
      }
      if (!this.isUnblockedMessage(m)) {
        lazy.ASRouterPreferences.console.debug(
          m.id,
          " filtered because blocked"
        );
        return false;
      }
      if (!this.isBelowFrequencyCaps(m)) {
        lazy.ASRouterPreferences.console.debug(
          m.id,
          " filtered because capped"
        );
        return false;
      }

      if (shouldCache !== false) {
        shouldCache = JEXL_PROVIDER_CACHE.has(m.provider);
      }

      return true;
    });

  if (!messages.length) {
    return returnAll ? messages : null;
  }

  const context = this._getMessagesContext();

  // Find a message that matches the targeting context as well as the trigger context (if one is provided)
  // If no trigger is provided, we should find a message WITHOUT a trigger property defined.
  return lazy.ASRouterTargeting.findMatchingMessage({
    messages,
    trigger: triggerId && {
      id: triggerId,
      param: triggerParam,
      context: triggerContext,
    },
    context,
    onError: this._handleTargetingError,
    ordered,
    shouldCache,
    returnAll,
  });
}

setMessageById({ id, ...data }, force, browser) {
  return this.routeCFRMessage(this.getMessageById(id), browser, data, force);
}

blockMessageById(idOrIds) {
  lazy.ASRouterPreferences.console.debug(
    "blockMessageById called, idOrIds = ",
    idOrIds
  );
  lazy.ASRouterPreferences.console.trace();

  const idsToBlock = Array.isArray(idOrIds) ? idOrIds : [idOrIds];

  return this.setState(state => {
    const messageBlockList = [...state.messageBlockList];
    const messageImpressions = { ...state.messageImpressions };

    idsToBlock.forEach(id => {
      const message = state.messages.find(m => m.id === id);
      const idToBlock = message && message.campaign ? message.campaign : id;
      if (!messageBlockList.includes(idToBlock)) {
        messageBlockList.push(idToBlock);
      }

      // When a message is blocked, its impressions should be cleared as well
      delete messageImpressions[id];
    });

    this._storage.set("messageBlockList", messageBlockList);
    this._storage.set("messageImpressions", messageImpressions);
```

```
    return { messageBlockList, messageImpressions };
  });
}

unblockMessageById(idOrIds) {
  const idsToUnblock = Array.isArray(idOrIds) ? idOrIds : [idOrIds];

  return this.setState(state => {
    const messageBlockList = [...state.messageBlockList];
    idsToUnblock
      .map(id => state.messages.find(m => m.id === id))
      // Remove all `id`s (or `campaign`s for snippets) from the message
      // block list
      .forEach(message => {
        const idToUnblock =
          message && message.campaign ? message.campaign : message.id;
        messageBlockList.splice(messageBlockList.indexOf(idToUnblock), 1);
      });

    this._storage.set("messageBlockList", messageBlockList);
    return { messageBlockList };
  });
}

resetGroupsState() {
  const newGroupImpressions = {};
  for (let { id } of this.state.groups) {
    newGroupImpressions[id] = [];
  }
  // Update storage
  this._storage.set("groupImpressions", newGroupImpressions);
  return this.setState(({ groups }) => ({
    groupImpressions: newGroupImpressions,
  }));
}

resetMessageState() {
  const newMessageImpressions = {};
  for (let { id } of this.state.messages) {
    newMessageImpressions[id] = [];
  }
  // Update storage
  this._storage.set("messageImpressions", newMessageImpressions);
  return this.setState(() => ({
    messageImpressions: newMessageImpressions,
  }));
}

_validPreviewEndpoint(url) {
  try {
    const endpoint = new URL(url);
    if (!this.ALLOWLIST_HOSTS[endpoint.host]) {
      console.error(
        `The preview URL host ${endpoint.host} is not in the list of allowed hosts.`
      );
    }
    if (endpoint.protocol !== "https:") {
      console.error("The URL protocol is not https.");
    }
    return (
      endpoint.protocol === "https:" && this.ALLOWLIST_HOSTS[endpoint.host]
    );
  } catch (e) {
    return false;
  }
}

_loadSnippetsAllowHosts() {
  let additionalHosts = [];
  const allowPrefValue = Services.prefs.getStringPref(
    SNIPPETS_ENDPOINT_ALLOWLIST,
    ""
  );
  try {
    additionalHosts = JSON.parse(allowPrefValue);
  } catch (e) {
    if (allowPrefValue) {
      console.error(
        `Pref ${SNIPPETS_ENDPOINT_ALLOWLIST} value is not valid JSON`
      );
    }
  }

  if (!additionalHosts.length) {
    return DEFAULT_ALLOWLIST_HOSTS;
```

```
  }

  // If there are additional hosts we want to allow, add them as
  // `preview` so that the updateCycle is 0
  return additionalHosts.reduce(
    (allow_hosts, host) => {
      allow_hosts[host] = "preview";
      Services.console.logStringMessage(
        `Adding ${host} to list of allowed hosts.`
      );
      return allow_hosts;
    },
    { ...DEFAULT_ALLOWLIST_HOSTS }
  );
}

// To be passed to ASRouterTriggerListeners
_triggerHandler(browser, trigger) {
  // Disable ASRouterTriggerListeners in kiosk mode.
  if (lazy.BrowserHandler.kiosk) {
    return Promise.resolve();
  }
  return this.sendTriggerMessage({ ...trigger, browser });
}

_removePreviewEndpoint(state) {
  state.providers = state.providers.filter(p => p.id !== "preview");
  return state;
}

addPreviewEndpoint(url, browser) {
  const providers = [...this.state.providers];
  if (
    this._validPreviewEndpoint(url) &&
    !providers.find(p => p.url === url)
  ) {
    // When you view a preview snippet we want to hide all real content -
    // sending EnterSnippetsPreviewMode puts this browser tab in that state.
    browser.sendMessageToActor("EnterSnippetsPreviewMode", {}, "ASRouter");
    providers.push({
      id: "preview",
      type: "remote",
      enabled: true,
      url,
      updateCycleInMs: 0,
    });
    return this.setState({ providers });
  }
  return Promise.resolve();
}

/**
 * forceAttribution - this function should only be called from within about:newtab#asrouter.
 * It forces the browser attribution to be set to something specified in asrouter admin
 * tools, and reloads the providers in order to get messages that are dependant on this
 * attribution data (see Return to AMO flow in bug 1475354 for example). Note - OSX and Windows only
 * @param {data} Object an object containing the attribtion data that came from asrouter admin page
 */
async forceAttribution(data) {
  // Extract the parameters from data that will make up the referrer url
  const attributionData = AttributionCode.allowedCodeKeys
    .map(key => `${key}=${encodeURIComponent(data[key] || "")}`)
    .join("&");
  if (AppConstants.platform === "win") {
    // The whole attribution data is encoded (again) for windows
    await AttributionCode.writeAttributionFile(
      encodeURIComponent(attributionData)
    );
  } else if (AppConstants.platform === "macosx") {
    let appPath = lazy.MacAttribution.applicationPath;
    let attributionSvc = Cc["@mozilla.org/mac-attribution;1"].getService(
      Ci.nsIMacAttributionService
    );

    // The attribution data is treated as a url query for mac
    let referrer = `https://www.mozilla.org/anything/?${attributionData}`;

    // This sets the Attribution to be the referrer
    attributionSvc.setReferrerUrl(appPath, referrer, true);

    // Delete attribution data file
    await AttributionCode.deleteFileAsync();
  }

  // Clear cache call is only possible in a testing environment
```

```
    Services.env.set("XPCSHELL_TEST_PROFILE_DIR", "testing");

    // Clear and refresh Attribution, and then fetch the messages again to update
    AttributionCode._clearCache();
    await AttributionCode.getAttrDataAsync();
    await this._updateMessageProviders();
    return this.loadMessagesFromAllProviders();
  }

  async sendPBNewTabMessage({ tabId, hideDefault }) {
    let message = null;
    const PromoInfo = {
      FOCUS: { enabledPref: "browser.promo.focus.enabled" },
      VPN: { enabledPref: "browser.vpn_promo.enabled" },
      PIN: { enabledPref: "browser.promo.pin.enabled" },
      COOKIE_BANNERS: { enabledPref: "browser.promo.cookiebanners.enabled" },
    };
    await this.loadMessagesFromAllProviders();

    // If message has hideDefault property set to true
    // remove from state all pb_newtab messages with type default
    if (hideDefault) {
      await this.setState(state => ({
        messages: state.messages.filter(
          m => !(m.template === "pb_newtab" && m.type === "default")
        ),
      }));
    }

    // Remove from state pb_newtab messages with PromoType disabled
    await this.setState(state => ({
      messages: state.messages.filter(
        m =>
          !(
            m.template === "pb_newtab" &&
            !Services.prefs.getBoolPref(
              PromoInfo[m.content?.promoType]?.enabledPref,
              true
            )
          )
      ),
    }));

    const telemetryObject = { tabId };
    TelemetryStopwatch.start("MS_MESSAGE_REQUEST_TIME_MS", telemetryObject);
    message = await this.handleMessageRequest({
      template: "pb_newtab",
    });
    TelemetryStopwatch.finish("MS_MESSAGE_REQUEST_TIME_MS", telemetryObject);

    // Format urls if any are defined
    ["infoLinkUrl"].forEach(key => {
      if (message?.content?.[key]) {
        message.content[key] = Services.urlFormatter.formatURL(
          message.content[key]
        );
      }
    });

    return { message };
  }

  async sendNewTabMessage({ endpoint, tabId, browser }) {
    let message;

    // Load preview endpoint for snippets if one is sent
    if (endpoint) {
      await this.addPreviewEndpoint(endpoint.url, browser);
    }

    // Load all messages
    await this.loadMessagesFromAllProviders();

    if (endpoint) {
      message = await this.handleMessageRequest({ provider: "preview" });

      // We don't want to cache preview messages, remove them after we selected the message to show
      if (message) {
        await this.setState(state => ({
          messages: state.messages.filter(m => m.id !== message.id),
        }));
      }
    } else {
      const telemetryObject = { tabId };
      TelemetryStopwatch.start("MS_MESSAGE_REQUEST_TIME_MS", telemetryObject);
```

```
    message = await this.handleMessageRequest({ provider: "snippets" });
    TelemetryStopwatch.finish("MS_MESSAGE_REQUEST_TIME_MS", telemetryObject);
  }

  return this.routeCFRMessage(message, browser, undefined, false);
}

_recordReachEvent(message) {
  const messageGroup = message.forReachEvent.group;
  // Events telemetry only accepts understores for the event `object`
  const underscored = messageGroup.split("-").join("_");
  const extra = { branches: message.branchSlug };
  Services.telemetry.recordEvent(
    REACH_EVENT_CATEGORY,
    REACH_EVENT_METHOD,
    underscored,
    message.experimentSlug,
    extra
  );
}

/**
 * Fire a trigger, look for a matching message, and route it to the
 * appropriate message handler/messaging surface.
 * @param {object} trigger
 * @param {string} trigger.id the name of the trigger, e.g. "openURL"
 * @param {object} [trigger.param] an object with host, url, type, etc. keys
 *   whose values are used to match against the message's trigger params
 * @param {object} [trigger.context] an object with data about the source of
 *   the trigger, matched against the message's targeting expression
 * @param {MozBrowser} trigger.browser the browser to route messages to
 * @param {number} [trigger.tabId] identifier used only for exposure testing
 * @param {boolean} [skipLoadingMessages=false] pass true to skip looking for
 *   new messages. use when calling from loadMessagesFromAllProviders to avoid
 *   recursion. we call this from loadMessagesFromAllProviders in order to
 *   fire the messagesLoaded trigger.
 * @returns {Promise<object>}
 * @resolves {message} an object with the routed message
 */
async sendTriggerMessage(
  { tabId, browser, ...trigger },
  skipLoadingMessages = false
) {
  if (!skipLoadingMessages) {
    await this.loadMessagesFromAllProviders();
  }
  const telemetryObject = { tabId };
  TelemetryStopwatch.start("MS_MESSAGE_REQUEST_TIME_MS", telemetryObject);
  // Return all the messages so that it can record the Reach event
  const messages =
    (await this.handleMessageRequest({
      triggerId: trigger.id,
      triggerParam: trigger.param,
      triggerContext: trigger.context,
      returnAll: true,
    })) || [];
  TelemetryStopwatch.finish("MS_MESSAGE_REQUEST_TIME_MS", telemetryObject);

  // Record the Reach event for all the messages with `forReachEvent`,
  // only send the first message without forReachEvent to the target
  const nonReachMessages = [];
  for (const message of messages) {
    if (message.forReachEvent) {
      if (!message.forReachEvent.sent) {
        this._recordReachEvent(message);
        message.forReachEvent.sent = true;
      }
    } else {
      nonReachMessages.push(message);
    }
  }

  if (nonReachMessages.length) {
    let featureId = nonReachMessages[0]._nimbusFeature;
    if (featureId) {
      lazy.NimbusFeatures[featureId].recordExposureEvent({ once: true });
    }
  }

  return this.routeCFRMessage(
    nonReachMessages[0] || null,
    browser,
    trigger,
    false
  );
```

```
  }

  async forceWNPanel(browser) {
    let win = browser.ownerGlobal;
    await lazy.ToolbarPanelHub.enableToolbarButton();

    win.PanelUI.showSubView(
      "PanelUI-whatsNew",
      win.document.getElementById("whats-new-menu-button")
    );

    let panel = win.document.getElementById("customizationui-widget-panel");
    // Set the attribute to keep the panel open
    panel.setAttribute("noautohide", true);
  }

  async closeWNPanel(browser) {
    let win = browser.ownerGlobal;
    let panel = win.document.getElementById("customizationui-widget-panel");
    // Set the attribute to allow the panel to close
    panel.setAttribute("noautohide", false);
    // Removing the button is enough to close the panel.
    await lazy.ToolbarPanelHub._hideToolbarButton(win);
  }

  async _onExperimentEnrollmentsUpdated() {
    const experimentProvider = this.state.providers.find(
      p => p.id === "messaging-experiments"
    );
    if (!experimentProvider?.enabled) {
      return;
    }
    await this.loadMessagesFromAllProviders([experimentProvider]);
  }

  async forcePBWindow(browser, msg) {
    const privateBrowserOpener = await new Promise(
      (
        resolveOnContentBrowserCreated // wrap this in a promise to give back the right browser
      ) =>
        browser.ownerGlobal.openTrustedLinkIn(
          "about:privatebrowsing?debug",
          "window",
          {
            private: true,
            triggeringPrincipal:
              Services.scriptSecurityManager.getSystemPrincipal({}),
            csp: null,
            resolveOnContentBrowserCreated,
            opener: "devtools",
          }
        )
    );

    lazy.setTimeout(() => {
      // setTimeout is necessary to make sure the private browsing window has a chance to open before the message is sent
      privateBrowserOpener.browsingContext.currentWindowGlobal
        .getActor("AboutPrivateBrowsing")
        .sendAsyncMessage("ShowDevToolsMessage", msg);
    }, 100);

    return privateBrowserOpener;
  }
}

/**
 * ASRouter - singleton instance of _ASRouter that controls all messages
 * in the new tab page.
 */
const ASRouter = new _ASRouter();

const EXPORTED_SYMBOLS = ["_ASRouter", "ASRouter", "MessageLoaderUtils"];
```