```
/* 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/. */

import { XPCOMUtils } from "resource://gre/modules/XPCOMUtils.sys.mjs";
import { LogManager } from "resource://normandy/lib/LogManager.sys.mjs";
import { PromiseUtils } from "resource://gre/modules/PromiseUtils.sys.mjs";

const lazy = {};

XPCOMUtils.defineLazyServiceGetter(
  lazy,
  "timerManager",
  "@mozilla.org/updates/timer-manager;1",
  "nsIUpdateTimerManager"
);

ChromeUtils.defineESModuleGetters(lazy, {
  ActionsManager: "resource://normandy/lib/ActionsManager.sys.mjs",
  BaseAction: "resource://normandy/actions/BaseAction.sys.mjs",
  CleanupManager: "resource://normandy/lib/CleanupManager.sys.mjs",
  ClientEnvironment: "resource://normandy/lib/ClientEnvironment.sys.mjs",
  FilterExpressions:
    "resource://gre/modules/components-utils/FilterExpressions.sys.mjs",
  LegacyHeartbeat: "resource://normandy/lib/LegacyHeartbeat.sys.mjs",
  Normandy: "resource://normandy/Normandy.sys.mjs",
  NormandyApi: "resource://normandy/lib/NormandyApi.sys.mjs",
  RemoteSettings: "resource://services-settings/remote-settings.sys.mjs",
  RemoteSettingsClient:
    "resource://services-settings/RemoteSettingsClient.sys.mjs",
  Storage: "resource://normandy/lib/Storage.sys.mjs",
  TargetingContext: "resource://messaging-system/targeting/Targeting.sys.mjs",
  Uptake: "resource://normandy/lib/Uptake.sys.mjs",
  clearTimeout: "resource://gre/modules/Timer.sys.mjs",
  setTimeout: "resource://gre/modules/Timer.sys.mjs",
});

const log = LogManager.getLogger("recipe-runner");
const TIMER_NAME = "recipe-client-addon-run";
const REMOTE_SETTINGS_COLLECTION = "normandy-recipes-capabilities";
const PREF_CHANGED_TOPIC = "nsPref:changed";

const RUN_INTERVAL_PREF = "app.normandy.run_interval_seconds";
const FIRST_RUN_PREF = "app.normandy.first_run";
const SHIELD_ENABLED_PREF = "app.normandy.enabled";
const DEV_MODE_PREF = "app.normandy.dev_mode";
const API_URL_PREF = "app.normandy.api_url";
const LAZY_CLASSIFY_PREF = "app.normandy.experiments.lazy_classify";
const ONSYNC_SKEW_SEC_PREF = "app.normandy.onsync_skew_sec";

// Timer last update preference.
// see https://searchfox.org/mozilla-central/rev/11cfa0462/toolkit/components/timermanager/UpdateTimerManager.jsm#8
const TIMER_LAST_UPDATE_PREF = `app.update.lastUpdateTime.${TIMER_NAME}`;

const PREFS_TO_WATCH = [RUN_INTERVAL_PREF, SHIELD_ENABLED_PREF, API_URL_PREF];

XPCOMUtils.defineLazyGetter(lazy, "gRemoteSettingsClient", () => {
  return lazy.RemoteSettings(REMOTE_SETTINGS_COLLECTION);
});

/**
 * cacheProxy returns an object Proxy that will memoize properties of the target.
 */
function cacheProxy(target) {
  const cache = new Map();
  return new Proxy(target, {
    get(target, prop, receiver) {
      if (!cache.has(prop)) {
        cache.set(prop, target[prop]);
      }
      return cache.get(prop);
    },
    set(target, prop, value, receiver) {
      cache.set(prop, value);
      return true;
    },
    has(target, prop) {
      return cache.has(prop) || prop in target;
    },
  });
}

export var RecipeRunner = {
  initializedPromise: PromiseUtils.defer(),

  async init() {
    this.running = false;
    this.enabled = null;
    this.loadFromRemoteSettings = false;
    this._syncSkewTimeout = null;

    this.checkPrefs(); // sets this.enabled
    this.watchPrefs();
    this.setUpRemoteSettings();

    // Here "first run" means the first run this profile has ever done. This
    // preference is set to true at the end of this function, and never reset to
    // false.
    const firstRun = Services.prefs.getBoolPref(FIRST_RUN_PREF, true);

    // If we've seen a build ID from a previous run that doesn't match the
    // current build ID, run immediately. This is probably an upgrade or
    // downgrade, which may cause recipe eligibility to change.
    let hasNewBuildID =
      Services.appinfo.lastAppBuildID != null &&
      Services.appinfo.lastAppBuildID != Services.appinfo.appBuildID;

    // Dev mode is a mode used for development and QA that bypasses the normal
    // timer function of Normandy, to make testing more convenient.
    const devMode = Services.prefs.getBoolPref(DEV_MODE_PREF, false);

    if (this.enabled && (devMode || firstRun || hasNewBuildID)) {
      // In dev mode, if remote settings is enabled, force an immediate sync
      // before running. This ensures that the latest data is used for testing.
      // This is not needed for the first run case, because remote settings
      // already handles empty collections well.
      if (devMode) {
        await lazy.gRemoteSettingsClient.sync();
      }
      let trigger;
      if (devMode) {
        trigger = "devMode";
      } else if (firstRun) {
        trigger = "firstRun";
      } else if (hasNewBuildID) {
        trigger = "newBuildID";
      }

      await this.run({ trigger });
    }

    // Update the firstRun pref, to indicate that Normandy has run at least once
    // on this profile.
    if (firstRun) {
      Services.prefs.setBoolPref(FIRST_RUN_PREF, false);
    }

    this.initializedPromise.resolve();
  },

  enable() {
    if (this.enabled) {
      return;
    }
    this.registerTimer();
    this.enabled = true;
  },

  disable() {
    if (this.enabled) {
      this.unregisterTimer();
    }
    // this.enabled may be null, so always set it to false
    this.enabled = false;
  },

  /** Watch for prefs to change, and call this.observer when they do */
  watchPrefs() {
    for (const pref of PREFS_TO_WATCH) {
      Services.prefs.addObserver(pref, this);
    }

    lazy.CleanupManager.addCleanupHandler(this.unwatchPrefs.bind(this));
  },

  unwatchPrefs() {
    for (const pref of PREFS_TO_WATCH) {
      Services.prefs.removeObserver(pref, this);
    }
  },

  /** When prefs change, this is fired */
  observe(subject, topic, data) {
    switch (topic) {
      case PREF_CHANGED_TOPIC: {
        const prefName = data;

        switch (prefName) {
          case RUN_INTERVAL_PREF:
            this.updateRunInterval();
            break;

          // explicit fall-through
          case SHIELD_ENABLED_PREF:
          case API_URL_PREF:
            this.checkPrefs();
            break;

          default:
            log.debug(
              `Observer fired with unexpected pref change: ${prefName}`
            );
        }

        break;
      }
    }
  },

  checkPrefs() {
    if (!Services.prefs.getBoolPref(SHIELD_ENABLED_PREF)) {
      log.debug(
        `Disabling Shield because ${SHIELD_ENABLED_PREF} is set to false`
      );
      this.disable();
      return;
    }

    const apiUrl = Services.prefs.getCharPref(API_URL_PREF);
    if (!apiUrl) {
      log.warn(`Disabling Shield because ${API_URL_PREF} is not set.`);
      this.disable();
      return;
    }
    if (!apiUrl.startsWith("https://")) {
      log.warn(
        `Disabling Shield because ${API_URL_PREF} is not an HTTPS url: ${apiUrl}.`
      );
      this.disable();
      return;
    }

    log.debug(`Enabling Shield`);
    this.enable();
  },

  registerTimer() {
    this.updateRunInterval();
    lazy.CleanupManager.addCleanupHandler(() =>
      lazy.timerManager.unregisterTimer(TIMER_NAME)
    );
  },

  unregisterTimer() {
    lazy.timerManager.unregisterTimer(TIMER_NAME);
  },

  setUpRemoteSettings() {
    if (this._alreadySetUpRemoteSettings) {
      return;
    }
    this._alreadySetUpRemoteSettings = true;

    if (!this._onSync) {
      this._onSync = this.onSync.bind(this);
    }
    lazy.gRemoteSettingsClient.on("sync", this._onSync);

    lazy.CleanupManager.addCleanupHandler(() => {
      lazy.gRemoteSettingsClient.off("sync", this._onSync);
      this._alreadySetUpRemoteSettings = false;
    });
  },

  /** Called when our Remote Settings collection is updated */
  async onSync() {
    if (!this.enabled) {
      return;
    }

    // Delay the Normandy run by a random amount, determined by preference.
    // This helps alleviate server load, since we don't have a thundering
    // herd of users trying to update all at once.
    if (this._syncSkewTimeout) {
      lazy.clearTimeout(this._syncSkewTimeout);
    }
    let minSkewSec = 1; // this is primarily is to avoid race conditions in tests
    let maxSkewSec = Services.prefs.getIntPref(ONSYNC_SKEW_SEC_PREF, 0);
    if (maxSkewSec >= minSkewSec) {
      let skewMillis =
        (minSkewSec + Math.random() * (maxSkewSec - minSkewSec)) * 1000;
      log.debug(
        `Delaying on-sync Normandy run for ${Math.floor(
          skewMillis / 1000
        )} seconds`
      );
      this._syncSkewTimeout = lazy.setTimeout(
        () => this.run({ trigger: "sync" }),
        skewMillis
      );
    } else {
      log.debug(`Not skewing on-sync Normandy run`);
      await this.run({ trigger: "sync" });
    }
  },

  updateRunInterval() {
    // Run once every `runInterval` wall-clock seconds. This is managed by setting a "last ran"
    // timestamp, and running if it is more than `runInterval` seconds ago. Even with very short
    // intervals, the timer will only fire at most once every few minutes.
    const runInterval = Services.prefs.getIntPref(RUN_INTERVAL_PREF);
    lazy.timerManager.registerTimer(TIMER_NAME, () => this.run(), runInterval);
  },

  async run({ trigger = "timer" } = {}) {
    if (this.running) {
      // Do nothing if already running.
      return;
    }
    this.running = true;

    await lazy.Normandy.defaultPrefsHaveBeenApplied.promise;

    try {
      this.running = true;
      Services.obs.notifyObservers(null, "recipe-runner:start");
      if (this._syncSkewTimeout) {
        lazy.clearTimeout(this._syncSkewTimeout);
        this._syncSkewTimeout = null;
      }

      this.clearCaches();
      // Unless lazy classification is enabled, prep the classify cache.
      if (!Services.prefs.getBoolPref(LAZY_CLASSIFY_PREF, false)) {
        try {
          await lazy.ClientEnvironment.getClientClassification();
        } catch (err) {
          // Try to go on without this data; the filter expressions will
          // gracefully fail without this info if they need it.
        }
      }

      // Fetch recipes before execution in case we fail and exit early.
      let recipesAndSignatures;
      try {
        recipesAndSignatures = await lazy.gRemoteSettingsClient.get({
          // Do not return an empty list if an error occurs.
          emptyListFallback: false,
        });
      } catch (e) {
        await lazy.Uptake.reportRunner(lazy.Uptake.RUNNER_SERVER_ERROR);
        return;
      }

      const actionsManager = new lazy.ActionsManager();

      const legacyHeartbeat = lazy.LegacyHeartbeat.getHeartbeatRecipe();
      const noRecipes =
        !recipesAndSignatures.length && legacyHeartbeat === null;

      // Execute recipes, if we have any.
      if (noRecipes) {
        log.debug("No recipes to execute");
      } else {
        for (const { recipe, signature } of recipesAndSignatures) {
          let suitability = await this.getRecipeSuitability(recipe, signature);
          await actionsManager.processRecipe(recipe, suitability);
        }

        if (legacyHeartbeat !== null) {
          await actionsManager.processRecipe(
            legacyHeartbeat,
            lazy.BaseAction.suitability.FILTER_MATCH
          );
        }
      }

      await actionsManager.finalize({ noRecipes });

      await lazy.Uptake.reportRunner(lazy.Uptake.RUNNER_SUCCESS);
      Services.obs.notifyObservers(null, "recipe-runner:end");
    } finally {
      this.running = false;
      if (trigger != "timer") {
        // `run()` was executed outside the scheduled timer.
        // Update the last time it ran to make sure it is rescheduled later.
        const lastUpdateTime = Math.round(Date.now() / 1000);
        Services.prefs.setIntPref(TIMER_LAST_UPDATE_PREF, lastUpdateTime);
      }
    }
  },

  getFilterContext(recipe) {
    const environment = cacheProxy(lazy.ClientEnvironment);
    environment.recipe = {
      id: recipe.id,
      arguments: recipe.arguments,
    };
    return {
      env: environment,
      // Backwards compatibility -- see bug 1477255.
      normandy: environment,
    };
  },

  /**
   * Return the set of capabilities this runner has.
   *
   * This is used to pre-filter recipes that aren't compatible with this client.
   *
   * @returns {Set<String>} The capabilities supported by this client.
   */
  getCapabilities() {
    let capabilities = new Set([
      "capabilities-v1", // The initial version of the capabilities system.
    ]);

    // Get capabilities from ActionsManager.
    for (const actionCapability of lazy.ActionsManager.getCapabilities()) {
      capabilities.add(actionCapability);
    }

    // Add a capability for each transform available to JEXL.
    for (const transform of lazy.FilterExpressions.getAvailableTransforms()) {
      capabilities.add(`jexl.transform.${transform}`);
    }

    // Add two capabilities for each top level key available in the context: one
    // for the `normandy.` namespace, and another for the `env.` namespace.
    capabilities.add("jexl.context.env");
    capabilities.add("jexl.context.normandy");
    let env = lazy.ClientEnvironment;
    while (env && env.name) {
      // Walk up the class chain for ClientEnvironment, collecting applicable
      // properties as we go. Stop when we get to an unnamed object, which is
      // usually just a plain function is the super class of a class that doesn't
      // extend anything. Also stop if we get to an undefined object, just in
      // case.
      for (const [name, descriptor] of Object.entries(
        Object.getOwnPropertyDescriptors(env)
      )) {
        // All of the properties we are looking for are are static getters (so
        // will have a truthy `get` property) and are defined on the class, so
        // will be configurable
        if (descriptor.configurable && descriptor.get) {
          capabilities.add(`jexl.context.env.${name}`);
          capabilities.add(`jexl.context.normandy.${name}`);
        }
      }
      // Check for the next parent
      env = Object.getPrototypeOf(env);
    }

    return capabilities;
  },

  /**
   * Decide if a recipe is suitable to run, and returns a value from
   * `BaseAction.suitability`.
   *
   * This checks several things in order:
   *  - recipe signature
   *  - capabilities
   *  - filter expression
   *
   * If the provided signature does not match the provided recipe, then
   * `SIGNATURE_ERROR` is returned. Recipes with this suitability should not be
   * trusted. These recipes are included so that temporary signature errors on
   * the server can be handled intelligently by actions.
   *
   * Capabilities are a simple set of strings in the recipe. If the Normandy
   * client has all of the capabilities listed, then execution continues. If
   * not, then `CAPABILITY_MISMATCH` is returned. Recipes with this suitability
   * should be considered incompatible and treated with caution.
   *
   * If the capabilities check passes, then the filter expression is evaluated
   * against the current environment. The result of the expression is cast to a
   * boolean. If it is true, then `FILTER_MATCH` is returned. If not, then
   * `FILTER_MISMATCH` is returned.
   *
   * If there is an error while evaluating the recipe's filter, `FILTER_ERROR`
   * is returned instead.
   *
   * @param {object} recipe
   * @param {object} signature
   * @param {string} recipe.filter_expression The expression to evaluate against the environment.
   * @param {Set<String>} runnerCapabilities The capabilities provided by this runner.
   * @return {Promise<BaseAction.suitability>} The recipe's suitability
   */
  async getRecipeSuitability(recipe, signature) {
    let generator = this.getAllSuitabilities(recipe, signature);
    // For our purposes, only the first suitability matters, so pull the first
    // value out of the async generator. This additionally guarantees if we fail
    // a security or compatibility check, we won't continue to run other checks,
    // which is good for the general case of running recipes.
    let { value: suitability } = await generator.next();
    switch (suitability) {
      case lazy.BaseAction.suitability.SIGNATURE_ERROR: {
        await lazy.Uptake.reportRecipe(
          recipe,
          lazy.Uptake.RECIPE_INVALID_SIGNATURE
        );
        break;
      }

      case lazy.BaseAction.suitability.CAPABILITIES_MISMATCH: {
        await lazy.Uptake.reportRecipe(
          recipe,
          lazy.Uptake.RECIPE_INCOMPATIBLE_CAPABILITIES
        );
        break;
      }

      case lazy.BaseAction.suitability.FILTER_MATCH: {
        // No telemetry needs to be sent for this right now.
        break;
      }

      case lazy.BaseAction.suitability.FILTER_MISMATCH: {
        // This represents a terminal state for the given recipe, so
        // report its outcome. Others are reported when executed in
        // ActionsManager.
        await lazy.Uptake.reportRecipe(
          recipe,
          lazy.Uptake.RECIPE_DIDNT_MATCH_FILTER
        );
        break;
      }

      case lazy.BaseAction.suitability.FILTER_ERROR: {
        await lazy.Uptake.reportRecipe(
          recipe,
          lazy.Uptake.RECIPE_FILTER_BROKEN
        );
        break;
      }

      case lazy.BaseAction.suitability.ARGUMENTS_INVALID: {
        // This shouldn't ever occur, since the arguments schema is checked by
        // BaseAction itself.
        throw new Error(`Shouldn't get ${suitability} in RecipeRunner`);
      }

      default: {
        throw new Error(`Unexpected recipe suitability ${suitability}`);
      }
    }

    return suitability;
  },

  /**
   * Some uses cases, such as Normandy Devtools, want the status of all
   * suitabilities, not only the most important one. This checks the cases of
   * suitabilities in order from most blocking to least blocking. The first
   * yielded is the "primary" suitability to pass on to actions.
   *
   * If this function yields only [FILTER_MATCH], then the recipe fully matches
   * and should be executed. If any other statuses are yielded, then the recipe
   * should not be executed as normal.
   *
   * This is a generator so that the execution can be halted as needed. For
   * example, after receiving a signature error, a caller can stop advancing
   * the iterator to avoid exposing the browser to unneeded risk.
   */
  async *getAllSuitabilities(recipe, signature) {
    try {
      await lazy.NormandyApi.verifyObjectSignature(recipe, signature, "recipe");
    } catch (e) {
      yield lazy.BaseAction.suitability.SIGNATURE_ERROR;
    }

    const runnerCapabilities = this.getCapabilities();
    if (Array.isArray(recipe.capabilities)) {
      for (const recipeCapability of recipe.capabilities) {
        if (!runnerCapabilities.has(recipeCapability)) {
          log.debug(
            `Recipe "${recipe.name}" requires unknown capabilities. ` +
              `Recipe capabilities: ${JSON.stringify(recipe.capabilities)}. ` +
              `Local runner capabilities: ${JSON.stringify(
                Array.from(runnerCapabilities)
              )}`
          );
          yield lazy.BaseAction.suitability.CAPABILITIES_MISMATCH;
        }
      }
    }

    const context = this.getFilterContext(recipe);
    const targetingContext = new lazy.TargetingContext();
    try {
      if (await targetingContext.eval(recipe.filter_expression, context)) {
        yield lazy.BaseAction.suitability.FILTER_MATCH;
      } else {
        yield lazy.BaseAction.suitability.FILTER_MISMATCH;
      }
    } catch (err) {
      log.error(
        `Error checking filter for "${recipe.name}". Filter: [${recipe.filter_expression}]. Error: "${err}"`
      );
      yield lazy.BaseAction.suitability.FILTER_ERROR;
    }
  },

  /**
   * Clear all caches of systems used by RecipeRunner, in preparation
   * for a clean run.
   */
  clearCaches() {
    lazy.ClientEnvironment.clearClassifyCache();
    lazy.NormandyApi.clearIndexCache();
  },

  /**
   * Clear out cached state and fetch/execute recipes from the given
   * API url. This is used mainly by the mock-recipe-server JS that is
   * executed in the browser console.
   */
  async testRun(baseApiUrl) {
    const oldApiUrl = Services.prefs.getCharPref(API_URL_PREF);
    Services.prefs.setCharPref(API_URL_PREF, baseApiUrl);

    try {
      lazy.Storage.clearAllStorage();
      this.clearCaches();
      await this.run();
    } finally {
      Services.prefs.setCharPref(API_URL_PREF, oldApiUrl);
      this.clearCaches();
    }
  },

  /**
   * Offer a mechanism to get access to the lazily-instantiated
   * gRemoteSettingsClient, because if someone instantiates it
   * themselves, it won't have the options we provided in this module,
   * and it will prevent instantiation by this module later.
   *
   * This is only meant to be used in testing, where it is a
   * convenient hook to store data in the underlying remote-settings
   * collection.
   */
  get _remoteSettingsClientForTesting() {
    return lazy.gRemoteSettingsClient;
  },

  migrations: {
    /**
     * Delete the now-unused collection of recipes, since we are using the
     * "normandy-recipes-capabilities" collection now.
     */
    async migration01RemoveOldRecipesCollection() {
      // Don't bother to open IDB and clear on clean profiles.
      const lastCheckPref =
        "services.settings.main.normandy-recipes.last_check";
      if (Services.prefs.prefHasUserValue(lastCheckPref)) {
        // We instantiate a client, but it won't take part of sync.
        const client = new lazy.RemoteSettingsClient("normandy-recipes");
        await client.db.clear();
        Services.prefs.clearUserPref(lastCheckPref);
      }
    },
  },
};
```