```
/* 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 { AppConstants } from "resource://gre/modules/AppConstants.sys.mjs";

const lazy = {};

ChromeUtils.defineESModuleGetters(lazy, {
  ClientEnvironment: "resource://normandy/lib/ClientEnvironment.sys.mjs",
  ClientEnvironmentBase:
    "resource://gre/modules/components-utils/ClientEnvironment.sys.mjs",
  FilterExpressions:
    "resource://gre/modules/components-utils/FilterExpressions.sys.mjs",
  TelemetryEnvironment: "resource://gre/modules/TelemetryEnvironment.sys.mjs",
  clearTimeout: "resource://gre/modules/Timer.sys.mjs",
  setTimeout: "resource://gre/modules/Timer.sys.mjs",
});

XPCOMUtils.defineLazyModuleGetters(lazy, {
  ASRouterTargeting: "resource://activity-stream/lib/ASRouterTargeting.jsm",
});

const TARGETING_EVENT_CATEGORY = "messaging_experiments";
const TARGETING_EVENT_METHOD = "targeting";
const DEFAULT_TIMEOUT = 5000;
const ERROR_TYPES = {
  ATTRIBUTE_ERROR: "attribute_error",
  TIMEOUT: "attribute_timeout",
};

const TargetingEnvironment = {
  get locale() {
    return lazy.ASRouterTargeting.Environment.locale;
  },

  get localeLanguageCode() {
    return lazy.ASRouterTargeting.Environment.localeLanguageCode;
  },

  get region() {
    return lazy.ASRouterTargeting.Environment.region;
  },

  get userId() {
    return lazy.ClientEnvironment.userId;
  },

  get version() {
    return AppConstants.MOZ_APP_VERSION_DISPLAY;
  },

  get channel() {
    const { settings } = lazy.TelemetryEnvironment.currentEnvironment;
    return settings.update.channel;
  },

  get platform() {
    return AppConstants.platform;
  },

  get os() {
    return lazy.ClientEnvironmentBase.os;
  },
};

export class TargetingContext {
  #telemetrySource = null;

  constructor(customContext, options = { source: null }) {
    if (customContext) {
      this.ctx = new Proxy(customContext, {
        get: (customCtx, prop) => {
          if (prop in TargetingEnvironment) {
```

```
          return TargetingEnvironment[prop];
        }
        return customCtx[prop];
      },
    });
  } else {
    this.ctx = TargetingEnvironment;
  }

  // Used in telemetry to report where the targeting expression is coming from
  this.#telemetrySource = options.source;

  // Enable event recording
  Services.telemetry.setEventRecordingEnabled(TARGETING_EVENT_CATEGORY, true);
}

setTelemetrySource(source) {
  if (source) {
    this.#telemetrySource = source;
  }
}

_sendUndesiredEvent(eventData) {
  if (this.#telemetrySource) {
    Services.telemetry.recordEvent(
      TARGETING_EVENT_CATEGORY,
      TARGETING_EVENT_METHOD,
      eventData.event,
      eventData.value,
      { source: this.#telemetrySource }
    );
  } else {
    Services.telemetry.recordEvent(
      TARGETING_EVENT_CATEGORY,
      TARGETING_EVENT_METHOD,
      eventData.event,
      eventData.value
    );
  }
}

/**
 * Wrap each property of context[key] with a Proxy that captures errors and
 * timeouts
 *
 * @param {Object.<string, TargetingGetters> | TargetingGetters} context
 * @param {string} key Namespace value found in `context` param
 * @returns {TargetingGetters} Wrapped context where getter report errors and timeouts
 */
createContextWithTimeout(context, key = null) {
  const timeoutDuration = key ? context[key].timeout : context.timeout;
  const logUndesiredEvent = (event, key, prop) => {
    const value = key ? `${key}.${prop}` : prop;
    this._sendUndesiredEvent({ event, value });
    console.error(`${event}: ${value}`);
  };

  return new Proxy(context, {
    get(target, prop) {
      // eslint-disable-next-line no-async-promise-executor
      return new Promise(async (resolve, reject) => {
        // Create timeout cb to record attribute resolution taking too long.
        let timeout = lazy.setTimeout(() => {
          logUndesiredEvent(ERROR_TYPES.TIMEOUT, key, prop);
          reject(
            new Error(
              `${prop} targeting getter timed out after ${
                timeoutDuration || DEFAULT_TIMEOUT
              }ms`
            )
          );
        }, timeoutDuration || DEFAULT_TIMEOUT);

        try {
          resolve(await (key ? target[key][prop] : target[prop]));
        } catch (error) {
          logUndesiredEvent(ERROR_TYPES.ATTRIBUTE_ERROR, key, prop);
          reject(error);
          console.error(error);
        } finally {
          lazy.clearTimeout(timeout);
        }
      });
    },
  });
}

/**
 * Merge all evaluation contexts and wrap the getters with timeouts
 *
 * @param {Object.<string, TargetingGetters>[]} contexts
 * @returns {Object.<string, TargetingGetters>} Object that follows the pattern of `namespace: getters`
 */
mergeEvaluationContexts(contexts) {
  let context = {};
  for (let c of contexts) {
    for (let envNamespace of Object.keys(c)) {
      // Take the provided context apart, replace it with a proxy
      context[envNamespace] = this.createContextWithTimeout(c, envNamespace);
    }
  }

  return context;
}

/**
 * Merge multiple TargetingGetters objects without accidentally evaluating
 *
 * @param {TargetingGetters[]} ...contexts
 * @returns {Proxy<TargetingGetters>}
 */
static combineContexts(...contexts) {
  return new Proxy(
    {},
    {
      get(target, prop) {
        for (let context of contexts) {
          if (prop in context) {
            return context[prop];
          }
        }

        return null;
      },
    }
  );
}

/**
 * Evaluate JEXL expressions with default `TargetingEnvironment` and custom
 * provided targeting contexts
 *
 * @example
 * eval(
 *   "ctx.locale == 'en-US' && customCtx.foo == 42",
 *   { customCtx: { foo: 42 } }
 * ); // true
 *
 * @param {string} expression JEXL expression
 * @param {Object.<string, TargetingGetters>[]} ...contexts Additional custom context
 *        objects where the keys act as namespaces for the different getters
 *
 * @returns {promise} Evaluation result
 */
eval(expression, ...contexts) {
  return lazy.FilterExpressions.eval(
    expression,
    this.mergeEvaluationContexts([{ ctx: this.ctx }, ...contexts])
  );
}
```

```
  /**
   * Evaluate JEXL expressions with default provided targeting context
   *
   * @example
   * new TargetingContext({ bar: 42 });
   * evalWithDefault(
   *   "bar == 42",
   * ); // true
   *
   * @param {string} expression JEXL expression
   * @returns {promise} Evaluation result
   */
  evalWithDefault(expression) {
    return lazy.FilterExpressions.eval(
      expression,
      this.createContextWithTimeout(this.ctx)
    );
  }
}
```