```
// -*- Mode: c++; tab-width: 2; indent-tabs-mode: nil; c-basic-offset: 2 -*-
//
// Copyright (C) 2016 Opera Software AS.  All rights reserved.
//
// This file is an original work developed by Opera Software AS

'use strict';

class VideoHandler {
  static get OBSERVER_ATTRIBUTE_FILTER() { return ['class', 'hidden']; }

  static get OBSERVER_FILTERS() {
    const filters = {
      'attributes': true,
      'childList': true,
      'subtree': true,
    };

    if (this.OBSERVER_ATTRIBUTE_FILTER) {
      filters.attributeFilter = this.OBSERVER_ATTRIBUTE_FILTER;
    }

    return filters;
  }

  static get SELECTOR_PLAYER() { return null; }
  static get SELECTOR_PLAYER_SKIP_AD() { return null; }
  static get SELECTOR_PLAYER_FFORWARD() { return null; }
  static get SELECTOR_PLAYER_SEEK_BAR() { return null; }
  static get SELECTOR_PLAYER_LIVE() { return null; }
  static get SELECTOR_PLAYER_VOLUME() { return null; }

  constructor() {
    this.player_ = null;
    this.playerObserver_ = new MutationObserver(() => this.onPlayerChange_());
    this.state_ = {};
    this.trackedVideo_ = null;
    this.isScrubbing_ = false;
    this.isPausedForScrubbing_ = false;

    this.initializeActionHandlers_();

    this.onCreateBound_ = this.onCreate_.bind(this);
    this.onDetachBound_ = this.onDetach_.bind(this);
    this.onReleaseBound_ = this.onRelease_.bind(this);
    this.addListeners_();

    VideoHandler.getCurrentURL = this.getCurrentURL_.bind(this);
    VideoHandler.SUPPORTS_SEND_TO_PHONE = this.supportsSendToPhone_();
  }

  initializeActionHandlers_() {
    // Maps action names to our own MediaSession action handlers.
    this.actionHandlers_ = new Object();
    this.actionHandlers_['skipad'] = () => {
      this.triggerClick_(this.constructor.SELECTOR_PLAYER_SKIP_AD);
    };
    this.actionHandlers_['nexttrack'] = this.triggerFforwardClick_.bind(this);
    this.actionHandlers_['seekto'] = this.onSeekToAction_.bind(this);

    // Maps action names to IDs of our own MediaSession action handlers.
    this.actionHandlerIds_ = new Object();
  }

  addListeners_() {
    VideoHandler.Events.onCreate.addListener(this.onCreateBound_);
    VideoHandler.Events.onDetach.addListener(this.onDetachBound_);
    VideoHandler.Events.onRelease.addListener(this.onReleaseBound_);
  }

  hasCustomDuration_() {
    return typeof this.getDuration_ === 'function';
  }

  checkCustomControls_() {
    let enabledActions = [];

    if (this.hasSkipAdControl_()) {
      enabledActions.push('skipad');
    }
    if (this.hasFforwardControl_()) {
      enabledActions.push('nexttrack');
    }
    if (!this.isLive_() && !this.isSeekBarHidden_()) {
      enabledActions.push('seekto');
    }

    this.updateMediaSessionActions_(enabledActions);
  }

  getPlayer_(video) {
    return this.constructor.SELECTOR_PLAYER &&
        video.closest(this.constructor.SELECTOR_PLAYER);
  }

  getPlayerElement_(selector) {
    return this.player_ && this.player_.querySelector(selector);
  }

  // Gets url to current video including time markers.
  // Note: This implementation was made for youtube.com and Twitch.tv only.
  // For other sites doubletheck if it works and override in subclasses if
  // necessary.
  getCurrentURL_(videoElement) {
    if (this.isLive_()) {
      return location.href;
    }

    const time = parseInt(videoElement.currentTime);
    const timeRegex = /t=[\dhms]+s/;
    let searchPart = location.search;
    if (location.search.search(timeRegex) >= 0) {
      searchPart = location.search.replace(timeRegex, `t=${time}s`);
    } else {
      searchPart += `${searchPart === '' ? '?' : '&'}t=${time}s`;
    }

    if (location.search === '') {
      return location.href + searchPart;
    }
    return location.href.replace(location.search, searchPart);
  }

  supportsSendToPhone_() {
    return false;
  }

  hasSkipAdControl_() {
    if (this.constructor.SELECTOR_PLAYER_SKIP_AD) {
      return Boolean(
          this.getPlayerElement_(this.constructor.SELECTOR_PLAYER_SKIP_AD));
    }

    return false;
  }

  hasFforwardControl_() {
    if (this.constructor.SELECTOR_PLAYER_FFORWARD) {
      return Boolean(
          this.getPlayerElement_(this.constructor.SELECTOR_PLAYER_FFORWARD));
    }

    return false;
  }

  isSeekBarHidden_() {
    if (this.constructor.SELECTOR_PLAYER_SEEK_BAR) {
      return !Boolean(
          this.getPlayerElement_(this.constructor.SELECTOR_PLAYER_SEEK_BAR));
    }
    return false;
  }

  isLive_() {
    if (this.constructor.SELECTOR_PLAYER_LIVE) {
      return Boolean(
          this.getPlayerElement_(this.constructor.SELECTOR_PLAYER_LIVE));
    }

    return false;
  }

  onCreate_(video) {
    const player = this.getPlayer_(video);
    if (!player) {
      return;
    }

    // The video element may have changed, so let the listeners below know.
    player[VideoHandler.VIDEO_ELEMENT] = video;

    if (this.player_ === player) {
      return;
    }

    this.player_ = player;
    video[VideoHandler.PLAYER_ELEMENT] = player;
    this.player_.addEventListener('mousemove', () => {
      const video = this.player_[VideoHandler.VIDEO_ELEMENT];
      if (video) {
        VideoHandler.Events.onVideoAreaOver.dispatch(video);
      }
    });
    this.player_.addEventListener('mouseout', () => {
      const video = this.player_[VideoHandler.VIDEO_ELEMENT];
      if (video) {
        VideoHandler.Events.onVideoAreaOut.dispatch(video);
      }
    });
  }

  onDetach_(video) {
    if (this.trackedVideo_) {
      if (this.trackedVideo_ === video) {
        return;
      }

      this.onRelease_(this.trackedVideo_);
    }
    this.trackedVideo_ = video;
    this.state_ = {};
    this.track_(video);

    this.updateMediaSessionActions_(['seekto']);
  }

  onPlayerChange_() { }

  onRelease_(video) {
    if (!video || this.trackedVideo_ !== video) {
      return;
    }

    this.updateMediaSessionActions_([]);

    this.trackedVideo_ = null;
    this.untrack_(video);
  }

  track_() {
    this.player_ = this.getPlayer_(this.trackedVideo_);
    if (this.player_) {
      this.playerObserver_.observe(
          this.player_, this.constructor.OBSERVER_FILTERS);
      this.onPlayerChange_();
    }
  }

  triggerClick_(selector) {
    if (!selector) {
      return false;
    }

    const element = this.getPlayerElement_(selector);
    if (element) {
      element.click();
      return true;
    }

    return false;
  }

  triggerFforwardClick_() {
    return this.triggerClick_(this.constructor.SELECTOR_PLAYER_FFORWARD);
  }

  onSeekToAction_(actionDetails) {
    if (!this.isScrubbing_ && actionDetails.fastSeek) {
      this.beginScrubbing_();
    }

    // TODO: Use HTMLMediaElement.fastSeek() when it's available.
    this.trackedVideo_.currentTime = actionDetails.seekTime;

    // §10 of the MediaSession spec says |fastSeek| "will be true if the action
    // is being called multiple times as part of a sequence and this is not the
    // last call in that sequence."
    if (this.isScrubbing_ && !actionDetails.fastSeek) {
      this.endScrubbing_();
    }
  }

  beginScrubbing_() {
    if (!this.trackedVideo_.paused) {
      this.trackedVideo_.pause();
      this.isPausedForScrubbing_ = true;
    }
    this.isScrubbing_ = true;
  }

  endScrubbing_() {
    if (this.isPausedForScrubbing_) {
      this.trackedVideo_.play();
      this.isPausedForScrubbing_ = false;
    }
    this.isScrubbing_ = false;
  }

  untrack_(video) {
    this.playerObserver_.disconnect();
  }

  // Make sure our handlers for actions specified in |actions| are enabled,
  // removing any action handlers set by us previously for actions that are not
  // in |actions|.
  updateMediaSessionActions_(actions) {
    for (const action in this.actionHandlers_) {
      if (actions.includes(action)) {
        this.maybeEnableMediaSessionActionHandler_(action);
      } else {
        this.maybeDisableMediaSessionActionHandler_(action);
      }
    }
  }

  // Set up our custom action handler iff we know the page doesn't provide its
  // own handler.
  maybeEnableMediaSessionActionHandler_(action) {
    const currentHandlerId = opr.detachedVideoPrivate.getActionHandlerId(
        navigator.mediaSession, action);
    if (currentHandlerId === this.actionHandlerIds_[action]) {
      // Already enabled.
      return;
    }

    const pageHandlesAction = currentHandlerId !== 0;
    if (!pageHandlesAction) {
      navigator.mediaSession.setActionHandler(
          action, this.actionHandlers_[action]);
      const newHandlerId = opr.detachedVideoPrivate.getActionHandlerId(
          navigator.mediaSession, action);
      this.actionHandlerIds_[action] = newHandlerId;
    }
  }

  maybeDisableMediaSessionActionHandler_(action) {
    if (!this.actionHandlerIds_[action]) {
      // Already disabled.
      return;
    }

    const currentHandlerId = opr.detachedVideoPrivate.getActionHandlerId(
        navigator.mediaSession, action);
    // The page may have overwritten our handler, that's fine.
    if (this.actionHandlerIds_[action] === currentHandlerId) {
      navigator.mediaSession.setActionHandler(action, null);
    }
    this.actionHandlerIds_[action] = null;
  }
}

VideoHandler.PLAYER_ELEMENT = Symbol();
VideoHandler.VIDEO_ELEMENT = Symbol();

VideoHandler.Events = {
  onCreate: opr.detachedVideoPrivate.onCreate,
  onDetach: opr.detachedVideoPrivate.onDetach,
  onRelease: opr.detachedVideoPrivate.onRelease,
  onVideoAreaOut: opr.detachedVideoPrivate.onVideoAreaOut,
  onVideoAreaOver: opr.detachedVideoPrivate.onVideoAreaOver,
```