# Keweenaw Time Traveler Mobile Storytelling Documentation

Bob Cowling, 6/12/2023

## Application and User Description

The Keweenaw Time Traveler Mobile Storytelling application enables users to explore historical maps and stories in the Keweenaw Peninsula in Michigan. The details of specific stories can be viewed by clicking on icons on the map. Users also have the option to submit their own stories including text, pictures, audio and video. Historical maps can be viewed by clicking on the maps tab. The application comes with a GPS button that will show the users geolocation on the map. Users can either navigate the map by manually panning and zoom or by using the Geolocation button and walking around the landscape. Maps will be shown in the Maps tab that correspond to the user's location on the map.

Although this application will work well on a desktop browser, the application was designed specifically for mobile devices. The target audience for the application was the general public and therefore it is designed to be intuitive and easy to use. The application was designed as a field app with the intent the user would be able to easily navigate the map, view historical maps and stories and even submit their own stories right from their mobile device.

## Resources for Reading this Documentation

This document is intended for web developers and provides an in depth description of the frontend code base for the application. A reader of this documentation should have a solid understanding of

- JavaScript
- CSS
- HTML

MDN is the best site for learning and referencing details of the above technologies, https://developer.mozilla.org/en-US/

The application was created using Ionic Framework, ArcGIS JavaScript SDK, ArcGIS REST JS, History API, Media Capture and Streams API, and JQuery. There are numerous resources available for learning about these technologies:

- Ionic Documentation, https://ionicframework.com/docs

- The map uses ArcGIS Maps SDK for JS, https://developers.arcgis.com/javascript/latest/
- ArcGIS JS REST is used to query data, https://developers.arcgis.com/arcgis-rest-js/
- Media Capture and Streams API for audio recording, https://developer.mozilla.org/en-US/docs/Web/API/Media_Capture_and_Streams_API
- History API for dynamic URL generation, https://developer.mozilla.org/en-US/docs/Web/API/History_API
- JQuery is also used to attach DOM listeners, https://api.jquery.com/

The code for the application is available on GitHub, https://github.com/Keweenaw-Time-Traveler/ktt-mobile. The application can be viewed live at, https://kett.geospatialresearch.mtu.edu/mobile or by going to https://kett.geospatialresearch.mtu.edu on a mobile device. The application can be launched by simply cloning or downloading the repository from GitHub and then running on a local web server.

# Example User Interaction

Below is an example of how a typical user might interact with the application and how the application would respond.

- The user visits the Keweenaw Time Traveler application on a mobile device.
- The app presents a modal popup window explaining the purpose of the application. The user closes the modal window.
- The user is presented with a map centered on the Keweenaw Peninsula full of story points, a Geolocation button, an opacity slider, and three tabs for Explore, Maps, and Share a Story.
- The application makes a series of API and map tile calls to load the story points FeatureLayer and populate a list of maps on the maps tab from a map indexes FeatureLayer. These layers will be explained later in the document.
- The user presses the geolocation button.
- The app makes a series of API and map tile calls, centers the map and places a point on the map at the user's geolocation. The maps list is updated to correspond with maps visible in the user's current  map extent.
- The user clicks on a map from the maps list.
- The app makes API and map tile calls and loads the corresponding map onto the screen.
- The user clicks on a story point. The app makes a series to API calls to query the stories layer and any corresponding media (attachments). The user is then presented with a modal window containing text and media for that specific story.
- The user clicks the ‘Share a Story’ tab.
- The user clicks the option to share the story ‘From a location on the map’

- The app switches to the ‘Explore’ tab and instructs the user to do a long press on the map for the location of their story. The app records the latitude and longitude values of the press.
- The user fills in story information in text fields and takes a picture with their phone.
- The app makes API calls to the Story Points layer to add a new feature and add the picture as an attachment using ArcGIS REST JS.

# App Architecture

The application is built in vanilla JavaScript and does not combine the Ionic Framework with any library such as React or Angular. All dependencies are included in the application using script and style tags in the <head> section of the index.html file. Therefor the applications architecture consists of the following:

## JS Packages

- Ionic 6.2.5, https://cdn.jsdelivr.net/npm/@ionic/core@6.2.5/dist/ionic/ionic.js
- JQuery 3.5.1, https://code.jquery.com/jquery-3.5.1.js
- Jquery-cookie 1.4.1, https://cdnjs.cloudflare.com/ajax/libs/jquery-cookie/1.4.1/jquery.cookie.min.js
- Linkify JS 3.0.5, "https://cdn.jsdelivr.net/npm/linkifyjs@3.0.5/dist/linkify.min.js
- ArcGIS JS SDK 4.24, https://js.arcgis.com/4.24/
- ArcGIS Rest Request 4.0.0, https://unpkg.com/@esri/arcgis-rest-request@4.0.0/dist/bundled/request.umd.js
- ArcGIS Rest Feature Service 4.0.0, https://unpkg.com/@esri/arcgis-rest-feature-service@4.0.0/dist/bundled/feature-service.umd.js

## CSS Packages

- Ionic 6.2.5, https://cdn.jsdelivr.net/npm/@ionic/core@6.2.5/css/ionic.bundle.css
- ArcGIS 4.24, https://js.arcgis.com/4.24/esri/themes/light/main.css
- Dafreefont, https://www.dafontfree.net/embed/cGF0cmljaWFuLXJlZ3VsYXImZGF0YS8yNi9wLzEzODA1Ny9QYXRyaWNhbi50dGY
- Font Lato, https://fonts.googleapis.com/css2?family=Gentium+Book+Plus&family=Lato:wght@300;400&display=swap

## Directory Structure

The main directory contains an index.html file and a folder called ‘/assets’ The application consists of one single index.html file. All of the HTML, CSS, and JavaScript code is in this file.

The /assets/ directory contains logos and markers used in the app. There are several files in the directory but the ones used in the app are:

- Main app Keweenaw Time Traveler logo, logo_mark_only%402x.a10ff43f.png
- Marker icon for stories, marker_story.png

## Layers

The application consists of two main layers all which are hosted on the Geospatial Research Facilities Portal1 Server. The layers are used to display stories and to display and query maps. The Stories layer produces the Story Points and is used to query story attribute information on click. The KeTT Mobile Map Indexes is an invisible polygon layer containing attributes with service URLs and titles of several historical map tile layers. It is used to populate the information for the maps tab. The layer service URLs are:

1. Keweenaw Time Traveler Stories, https://portal1-geo.sabu.mtu.edu:6443/arcgis/rest/services/KeweenawHSDI/story_pts_watts2/FeatureServer
2. KeTT Mobile Map Indexes, https://portal1-geo.sabu.mtu.edu/server/rest/services/Hosted/KeTT_Mobile_Map_Indexes/FeatureServer

# Major program functions

This section will explain the major functions of the application's code. The code for the application itself is fairly basic so only major functions will be covered in this documentation. The main functions of the application's code include querying maps for the 'Maps' tab and Submitting and viewing existing story submissions.

## Querying Maps

When the application is loaded, the applications waits for the layer (indexLayer) to be added to the maps view and checks it using whenLayerView (https://developers.arcgis.com/javascript/latest/api-reference/esri-views-MapView.html#whenLayerView). It then watches the layer view for updating values such as a change in the map's extent or zoom level. The application then waits for the layer to finish updating.

```
view.whenLayerView(indexLayer).then(function(layerView) {
  layerView.watch("updating", function(value) {
    If (!value) {
```

…

The application then queries the layerView for geometries intersecting with the map's current extent (view.extent). The results are returned to the browser as graphics.

```
layerView
  queryFeatures({
     geometry: view.extent,
     returnGeometry: true
  })
     .then(function(results) {
         // do something with the resulting graphics
          graphics = results.features;
          …
```

The attributes of the resulting graphics are then appended to the Map tabs ion-content as ion-items.

```
graphics.forEach(function(result, index) {
    const attributes = result.attributes;
     const title = attributes.title;
     const year = attributes.year;
     console.log(title);
      item = document.createElement("ion-item");
      const label = document.createElement("ion-label");
      const icon = document.createElement("ion-icon");
      item.href = "#";
      icon.name = "map-outline";
      item.detail = "false";
      icon.slot = "end";
       label.innerHTML = year + " " + title;
      item.appendChild(icon);
      item.appendChild(label);
      item.addEventListener("click", () => resultClickHandler(result, index));
      fragment.appendChild(item);
      listNode.appendChild(fragment);
});
```

The final piece of this component is the resultClickHandler function. This is what happens when a user clicks on one of the list-items in the Maps tab. The application loops over all of the existing layers on the map and checks for ones that are a type of ‘tile’ If there is an existing one, it is removed. It then gets the service URL from the clicked item’s attributes and uses it to add a new tileLayer to the map.

```
function resultClickHandler(result, index) {
    tabs.select('home');
    const popup = graphics && graphics[parseInt(index, 10)];
     map.allLayers.filter(function(layer) {
         if (layer.type === "tile") {
             map.remove(layer);
         }
     });

     let current_layer = new TileLayer({
         url: result.attributes.service_url
     });

      map.add(current_layer);
      …
```

We can then allow the user to adjust the opacity of the current_layer using the ion-range slider.

```
let rangeVal = $( "ion-range" ).val();

current_layer.opacity = rangeVal / 100;
```

## Viewing Stories

The loading of stories in the Keweenaw Time Traveler Mobile application is a little complex because it does not use the standard Esri Popup but uses an ion-modal instead. So all attributes and attachments must be queried and added to the modal through HTML and JS.

When the user clicks on a point from the storyLayer the application calls a function called buildStory(graphic) which takes the currently clicked graphic which takes the graphics attributes and add them to HTML elements.

```
function buildStory (graphic) {
  document.getElementById("pictures").innerHTML = "";
  document.getElementById("videos").innerHTML = "";
  document.getElementById("sound").innerHTML = "";
  const objectId = graphic.attributes.objectid;
  const title = graphic.attributes.title;
  const desc = linkifyHtml(graphic.attributes.description);
  const date = graphic.attributes.userdate;
  const name = graphic.attributes.name;
  $('#subtitle').html(date);
  $('#storydesc').html(desc);
  $('#storytitle').html(title);
```

```
  $('#storymodaltitle').html(title);
   …
```

The function also queries any existing attachments from the clicked graphic.

```
storyLayer.queryAttachments({
          objectIds: objectId
         }).then(function (attachmentsByFeatureId) {
          console.log(attachmentsByFeatureId);
          if (!attachmentsByFeatureId) {
              return;
           }
          if (Object.keys(attachmentsByFeatureId).length === 0){
           .
            $("#storyimg").attr("src", "");
           }
          // Display the attachments
          Object.keys(attachmentsByFeatureId)
            .forEach(function(objectId) {
                …
```

The application then loops over the attachments and pushes them into arrays corresponding to different types of media (video, images, etc). The following will show an example of an image attachment:

```
const attachments = attachmentsByFeatureId[objectId];
attachments.forEach(function (attachment) {
  if (attachment.contentType == "image/jpeg" || attachment.contentType == "image/png" ||
attachment.contentType == "image/gif") {
  const imgarray = [];
  imgarray.push(attachment);
  const attachUrl = imgarray[0].url;
  $("#storyimg").attr("src",attachUrl);
  …
```

## Submitting a Story

The story submission is the most complex functionality of the application. It includes a function for parsing date values and a custom audio recorder. All parts of this functionality will not be covered in this documentation.

The custom audio recorder simply uses the Media Capture API. The recorder checks if the user's device supports getUserMedia and then starts recording audio.

```
 // check if the user has a microphone
if (navigator.mediaDevices && navigator.mediaDevices.getUserMedia) {
  navigator.mediaDevices
   .getUserMedia(
    // constraints - only audio needed for this app
    {
     audio: true,
    }
   )
   // Success callback
   .then(function (stream) {
      const mediaRecorder = new MediaRecorder(stream, options);
      mediaRecorder.start();
      …
```

When the recorder is stopped the application turns the recording into a blob, converts it into an .mp3 and pushes it into a files array for the story submission.

```
mediaRecorder.onstop = function (e) {
    const blob = new Blob(chunks, { type: "audio/mpeg" });
     chunks = [];
     const audioURL = window.URL.createObjectURL(blob);
     const file = new File([blob], 'audiostory.mp3', {
     type: blob.type,
});

files.push(file);
…
```

When the user submits a story the application gets all of the values from the story submission form fields and adds them as attributes in the existing storiesLayer along with the geographic location (latitude and longitude values) of the story. The application uses a function called parseDates(date) to determine the type of date format, if it is a valid date, and if it is not in the future.

```
const parsedDate = parseDateEntry(date);

const featureServiceLayerUrl =
"https://portal1-geo.sabu.mtu.edu/server/rest/services/KeweenawHSDI/story_pts_watts2/FeatureServer/0";
// create a new feature to add using the REST API
const featureToAdd = {
  attributes: {
```

```
    title: title,
    description: desc,
    name: name,
    beginDate: parsedDate.beginDate.getTime(),
    endDate: parsedDate.endDate.getTime(),
    userdate: date,
    mapyear: mapyear,
    Tags: "mobile"
    },
    geometry: {
        x: lon,
        y: lat,
        spatialReference: {
          wkid: 4326
       }
    }
};

// begin by adding a new feature to the feature service layer
arcgisRest
  .addFeatures({
   url: featureServiceLayerUrl,
   features: [featureToAdd]
})
```

Next any attachments are added and the storyLayer is refreshed showing the user's newly submitted story.

```
function handleAdded(response) {
      if (response.addResults[0].success === true) {
        const storyObjId = response.addResults[0].objectId;          ;
        if (files.length > 0) {
          files.forEach(function(file) {
          arcgisRest
            .addAttachment({
              url: featureServiceLayerUrl,
              featureId: storyObjId,
              attachment: file
            })
              .then(attachmentAdded);
              function attachmentAdded (newresponse) {
                submitAlert.dismiss();
                slides.update();
                slides.slideNext();
```

```
            storyLayer.refresh();
        }
    });
```

Note that the location of the story point corresponds either to the user's current location on GPS or from a point the user manually selects on the map, whichever option they choose.

```
lat = position.coords.latitude;
lon = position.coords.longitude;
```