MyPhotos: An Offline Web App Example

Robert Pastel, 4/3/2015 12:07 PM, 4/6/2015 7:57 PM, 4/9/2015 7:57 PM, 4/16/2015 6:43 AM, 4/18/2015 10:06 AM, 9/30/2015

myPhotos Web App

The purpose of the myPhotos web app is to illustrate how an offline web app can be developed from a completed online only web app. Code was written by Adam Weidner during April 2015 as a senior design project.

This documentation assumes that the reader understands grails and the basic framework of online web apps, meaning model-view-controller. The documentation also assumes basic knowledge of JavaScript.

MyPhoto web app is designed for users to maintain an online photo album that is subdivided into topics. Topics can have many photos. While online, a user can create and save topics to the server, edit and delete topics that have been saved on the server. While offline, a user can create and save topics to the device, edit and delete topics saved to the device. After reacquiring an Internet connection and returning to online use, the user can submit the topics saved on the device to the server.

The myPhotos code is located at

https://bitbucket.org/grailsdirectedstudy/myphotos-complete

I encourage the reader to clone the repository, run the app, and browser the repository or project files as the code is described below. It uses grails 2.4.

Offline Development Techniques

There are two fundamental techniques for developing an offline web app, meaning an app that can function without a data connection. One technique is to use the same controllers and views for both the online and offline functioning of the app. The app's JavaScript code uses AJAX calls to provide the dynamic content from the server and LocalForage to provide content from the device storage. Another technique is to use separate controllers and views for the online and offline functions of the app. This technique can use backend framework calls (groovy code for Grails or php code for php frameworks) to generate the dynamic content for the online views and JavaScript calls to the device storage to generate the offline dynamic content.

Both techniques have advantages and disadvantages. Using the same views for online and offline has the advantage that it maintains the same views for online and offline use, but it has the disadvantage that a lot of JavaScript code is required and the developer cannot harness the power of the backend web framework. Using different controllers and views for online and offline has many advantages. Most of the JavaScript code is confined to the offline views, so there is a division of concern in the web app code structure. Less JavaScript code is required and the JavaScript logic is simpler. Another major advantage is the online version of the web app can be developed quickly and portions of the web app can be user tested early in the development phase. I believe that for developers new to JavaScript using different views and controllers for online and offline functionality is probably the easier and safer development route. The myPhotos web app demonstrates the development technique for using different controllers and views for the online and offline functions of the app.

Structure of MyPhotos Code

Domain

The online portion of the app uses a Model View Controller design pattern. Domain classes represent the model. MyPhotos has two domain classes, Topic and Photo. Topic has many Photos, and a Photo belongs to a Topic. Updates and deletes cascade from Topic to Photo. See

http://grails.github.io/grails-doc/latest/guide/GORM.html#oneToMany

Online Controllers

The online version of myPhotos has two controllers, TopicController and PhotoController. PhotoController has only delete and view actions. There are no views associated with PhotoController because photos are only displayed in the views for the topic. In many circumstances, Photo would have views associated with it.

TopicController has index, view, create, edit, confirmEdit, delete, createTopic, submitTopic, addPhotos and submitPhotos actions. The actions index, view, addPhotos, edit, and create primarily render their associated views. The actions submitPhoto and submitTopic interface with the Topic and Photo domain class and save the topic name and photo to the database. The action submitTopic is used by the create view to submit topic form data to the serve. Likewise submitPhoto action is used by the addPhoto view to submit form data, a list of photos, to the server. Both submitPhoto and submitTopic are actions for forms, and consequently the code that runs after the user clicks on the submit button.

Offline Controllers

The controllers associated with offline functioning of myPhoto are ManifestController and OTopicController. The role of the Manifest controller is to render the app cache manifest and has only one action, index. The manifest is an ASCII document specifying the web pages and assets that should be cached on the browser. There is more about the app cache and the manifest below.

OTopicController is the offline controller for TopicController. It is has actions index, view, addPhotos, edit, create and submitTopic. All except submitTopic just render their associated views. The exception, submitTopic, does not render a view; rather it use the domain objects Topic and Photo to save the topic and photos to the database server. The general conception of OTopicController is that it is the controller for the offline behavior, but that is not strictly true. Some of OTopic's views are rendered when the myPhoto app is operating online. For example the view views/OTopic/view uses the OTopicController submitTopic action which must be online to function. A more specific conception of OTopicController is that it interfaces with device storage rather than the server. Supporting the notion that OTopicController renders the offline views is that corresponding action views of OTopicController are the fallback for TopicController's views.

Views

Layouts and Templates

There are two layouts: first.gsp and main.gsp. The layouts are nearly identical. The only difference is that first.gsp has the manifest attribute in the html tag, as outlined below

```
<!DOCTYPE html>
<html manifest="/myphotos/manifest">
        <head> … </head>
        <body> …</body>
</html>
```

The manifest attribute causes the browser to read the manifest file rendered by the ManifestController and cache the assets and views listed in the manifest. Only one view, views/index.gsp, uses the first.gsp layout and invokes the manifest. The section on app cache and manifest explains more.

In addition, both layout link the assets and render the navigation bar for the views. The assets and navigation bar are locate at views/_assets.gsp and views/_navbar.gsp.

Home View: Available On and Offline

The view views/index.gsp is the home page for myPhotos. It is a static view without any dynamic content. As mention above it has a different layout from the rest of views. Only from the home page can the web app be cached.

Topic Views: Online views

The Topic views are addPhotos.gsp, create.gsp, edit.gsp, index.gsp, view.gsp. The view topic/index.gsp lists all the topics stored on the server and on the device. This view is an exception to all other views because it interacts with both the databases on the server and the storage on the device. All of the other topic views interact only with server database.

The view topic/create.gsp is primarily a form for users to create topics online and store them on the server. The view topic/addPhoto.gsp is a form for adding and storing photos to a topic stored on the server. The view/edit.gsp is a form for editing the topic name.

The views use g-tags to construct the forms. Grail's Groovy Server Page (GSP) tags. They are similar to Java Server Page (JSP) tags, except they are coded using groovy. You can learn more about g-tags in general at

http://grails.github.io/grails-doc/latest/guide/theWebLayer.html#tags

The specifics for the g-form and g-uploadForm tags is at http://grails.github.io/grails-doc/latest/ref/Tags/form.html

http://grails.github.io/grails-doc/latest/ref/Tags/uploadForm.html

Note that g-uploadForm tag specifies both the controller, topic, and action, submitTopic, to construct the form action. For example the topic/create view constructs

/myphotos/topic/submitTopic

OTopic Views: Offline Views

The OTopic views are addPhotos.gsp, create.gsp, edit.gsp, index.gsp, view.gsp, the same as the Topic views. In the manifest these views are fallbacks for the Topic views. They are very similar to the Topic views except at the bottom they link the JavaScript files required to generate the dynamic content from the device storage. We will study the JavaScript code associated with these views later.

Also different in these views is that submitting forms is not handled by the controller and the server side code. Rather they are handled by the browser and JavaScript code. So the code uses standard html form tag

```
<form onSubmit = "return false;" action="javascript:return false;">
```

The form onSubmit attributes specifies the JavaScript code to run when the user clicks the submit button

http://www.w3schools.com/jsref/event_onsubmit.asp

and bubbles up to the running JavaScript code. Because the JavaScript code for the page will use JQuery to define a click function for the form store button, we want to disable the onSubmit JavaScritpt code and prevent it from bubbling up by returning false.

The form action attribute specifies the URL that the form data should be posted to.

http://www.w3schools.com/tags/att_form_action.asp

We want to save the form data to the device and not post it to the server. Consequently, we disable the form action by sending it a javascript that returns false.

The OTopic views also differ from the topic views in that they use the asset plugin to link the offline JavaScript code at the bottom of the page. For example the OTopic/create view links

```
<asset:javascript src = "util.js" />
<asset:javascript src = "topic.js" />
<asset:javascript src = "create.js" />
```

Linking the JavaScript code their operations are discussed in the myPhotos Offline Operations section.

Manifest View
The manifest view renders the ASCII file that defines what views and assets the browser should cache. See the section describing the app cache and manifest for more details.

JavaScript Files

The JavaScript files for myPhotos web app are located in grails-app/assets/javascript. There are fundamentally two types of JavaScript files, JavaScript libraries and web app JavaScript files. Web app JavaScript files are code written by the web app developers specifically for the web app. The JavaScript libraries are bootstrap.js, jquery.js and localforage.js. The bootstrap.js file is used by Twitter Bootstrap for styling. The jquery.js is used by nearly all the other JavaScript code including the web app's JavaScript code; it is the de facto JavaScript framework. The localforage.js JavaScript file is used to store data on the device and consequently only used by the web app JavaScript code.

The web app's JavaScript files generally subdivide into two types. There are web app's JavaScript files that are used like libraries and JavaScript files that generate the content for the view or activate the view. I call the later "page activation" JavaScript files. The web app JavaScript files that are like libraries define JavaScript Objects or utility functions used by other JavaScript code. The file topic.js defines the Topic and Topics JavaScript Objects, and util.js defines a few utility functions used by the other JavaScript code. The web app activation JavaScript files are addPhotos.js, create.js, edit.js, view.js and viewTopics.js. Generally they are named after view that they provide the dynamic content for or activate. The files viewTopics.js is an exception to this naming convention. The JavaScript file viewTopics.js activates the view OTopic/index.gsp. The web app's JavaScript files are explained in more detail in the section about offline operations.

There are two web app JavaScript files that do not fit into the category of either a library or page activation JavaScript file. They are offline.js and online.js. They are used by the activation JavaScript files to identify the status of the web app, online or offline. They have only a single line of code that set the global JavaScript Boolean, online, to true or false. In the manifest, offline.js is the fallback for online.js. The JavaScript file online.js is linked to every web page in the _assets.gsp template. The template is render in the head for the layouts and uses the Grails asset plugin tag:

```
<asset:javascript src = "online.js" />
```

which links the JavaScript file and the browser will run it.

Application Cache and the Manifest

This section is only a brief introduction to the app cache and describes the specifics techniques used by the myPhotos app. Please read

http://diveintohtml5.info/offline.html

for a more general introduction, and

https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache for detail specification.

Manifest File Structure

The manifest file is located at

views/manifest/manifest.gsp

The manifest specifies the assets and webpages to cache on the device. The basic structure of the manifest is

```
CACHE MANIFEST
# Comment line specify either the date or version number of the manifest file

CACHE:
# A list of explicitly cached resources

FALLBACK:
# A list of URI pair specifying the URI to display when the first URI is not available

NETWORK:
# list of URI to retrieve only from the network
# Most app use an asterisk, meaning all other pages
*
```

The file must begin with the line "CACHE MANIFEST." Traditionally, immediately below the first line of the file is a comment line that specifies the version of the manifest. Any line in the manifest beginning with "#" is a comment line. It is important that the second line specifies the manifest version because the browser will not rebuild the app cache unless the manifest file has changed. This can be a problem during development because the developer could have changed or updated the web pages without changing the manifest. The behavior of the browser is to always retrieve the resources from the cache, so without re-caching the older resources will be displayed and not retrieved from the server. It is sufficient to change a comment line for the browser to recognize that the manifest is new and re-cache the resources. The tradition is to change the version number or date in the comment line just below the first line of the manifest to invoke the browser to re-cache.

The manifest has three primary sections: cache, fallback and network. The cache section is a list of URI or resources to explicitly cache. After reading the manifest file, the browser will attempt to cache all the pages and resources listed in the cache section. After caching, the browser will then only retrieve the resources from the app cache. It will not try to access the network for the resources.

The fallback section lists a pair of URI. The pair represent online and offline URI resources. The first URI of the pair represents the online version and will not be cached. The second URI represents the offline version (fallback for the first URI) to be used offline. It will be cached. The browser after reading the manifest will cache the offline URI (the second in the pair). If the browser cannot access the first URI because the browser is offline, it will retrieve from cache and render the second of the URI pair.

The network section specifies a list of URI resources to access only online. Most web apps just specify an asterisk meaning all other pages.

Specifying the App Cache

The two different techniques for developing offline web apps, using the same controllers and views or using different controllers and views for online and offline, imply different manifest specifications. Manifests for web apps using the same controllers and views will extensively use the manifest's cache section, listing every page that should function offline to be explicitly cached. Manifests for web apps that use different controllers and views for offline functionality extensively use the fallback section to invoke the pages that should be cached for offline use. Note that both techniques will use both the cache and fallback sections; only the emphasis is different. The myPhoto's manifest cache section lists the JavaScript and CSS files to cache for offline use in the cache section:

```
CACHE:
/myphotos/assets/jquery-2.1.3.min.js?compile=false
/myphotos/assets/bootstrap.min.js?compile=false
/myphotos/assets/bootstrap.min.css?compile=false
/myphotos/assets/style.css?compile=false
…
```

No views are listed in the cache section. Rather the myPhoto's manifest uses the fallback section to cache the actual pages to use offline

```
FALLBACK:

# Offline pages
/myphotos/topic/create /myphotos/OTopic/create
/myphotos/topic /myphotos/OTopic
/myphotos/topic/view /myphotos/OTopic/view
/myphotos/topic/addPhotos /myphotos/OTopic/addPhotos
/myphotos/topic/edit /myphotos/OTopic/edit
```

The fallback is also used for another reason. For example, the server would render "/myphotos" and "/mphotos/" the same, but the browser does not recognizes the two URLs as the same. Caching one version of the two URLs will not insure that the other URL will be render when the browser is pointed to it. A trick is required to insure that one version is cached and the other fallbacks to the cached version. The fallback section list the two as pairs, for example:

```
FALLBACK:
# Explicitly cache both "versions" of the homepage
/myphotos /myphotos/
```

Then if the browser is pointed to "/myphotos/" it will retrieve the view from cache. If while the browser is offline it is pointed to "/myphotos", it will fallback and retrieve "/myphotos/".

The same trick is used for all the online views in the myPhoto manifest:

```
FALLBACK:
…
```

```
# A copy of the above but with / added to the end,
# it might pay off to be paranoid and this is cheap
# to cache
/myphotos/topic/create/ /myphotos/OTopic/create
/myphotos/topic/ /myphotos/OTopic
/myphotos/topic/view/ /myphotos/OTopic/view
/myphotos/topic/addPhotos/ /myphotos/OTopic/addPhotos
/myphotos/topic/edit/ /myphotos/OTopic/edit
```

Finally notice the following line in the fallback section

```
FALLBACK:
...
# For telling if we are online or offline in the javascript
/myphotos/assets/online.js?compile=false /myphotos/assets/offline.js
```

As explained above, this uses the fallback to determine the online or offline status of the web app. Both JavaScript files are a single line of code that set a global Boolean to true or false. The rest of the JavaScript code can access the variable.

App Cache Specification Summary

Below is a summary of the procedure for writing the manifest file for an offline web app developed using different controllers and views for online and offline use.

1. Make a controller, ManifestController, that serves a single ASCII file. It can be rendered by the index action. Begin the file with "CACHE MANIFEST" and add a second comment line specifying the manifest version.

2. Add a line to the CACHE section of the manifest for each static resource that should be accessible offline. Static resources are JavaScript, CSS and image files.

3. For each online view that should have a corresponding offline view, specify a line in the FALLBACK section of the manifest. For example

```
/myphotos/topic/create /myphotos/OTopic/create
```

4. In the NETWORK section specify all the other files by specifying an asterisk.

5. Be sure to fallback on different versions of the same URL by specifying them in the FALLBACK section. For example

```
/myphotos /myphotos/
```

6. Use the fallback in the manifest file for web app to determine online or offline status by adding to the FALLBACK section the following line

```
FALLBACK:
...
myphotos/online.js myphotos/offline.js
```

and write the two short JavaScript files.

7. At least one page, probably the home the page, of your web app should specify the manifest attribute in the html tag. For example

```
<!DOCTYPE html>
<html manifest="/myphotos/manifest">
        <head> … </head>
        <body> …</body>
</html>
```

App Cache Gotchas

There a few gotchas to consider when using and developing the app cache.

1. The first gotchas to remember is that the browser will not re-cache unless the manifest has changed or has been removed from the browser. This can be troublesome when developing. You can view and remove app caches by pointing the browser to

chrome://appcache-internals/

You can also use your browser "incognito mode" during the development. Then closing the browser window will automatically delete the app cache from the browser. The next time you open the browser and point it to the website, it will be re-cache the files specified in the manifest.

2. The browser will cache any page that has the manifest attribute in its html tag weather or not the page is listed in the manifest. This effect has several implications. First, you need to be careful what pages have the manifest attribute. When developing using different controllers and views for online and offline functionality you probably do not want to cache the online pages because this will prevent the offline pages from being rendered. MyPhotos web app only has the home page with the manifest attribute. This can cause a problem for users because for the browser to see a new manifest the user must first visit the home page.

Even when developing offline apps using the same controller and views, not every page should be cached. For example a view specifying a parameter such as

/myphotos/topic/view/3

should not be cached, but the view /myphotos/view/topic should be cached, so that the JavaScript or backend code can generate the dynamic content. Specifying the difference can be challenging for developers using the same controllers and views and requires conditional construction of the html tag. See the Mushroom Mapper for an example on how to use Grails' Sitemesh to make conditional html tags.

3. Another related challenge is specifying parameters in the offline version even when different controller and views are used. For example the URL

/myphotos/OTopic/view/3

might mean to render a topic view 3 from the device storage. But the browser will not have the URL cached. Remember we cached only

/myphotos/OTopic/view

so the browser will not find it and will generate 404 error. We don't want the browser to cache the complete view; rather we want the browser to generate the view from the device storage. We need to interpret the "3" not as part of the URL but as a parameter. The trick is to use the hash, "#", in the URL, for example my photos uses

/myphotos/OTopic/view#id=3

The browser interprets the above URL as referencing

/myphotos/OTopic/view

Then the JavaScript code activating the page extracts the "3" from the URL address and retrieves the content from the device storage.

4. The browser caches all the files specified in the manifest or none of them. This means that if there is an error in the manifest, for example a file listed in the manifest that cannot be retrieved, then the browser will not cache any of the files. The error occurs rather silently; meaning the online behavior will not change and an error is only sent to the JavaScript console. You can access the JavaScript console in the browser's "developer's tools." I recommend keeping the JavaScript console open while you developing the manifest or JavaScript code.

Local Forage

HTML 5 has three api specifications for providing storage on the device, LocalStorage, IndexedBD and WebSQL. The difficulty for web developers is that only LocalStorage is a simple interface and implemented by all browsers, but the Local Storage implementation is typically limited to 5 Mbytes. It is typically used to save a content of a form so that the form data can be regenerated after the user has navigated away from the page and returned. IndexDB and WebSQL implementations typically have unlimited storage but have a more difficult "SQL like" interface. Also web browsers typically implement one or the other; only LocalStroage is implemented by most browsers. Consequently, web app developers have a challenge developing unlimited device storage across all browsers. Mozilla has come to rescue by developing a simple Local Storage like interface that uses either IndexDB and falls back to WebSQL or LocalStroage if the browser does not support IndexDB. It is called LocalForage.

https://github.com/mozilla/localForage

http://mozilla.github.io/localForage/

LocalForage is a Map or Hash Table like interface, meaning values are stored by keys. Although keys should be strings, values can be almost any type. LocalStorage makes the calls to the device storage asynchronously. Even though JavaScript is single threaded, I/O processing is implemented by the browser making a request to the I/O, after the I/O task is completed an event is place on the

JavaScript event queue which is later processed by the event handler or callback functions defined in the web app code. This has the advantage that calls to device storage do not slow down the user experience, but does require the use of callbacks or promises in the JavaScript code. LocalForage API uses either callbacks or promises design patterns. The callback design pattern is more fundamental to JavaScript and is what the myPhotos web app uses. In fact, must of the coding is dealing with defining and synchronizing the callbacks.

The basic LocalForage API functions with callbacks are

- getItem(key, successCallback)
- setItem(key, value, successCallback)
- removeItem(key, successCallback)

where successCallback is the callback function. Values can be of these types

- Array
- ArrayBuffer
- Blob
- Float32Array
- Float64Array
- Int8Array
- Int16Array
- Int32Array
- Number
- Object
- Uint8Array
- Uint8ClampedArray
- Uint16Array
- Uint32Array
- String

MyPhotos LocalForage Schema

Thought must be devoted to developing the storage schema for such a simple API. The storage schema should be hierarchical, meaning that the top most objects should store keys for lower level objects that they access. For example myPhotos web app uses the key "topics" to store an array of keys that map to each topic stored on the device. Each topic key maps to a Topic object which contains the topic name and an array of keys for each photo associated with the topic. The photo keys map to a blob for each photo.

This is a hierarchical schema

http://en.wikipedia.org/wiki/Hierarchical_database_model

The hierarchical schema has the advantage that the Topic object can be access without accessing all the photos blobs. This permits making a list of topics without loading all the photos into memory. Loading all the photos into memory could be time consuming and make for an unpleasant user experience. But in order to access a photo, the program must already have a reference to the Topic object.

For convince all keys should be Global Universal Identifies (GUID). True GUIDs are hard to generate, but myPhotos uses a fairly reliable JavaScript function to generate the GUID. The GUID generator is defined in util.js:

```
// Generate guid from stackoverflow.com/questions/105034/create-guid-uuid-in-javascript
function guid() {
   "use strict";
   function s4() {
      return Math.floor((1 + Math.random()) * 0x10000)
         .toString()
         .substring(1);
   }

   return s4() + s4() + "-" + s4() + "-" + s4() + "-" +
      s4() + "-" + s4() + s4() + s4();
}
```

A more reliable GUID generator would check if the key is already being used by the device storage and if so generate an alternative key. Because localforage calls are asynchronous, we cannot just use localforage.getItem() call on the proposed key to check if the key is already used.

Care must be used with the hierarchical schema. Saving a Topic object requires knowing all the photos keys before saving the Topic. Also, getting a photo requires getting the Topic object first. This is generally not a problem. Care must be made when removing a Topic. The Photo objects should be removed before the Topic object is removed from the store.

MyPhotos Offline Operations

This section reviews some of myPhotos' JavaScript code. In particular, it will trace through the code for:

1. Creating a topic offline
2. Viewing a topic offline
3. Viewing a list of all topics stored on the device or on the server
4. Submitting a topic stored on the device

The basic approach will be to study first the view and then the JavaScript code that activates the page. When necessary, the JavaScript code defining objects and library files will be described.

Creating a Topic Offline

When myPhotos is operating offline, clicking the "Create Topic" link in the navigation bar will send the user to the OTopic/create.gsp view. The OTopic/create.gsp is basically a form with a textbox for specifying the topic name and file upload field for adding photos. At the bottom of the view is a list of JavaScript files that are used by the view.

```
<asset:javascript src = "util.js" />
<asset:javascript src = "topic.js" />
<asset:javascript src = "create.js" />
```

The order of the above list is important. The JavaScript files util.js and topic.js are library and object JavaScript files. The JavaScript file create.js activates the page and uses the code in util.js and topic.js, so it should be listed last.

create.js
The JavaScript file create.js uses the JQuery function to abbreviate "document.ready(function ())".

```
$(function () {
// JavaScript code to activate the page
...
});
```

The above JavaScript file structure is another hint that create.js activates the view.

The JavaScript file create.js has only one function, to set the "click" button's handler for the "store" button, which stores the topic on the device. To store the topic and photos, the JavaScript code makes a new Topic object and retrieves the photo files from the form. It then cycles through the list of photo files and finally uses the Topic object save method to store the topic in LocalForage. Note that the argument for "topic.save" call is an anomalous function, in this case a callback function.

Generally, but not always when you see an anomalous function in a function call, you should think that it is a callback function and that it is the code that will execute after returning from the calling function. But, we cannot be sure when in time the callback will execute. The function of the JQuery.click() method is another example of a callback function. The anomalous function defined in JQuery.click() will execute as soon as the submit or store button is clicked. The callback design pattern is very common in JavaScript. It enables asynchronous behavior and user interface elements. You should learn to read code written using anomalous callback function naturally.

Returning to the topic.save() call, the anomalous function defined as the argument of the save() call

```
topic.save(function(id) {
        window.location.href = "/myphotos/OTopic/view#id=" + id;
    });
```

is a callback function which will be called and executed by Topic.save(). The callback function basically redirects the browser to view for the topic. Note that the callback function has an argument, id. The argument will be provided by the calling function, Topic.save. Think of the argument as a return value from the calling function. In this case, the argument, id, is the id for the view stored on the device. We can imagine that OTopic/view uses the id to retrieve the stored topic from LocalForage. We study that in the next view, views/OTopic/view.js. First, we study the JavaScript file topic.js.

topic.js

Note that the JavaScript file topic.js does not use the JQuery.(function()) design pattern, so topic.js is not a JavaScript activation file. Also note that the file begins with

```
Topic = function(name) {
…

}
```

This is a convention for defining a constructor for a JavaScript object. Consequently, we can assume that JavaScript file topic.js defines a JavaScript object, Topic.

Note that the object Topic has a name, a hash map or object for photos, and a key generated by the function call guid(). See above, in the app cache and manifest section, for the description of guid() function.

The addPhoto method is simple. It generates a GUID for the photo and then adds the photo to the topic's photos hash map.

save function in topic.js

The save method is little more complicated. First it generates an array of photo keys using Object.keys() function. See

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys

Then it calculates the number of calls it will make to LocalForage in order to synchronize the calls back from LocalForage. We will look into synchronizing the LocalForage calls in minute. For now, notice that the callback function used for all the following localforage.setItem calls is "sync."

```
var boundCallback = cb.bind(this, this.key);
var sync = barrier(callsRemaining, boundCallback, this);
```

This definition does two things. The first thing it does is adding a context, "this," to the original callback and defines the argument for the original callback. Note that the original callback is the anomalous function defined in the topic.save() called in the create view. It needs the "id," the key for the topic. The anomalous function in topic.save() does not really need the "this" context.

The second thing the definition for the sync function does is that it creates a single reference for the callbacks for all the localforage.setItem calls, which implies that all localforage.setItem will call back to the same instantiated function. You will understand why this is important when we study the barrier() function in util.js, next.

barrier() in util.js
The barrier function in util.js has one responsibility, count all the calls made to it and then call its callback after they are made.

```
// Make a barrier for synchronizing async callbacks

barrier = function(n, callback) {
      "use strict";
       return function() {
           n--;
           if (n === 0) {
                  callback();
           }
      };
};
```

Counting the calls is enabled by making a single reference to barrier, "sync," and using that reference in all the callbacks for all the localforage.setItem() calls. When barrier() is called it returns a function definition with the value of "n" set to the number of localforage.setItem calls to be synchronized. This is the definition for "sync," and the variable "n" is defined only once in the context for "sync." Each callback to "sync" from localforage.setItem decrements "n." After the final callback the intended original callback function is finally called. This is how localforage asynchronous calls are synchronized. Barrier does not synchronize the order of each localforage call, but guarantees that the original callback is not called until all the localforage calls have completed.

back to the save in topic.js
After the callback for the localforage calls has been defined, "sync," the Topic.save method first stores the serialized topic object. Then it cycles through all the photos and stores them on the device. The final localforage call is made by Topics.push() and it stores the key for the topic in the value for the key"topics" in LocalForage.

Topics in topic.js
The line in topic.js

```
Topics={};
```

defines and creates a Topics as a global object or hash map. In JavaScript there is no difference between a hash map and a JavaScript object. Topics is not created with a constructor. This is because only one Topics object is created for the page and Topics only has methods. Topics does not have any properties other than the methods. So, you can think of Topics as a static class in Java. The methods or Topics are getAll() which loads all the topics stored on the device into memory, push() which adds another topic to the list of keys in "topics," and delete() which removes a topic from localforage.

The method Topics.push(t, cb) has two arguments, "t" which is a reference to the Topic object being added to the list of topics in localforage, and "cb" which is a reference for a callback function. The method Topics.push() makes two localforage calls. The first is to retrieve the list of topic keys associated with the key "topics." This is done by localforage.getItem("topics", ...). The second argument of localforage.getItem is an anomalous callback function and is not the "sync" function. Whenever it completes it pushes the new topic key on to the list for "topics," and then makes the localforage.setItem() call. So the localforage.setItem call is synchronized with the localforage.getItem call. It will not be called until after the localforage.getItem call is completed. The callback for localforage.setItem call is also an anomalous function, but it just calls the callback function passed in through Topic.push(), which in this case is "sync."

Note that Topics.push() is coded to be very general. The callback function passed to Topics.push(), cb, can use the data returned from the localforage.setItem call. (In this case, the data returned from localforage.setItem is the array of topic keys.) But the callback function, cb, does not have to use the argument passed to it, and in JavaScript, functions need not be defined with arguments to be called with arguments. Note that JavaScript does not check if arguments types or number of arguements match across calls of the function with the function definition. In this case, the callback function is "sync" and it does not use the argument.

Viewing a Topic Offline

After the topic is created and stored on the device, the browser is redirected to the topic view. The topic view can also be reached by clicking on the topic in the list of topics in either topic/index.gsp or OTopic/index.gsp views.

The OTopic/view.gsp has divs for the topic name, id="topicName", button group, id="buttonGroup" which has buttons editing the topic, id="edLink", and deleting the topic, id="delink" and adding photos, id="addPhotoLink". Near the bottom of the file The OTopic/view.gsp also contains a div for the container of photos, id="photoContainer". At the bottom of the file OTopic/views.gsp links the JavaScript files:

```
<asset:javascript src = "util.js" />
<asset:javascript src = "topic.js" />
<asset:javascript src = "submitTopic.js" />
<asset:javascript src = "view.js" />
```

The JavaScript files util.js and topic.js JavaScript files have already been discussed; see above in the creating topics offline section. The JavaScript file submitTopic.js is only a function definition for the submit button handler. The submit button will appear if the web app is online. The function contains the AJAX call to submit the topic to server.

view.js

The JavaScript file view.js uses the JQuery design pattern $(function (){...}); so it will not run until after the page has loaded. Any JavaScript activation files using JQuery should use either

```
$(document.ready( function (){ ... } )
```

or the equivalent

```
$( function() { ... } )
```

http://www.w3schools.com/jquery/jquery_syntax.asp

The view.js file first defines the html template photoTemplate that will be used to add the photos to the offline topic view or more properly the view for a topic stored on the device. The photoTemplate also contains a button to delete or remove the photo from the topic. Also the submit html template submitButtonTemplate is defined. It will appear on the page if the web app is online.

The JavaScript code gets the id for topic to be displayed from getId() function defined in util.js. The function getId() uses the URL or routing for the page to get the id much like online version of code.

```
// Get the id of a topic from the url
getId = function() {
    "use strict";
    var uri = window.location.href;
    return uri.substring(uri.indexOf("id=") + 3);
};
```

As explained in the section above about creating a topic, the routing for offline and online differs because of restrictions of the app caching. The online version of the URL for a viewing a topic would be

/myphotos/topic/view/3

where 3 would be id for the topic stored on the server. For offline views of a topic, the URLs would have the form

/myphotos/OTopic/view#id=1234-1234-1234

where 1234-1234-1234 is the id for the topic stored on the device. We use the hash, #, instead of the slash, /, because of the appcache. The browser ignores the hash and everything fallowing the hash, so it will try to serve the page

/myphotos/OTopic/view

which has been cached. If the URL had been specified with the slash

/myphotos/OTopic/view/1234-1234-1234

the browser would try to retrieve /myphotos/OTopic/view/1234-1234-1234, but that page has not been cached, and we do want the page to be cached.

The "id=" part of the URL is not really required. We use it to make the routing a little safer and it mimics the CGI type URL, "?key1=value2&key2=value2", used to get resources through a script on the server.

After determining the id for the topic, JQuery is used to link the button. Buttons that link to offline OTopic actions can just use href attribute, for example

```
$("#addPhotoLink").attr("href", "/myphotos/OTopic/addPhotos#id=" + id);
```

Delete button and deleting a topic
Deleting a topic is more elaborate, and there is no view associated with the action, so JQuery is used to define the click event

```
$("#deLink").click(function() {
        Topics.delete(id,  function() {
              window.location.href = "/myphotos/topic";
        });
});
```

Recall that the Topics.delete function is defined in topic.js and uses a callback. The callback for Topics.delete call above is an anomalous function definition which just redirects to

/myphotos/topic

the online view that list both the topics stored both on the server and on the device. If the myPhotos app is offline then it will fall back to

/myphotos/OTopic

Topics.delete(..) must make several calls to LocalForage. Recall that LocalForage makes the calls asynchronously, also we do not want to redirect until after the topic has been deleted, so the redirect will need to be wrapped with the barrier, the barrier function defined in util.js.

You can study the details of deleting a topic from the storage on the device in Topics.delete.

Back to view.js
If the myPhotos app is online then the submit button can be displayed and we set the event handler for the submit button to the submitTopic function defined in submitTopic.js. We'll study the submit details later. It is primarily an AJAX to call to /myphotos/OTopic/submitTopic and then deletes the topic from LocalForage and redirects to the topic index view.

To display the topic, we will need to retrieve the topic from localforage and add the topic content to the view using JQuery. This will naturally involve retrieving the photos from localforage and adding them to the view.

The JavaScript code in view.js uses the function Topics.get(id, callback) defined in topic.js

```
Topics.get = function(id, cb) {
    "use strict";
     localforage.getItem(id, function(err, data) {
          cb(data);
      });
};
```

It just makes the localforage.getItem call and passes on the callback function, cb, that was given to it. The callback function is the anomalous function defined in the Topics.get call in view.js

```
Topics.get(id, function (topic) {
...
}
```

The Topics.get() function definition gives the callback function its argument, data, which is the topic object stored in localforage. So topic in the anomalous function is "data" from the localforage.getItem call.

Consider the synchronization that is required at this point. The JavaScript must retrieve the topic object before it can add the topic name to the topicName div. This only requires putting the JQuery call

```
$("#topicName").append(topic.name);
```

into the callback function. That is first thing that the anomalous function in Topic.get does. Note because the DOM element content only requires a single localforage.getItem call, synchronization only requires putting the JQuery element append in the callback and the code does not need to use the "barrier" function.

Next the code needs to retrieve multiple photo data from localforage, create a photoTemplate element for the DOM, fill/source the new element with the photo data, define the delete button click handler, and finally append the new element to the photoContainer div tag. Naturally, this will require iterating through all the photos in the topic with a for-loop.

```
for (var i = 0; i < topic.photoKeys.length; i++) {
        appendPhotoRow(i);
}
```

If we do not care about the order of the photos in the topic view then the only synchronization required is associating the correct delete handler with the photo, meaning that the delete button must delete the correct photo. This synchronization implies that the loop index, i, needs to be the same for the localforage.getItem and the definition for the delete handler. Recall that localforage calls happen asynchronously, so we cannot be guaranteed that it will return before the definition of the delete handler, but the correct association can be made if the index, i, is the same. This requires that the context remains the same. JavaScript has only one tool to define context and that is the function. Recall we have already seen context, sometimes called closure, preserving using the "barrier" function to define the "sync" function. In this case, we require the body of the loop to be a function, appendPhotoRow(i). So, now the index used in localforage.getItem call and the definition for the delete handler will be in the same closure or context, in other words the value of the will be same. To review, if synchronization only requires preserving the index value in a for-loop then the code can use this design pattern.

```
function f(index){...}; // function definition for loop body

for (var i=0; i<array.length; i++) { // loop iteration
.... // loop body with localforage calls and DOM element association
}
```

and astute JavaScript programmer might think, "why not use an anomalous function definition for the body of the loop

```
for (var i=0; i<array.length; i++) { // loop iteration
    function (i) {
       .... // function and loop body with localforage calls and DOM element association
    }
}
```

Then the code would look much more natural. Although this would work, we should NOT code the closure this way because a new function definition will be created for each iteration. Besides the context being store in memory, also all the function definitions will be stored in memory for each iteration.

appendPhotoRow(index)
The first task of the functional loop body, appendPhotoRow(index), is to retrieve the photo, the blob, associated with the key, photoKeys[index] using localforage.getItem(...). Sourcing the photo with the DOM element and appending the element to the DOM tree must occur after the photo is retrieved from localforage, so it must be in the callback for localforage.getItem(). Although the definition of the delete button only needs to be associated with the loop index, the definition of the event handler is defined in the callback for localforage.getItem() because we want the delete button to be in the same div element containing the sourced photo image. Consequently the rest of the appendPhotoRow() is the callback for localforage.getItem().

The callback for localforage.getItem() first creates the element that will be appended to the DOM, using

```
var elem = $(photoTemplate)
```

The JQuery call, $(photoTemplate), is on the string photoTemplate, so it converts the string to a JQuery element

http://jqfundamentals.com/chapter/jquery-basics

See the "Creating new elements" sections in the above document. Also you can study the official JQuery tutorial https://learn.jquery.com/using-jquery-core/jquery-object/

You can imagine that the string, photoTemplate, is a definition for a DOM element and that the $(photoTemplate) call instantiates the JQuery element from the string. It will become part of the DOM, a DOM element, after it is appended to the DOM.

After the JQuery element is created, the delete button handler can be defined. Note that deleting the photo requires localforage.removeItem() call with the photo key and removing the photo key from the photo key list stored with the topic requiring a localforage.setItem on the topic key. We want these localforage calls synchronized so the code creates a "sync" with n=2 in the "barrier" function.

All that remains is sourcing the photo bolb into the JQuery element, elem. The basic technique uses FileReader to read the bulb as a Data URL. Data URL scheme provides a way to provide data (like images) in-line in the web page.

http://en.wikipedia.org/wiki/Data_URI_scheme

The traditional technique for adding images to a web page is to link the image file in the html img tag:

http://www.w3schools.com/html/html_images.asp

But we cannot link the blob in localforage to an img tag directly, so we use FileReader

https://developer.mozilla.org/en-US/docs/Web/API/FileReader

The technique for using FileReader is to create the FileReader, reader, define the event handler for the onload event and then to read the bulb as a data URL.

http://www.javascripture.com/FileReader

Note that the onload event handler also appends the JQuery element, elem, to the photoContainer div.

Viewing a List of Topics Stored on the Device

A list of topics stored on the device can be viewed from the views/topic/index.gsp or views/OTopic/index.gsp. Note that /myphotos/OTopic is the fallback for /myphotos/topic in the manifest. Both views have a topicContianer div, which is the div for the list of topics stored on the device. The views/topic/index.gsp also has a serverTopicsHeading div which is for the list of topics store on the server. You'll notice that list of topics is formatted in rows for four topics with their name and an example photo. The topic list for topics stored on the server uses a custom grails tag

<mp:topicCollection>

to generate the list of topics. We'll study the custom tag after studying generating the list of topics stored on the device. Both views are activated with the JavaScript file, viewTopics.js.

viewTopics.js
The viewTopics.js JavaScript file use the $(function () {...}) JQuery call to load the JavaScript after the page is ready. The viewTopic.js JavaScript first defines html templates for the images and the online or offline jumbotrons. Then it gets all the topics with Topics.getAll(...) call. Topics.getAll() function in topics.js file will return an array of topic objects (not the array of topic keys) or an empty array if there are no topics stored in localforage. The synchronization required for Topics.getAll function is only that all the topic objects should be pushed on to the array before invoking the callback with topicObjects. I'll let you study the Topic.getAll code on your own.

The rest for the JavaScript code in viewTopics.js defines the callback for Topics.getAll(). First the callback checks topicObjects for content and decides on what jumbotron to append to the DOM at the topicContainer div. The final "else" is the case when there are topics stored on the device.

Consider the association that is required to list the topics. The imageTemplate contains a link to the topic view and an example image from the topic. These two must agree. The topic link can be created from the properties of the topic object, but we must load the example image coming from a localforage.getItem call into the correct imageTemplate element. The code does this by maintaining an array, imagesToLoadFromLocalforage, which is array of objects with properties of currentImage (the JQuery imageTemplate elment), row (the JQuery row div element), and photo (the photo key for localforage). The association of images from the localforage.getItem calls with the JQuery elements is maintained by this array. In JavaScript, if a variable is not declared with a var, the parser will seek the variable reference in the containing closures. This implies that if the localforage.getItem callbacks and the array imagesToLoadFromLocalforage are defined in the same closure then the callbacks can access the imagesToLoadFromLocalforage array.

The only synchronization required is that the JQuery row element should not be loaded on to the page until after all the JQuery imageTemplate elements are constructed and the images sourced. The JavaScript code in the viewsTopics.js actually waits until all the JQuery row elements have construction completed.

We can return to examining the Topics.getAll callback anomalous function in more detail now. After deciding what jumbotron to append to the topicContainer div, the code defines the "sync" function using the "barrier" function. Note that the barrier is after all the topic objects have been handled. Also, note that the callback for the barrier is just appending all the rows elements.

The array imagesToLoadFromLocalforage is declared just above the loops that it provides the body for. Must of the logic of the loop structure is for setting up rows for four topics in a row. I will let you figure that out on your own. The inner loop creates JQuery imageTemplate element, called currentImage, it appends the topic name and sets the link from the topic in topicObjects. If there are photo keys for the topic it fills the imageToLoadFromLocalforage.

Note that some of the topics in a row may not have images. These are not added to the imageToLoadFromLocalforage array. This will foul the synchronization because they will not generate localforage.get calls, so artificial "sync" calls are made for each topic that does not have an image.

Now that the array imageToLoadFromLocalforage is constructed, the JQuery imageTemplate elements are ready to be loaded with images. Naturally, this will be done in a for-loop and passing the loop index to the function for the loop-body. We will use the design pattern of defining the body of the loop in a function, loadImage.

The loop body function, loadImage, uses FileReader to source the bulb by reading it as data URL. Note how the loop index passed to loadImage is used to source the proper image into the imageTemplate JQurey element and then to append the imageTemplate to the proper row element.

mp:topicCollection
The definition of the mp:topicCollection tag is in taglib/myphotos/TopicCollectionTagLib.groovy. You can learn about g-tags at

http://grails.github.io/grails-doc/latest/guide/theWebLayer.html#tags

and about creating your own tag libraries at

http://grails.github.io/grails-doc/latest/guide/theWebLayer.html#taglibs

In essence, creating a custom tag generates a string that is outputted to the page. You can use any groovy code and access the domain classes to generate the string. Your tag labels are actions in the tag lib class. TopicCollection has only one method to use as a tag, topicCollection. The method topicCollection calls createCollection to actually build the string using StringBuilder. StringBuilder overloads the left shift operator with append.

http://docs.groovy-lang.org/docs/next/html/groovy-jdk/java/lang/StringBuilder.html

And groovy uses triple quote, """, literals to include multiple lines in the literal defining the string. These two syntax makes it possible to construct human readable strings.

The helper method createCollection gets the list of topics from the Topic domain and proceeds to construct html code for the topic. Note how the img tag is sourced.

```
<img src="${photoSrc}" />
```

The Grails complier will interpret the syntax "${photoSrc}" to output the value of the variable photoSrc as string. The variable photoSrc is defined in the preceding lines as the routing to the controller and action appended with the photo id.

Submitting a Topic from LocalForage

The general flow for submitting a topic from the device storage to the server is to retrieve the topic and photos from localforage, create the AJAX call. The server then receives the data from the AJAX call and loads them into the database. The JavaScript file submitTopic.js retrieves the topic data from local forage and formats the data for the AJAX call. The data in the AJAX call is sent to the OTopicController submitTopic action which constructs the Topic and Photo domain objects and save them to the database.

submitTopic.js
The JavaScript file submitTopic.js defines the AJAX call that submits a topic and all it photos stored on the device to the server. The function defined in submitTopic.js is the event handler in the views/OTopicController/view.gsp submit button.

The AJAX call is created by JQuery.ajax method.

https://learn.jquery.com/ajax/

http://api.jquery.com/jquery.ajax/

See the examples at the bottom of the page. Because the AJAX call will load data into the database on the server it should use the POST method. The data uploaded into server should be constructed using FormData

https://developer.mozilla.org/en-US/docs/Web/API/FormData

Topics.get function will make the localforage.getItem call to get the topic topic name and list of photo keys. The second argument for Topics.get is the callback that is passed to the localforage call. The rest of the code in submitTopic.js defines this callback. The synchronization of retrieving the topic object from localforage is handled naturally. In this code, only one AJAX call is made to upload the topic and all the photos. The code will need to get the photos from localforage, so we should think about the required synchronization getting all the photos. The "sync" function is defined by the "barrier" function with "n" set to the number of photos. Also notice that the "sync" function makes the actual AJAX call to /myphotos/OTopic/submitTopic. To add the photos to the FormData we use the design pattern of loop body function to pass the loop index to the localforage.getItem call backs.

The event handler for AJAX "done" calls Topics.delete which removes both the topic and photos from localforage. It then redirects to the topic list.

OTopicController/submitTopic
There is not much to this groovy code. Note how the params object

http://grails.github.io/grails-doc/latest/ref/Controllers/params.html

is used to get the topic data and the request object to get at the photos.

http://grails.github.io/grails-doc/latest/guide/theWebLayer.html#uploadingFiles

Finally it creates the response to the AJAX and returns "success" as a JSON.

Summary Using Asynchronous Callbacks
When coding asynchronous calls, like LocalForage calls, there are several factors to consider in the code design:

- Dependences using the values generated by asynchronous call (localforage call)
- Associations across pages elements using the values generated by asynchronous calls

- Synchronization of the asynchronous calls and code for activating the page

Dependences of the asynchronous data occur when the data generated or returned from the asynchronous calls is used by code to calculate values for other variables. Associations occur when the data from the asynchronous calls is used in multiple page elements and the appropriate asynchronous data needs to be associated with the page elements. Finally, synchronization is required when timing of events or actions on the page need to occur only after the asynchronous calls have finished. These three concerns are related and frequently occur together. We'll consider each separately in more detail in the following sections.

When you start to code localforage calls, you should design the program before coding. Consider the three aspects: Dependence, Synchronization, and Association. If you make sure that these requirements of the code are satisfied then your program will be fairly bug free.

Asynchronous Calls in the Abstract

Before studying the three aspects of asynchronous calls, we should first study how asynchronous calls are handled by JavaScript.

Mozilla's documentation on the Concurrency Model and Even Loop

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

explains how the JavaScript interrupter executes the code. There are three components, the Stack, the Heap and the Queue. The Stack is used for processing function calls, executing and returning from the function calls. You should be familiar with interrupters pushing the context of the function call on to a stack for each function call. Returning from the function call, the context of the function is popped off the Stack, the return values are passed onto the context of the calling function now on top of the Stack. There is a pointer in this new context which indicates where to continue the execution.

The Heap is the memory allocated for storing all the variables and objects values and function definitions. A Symbol table is used to map variable names to memory locations.

The Queue is the data structure for handling asynchronous behavior. JavaScript does not have multiple threads; rather the JavaScript interrupter puts messages on the Queue and the messages are handled as first come first served. Messages can be events from the user interface or returns from the localforage calls. Messages must have an associated function, called the listener, which will run when the message reaches the front of the Queue. For UI events, the event handlers are the listeners, and the callbacks defined in the localforage calls are the listeners for the messages generated by the localforage calls.

The listener function associated with the message will be place on the stack when it reaches the front of the Queue. It will most likely call more functions which will be pushed onto the stack and executed. Finally the Stack empties and the next message is removed from the front of the Queue and listener placed on the Stack. This techniques guarantee that each message is run to completion before the next message is processed.

When thinking about how the asynchronous localforage calls are processed, always assume that there are already messages in the Queue. Also, that the message from the localforage call will not be placed in the Queue until after the values have been retrieved from localforage, so there is ample time for more messages to be placed in the Queue. Think that many messages can be before the localforage in the Queue. In addition, consider the current context or closure that made the localforage calls, it must probably have functions still in the Stack; they will be processed before the localforage listener (callback) is processed. In other words, the code following the localforage call will execute immediately and will not have the benefits of the values retrieved from localforage. The interpreter will push any function calls on the stack made by the current closure and process these function calls before getting to the message and listener generated by the localforage call.

If you want to play around with the Queue and experiment with its timing, there is only one JavaScript function, setTimeout, that you can code directly and place messages in the Queue

https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setTimeout

It will place the function, the first argument of setTimeout, in the Queue after the specified milliseconds.

Complying to Dependences
Dependences of objects and variable values are dealt with by placing the dependent object or variable definition in the asynchronous callback. For example in myPhotos, a topic object must be retrieved from localforage before it call be used. The function Topics.get() is responsible for getting the topic, given an id, and passing the topic object on to the view that will use the topic object. The definition for Topics.get() in topic.js is very simple; it makes the localforage call and passes second argument, a function, to the localforage's callback. The JavaScript file view.js that activates the view OTopic/view.gsp uses Topics.get() and defines the function to be passed to Topics.get() in the anomalous function definition. The anomalous function constructs the topic view using the topic object retrieved from localforage.

A slightly more complex example is Topic.push() in topic.js. The function Topics.push() is used when a topic object is saved to localforage. Saving a topic requires adding the topic key, id, to the array of topic keys mapped to the key "topics." So, a dependency is that the array of topics keys must first be retrieved before the topic key can be added to the array and then saved in localforage. The definition for Topics.push() handles this dependency by putting the localforage.setItem("topics",…), which saves the array of topic keys to localforage, in the callback for localforage.getItem("topics," …), which retrieves the array of topic keys.

The design pattern for dependences in both examples is to place the dependent variable definitions in the callback. It is a simple design pattern but it does scatter the code, and the code gets more scattered when asynchronous calls depend on asynchronous calls. In this case, the callbacks will become nested.

Design Pattern for Complying with Dependences

var firstIn = … // values for first asynchronous call

```
asynchronous.firstCall( firstIn, function (err, firstOut) {
        // code dependent on firstOut
         var someVar = f(firstOut);

        // and/or possibly make a dependent asynchronous call
       asynchronous.dependentCall(someVar, function(err, dependentOut)
             // Callback for dependent asynchronous call

       });
});
```

Asynchronous Associations

Asynchronous association is required when values retrieved from localforage are used to construct DOM elements and the DOM elements have an association with each other. An example of an association in myPhotos is the delete button for a photo must be correctly associated with the photo. Associations are complicated when the data for DOM elements comes from localforage because the localforage calls are asynchronous. Fundamentally, the association is made in the same context, meaning within the same function, which would be the localforage callback. In the case of associating delete button with photos in a topic view, view.js, this association needs to be made with each photo, naturally this implies a for-loop. The body of the for-loop is a function, appendPhotoRow, to preserve the context. Also the loop index is passed to the function. Notice how this same index is used in the topic.photoKeys to retrieve the photo for use in FileReader and to define the item to remove from localforage for the delete button's event handler.

A more complex example of association is constructing the list of topics in views/topic/index.gsp and views/topic/index.gsp. Constructing the list of topics requires associating the imageTemplate with a row, associating a topic photo with the imageTemplate, and associating the topic view link with the imageTemplate. This is handled in viewTopics.js by creating an array, imagesToLoadFromLocalforage. The array stores references to the current imageTemplate (called currentImage for short), row and the localforage key for the photo. During the array construction, the link to the topic view is inserted into the image Template, currentImage. So the localforage callbacks only need to associate (meaning in this case source) the appropriate photo in the imageTemplate and associate (meaning in this case append) the imageTemplate to the row. Again these associations are achieved by passing the for-loop index to the loop body function, loadImages. Note how the loop body function, loadImages, use the index in imagesToLoadFromLocalforage array to make the associations.

The design pattern for association in both examples is to construct an array to use for the associations and then to use a loop body function to make the localforage calls. The array is constructed in the same closure of the loop, so the loop body function has access to the array. Also the array is constructed so that it elements will correspond with the loop iterations. The loop body function is passed the loop index, so that the localforage callback accesses the appropriate element in association array.

Design Pattern for Asynchronous Associations

```
// array for the association data
var associations = []; // array or hash map

// loop for defining association data
for(var i; i < numOfAsscoci; i++){
    associations[i] = f(i); // define the assciation
}

// loop body Function for making the association
var loopBodyFunction = function(index) {

    // possible asynchronous call with association data
    asynchronous.calling(association[index], function(err, callingOut){
        // use association data with page element
        var element = $photo(template);
        useAssciation(element, association[index]);
    });
};

// loop to make the associations
for(var i; i < numOfAsscoci; i++){
    loopBodyFunction(i);
}
```

Synchronization

Synchronization is required when a page event should not occur until after one or more asynchronous calls. An example of synchronization is deleting a photo from the topic. Because the refreshed view is a confirmation to the user that the photo has been deleted, the refreshed view should not occur until after the photo has been removed from localforage. The event handler for the photo delete button is defined in the callback for localforage.getItem() in the appendPhotoRow function in view.js. Two localforage calls need to be made. One localforage call removes the photo from localforage and the second localforage call needs to update the topic in localforage. The page action that should occur after the photo is deleted is one line, window.location.reload(); It is defined in the callback for the "barrier" function. We have already explained how the "barrier" function defines a new function that counts the localforage and then calls the callback. Passing the "sync" function defined by "barrier" to each localforage call insures that the refresh is synchronized only after the photo delete is complete.

Deleting a topic from localforage requires a similar synchronization. The user should be redirected to the topic list view after topic has been deleted entirely from localforage. But to delete a topic requires removing all the photos from localforage, removing the topic from localforage and updating the list of topics in localforage, in other words, two more than the number photos. The function Topics.delete() in topics.js creates the "sync" function using the "barrier" function with this number. Then the localforage calls are made with "sync" as the callback. Note that the sync is used in more than just the localforage calls in the loop body.

The design pattern for synchronization in both examples is to count how many asynchronous calls need to be synced together, then define a single function that counts the asynchronous calls and after all the calls have been made call the ultimate callback passed to it.

Events that need to occur after asynchronous calls can generate other action then redirects. For example, suppose that a spinner is displayed while constructing the list of topics. After all the topics have been added to the list and rendered on the page, the JavaScript for page should hide the spinner. Hiding the spinner would be in the ultimate callback.

Instead of wrapping the ultimate callback in a "sync" function, synchronization can be made by using a global variable to register the localforage calls and to have an if statement in the callback checking the global variable in order to determine when to make the ultimate callback. This technique is not as safe because the global variable can be unknowingly overwritten. Also the technique of using a global variable is only natural when all the localforage calls are in a loop. Defining a function that contains counting variable in its closure and decides when to make the ultimate callback is safer and easily generalizes to more than localforage calls in a loop.

Design Pattern for Synchronization

```
// Barrier function for synchronous function
barrier = function(n, callback){
    return function() {
        n--;
         if ( n === 0) {
              callback();
         }
    };
};


// Count the number of asynchronous calls
var numAsyncCalls = countCalls();

var ultimateCallback = function(){
      // do something on the page
 };

var sync = barrier(numAsyncCalls, ultimateCallback);

// now make the asynchronous calls
asynchronous.Calling(callingIn, sync)
asynchronous.Calling(callingIn, sync)
…
```

Promises the New Hope

Promises are an alternative syntax to callbacks. LocalForage can use the promise syntax http://mozilla.github.io/localForage/#localforage

A promise allows for defining two callbacks, one for resolve and the other for reject.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

The promise decides which callback to make. This helps separate out error handling. The callbacks can also be added to the promise after it has been constructed using the promise's "then" method. This technique is frequently used and is the technique illustrated in the LocalForage documentation.

Some additional references motivating and describing the use of promises:

https://www.promisejs.org/

http://www.html5rocks.com/en/tutorials/es6/promises/

For the most part promises are only a syntax difference that might be easier to read and use. They do add a new feature, checking the status of a promise: pending, fulfilled, rejected, and settled. But the need to check the status is not common because all code really needs is when the asynchronous call completes.

Offline Web App Architecture

Besides illustrating some of the techniques and tools for developing offline web apps, this documentation of myPhotos also illustrates a code architecture for offline web apps. The foundation of the architecture is the "separation of concerns." Separation of concerns for a web app that has two data stores, one on the server and the other on the device, implies two sets of model-view-controller. One set of model-view-controller interacts with the database on the server. The other set of model-view-controller interacts with the storage on the device. Making this separation allows the developer to access all the features of a web framework for developing the model-view-controller that interacts with the database on the server. It also confines most of the JavaScript development to specific views.

Unfortunately, there are no true model-view-controller web frameworks that interact with storage on the device. Also, the controllers that are understood for current frameworks do not run on the device. They run on the server side. The consequence is that the controller functions are in the JavaScript files, and they can be easily mixed or confused with model functions. In addition, current technology does not provide web developers with a framework for domain models. Web developers must code at a low level, directly coding LocalForage calls. Finally, the API for storing data on the device is primitive and for the most part restricts web developers to a flat or hierarchical schema.

Nevertheless, I believe even with the current technology the web developer can adhere to some of the model-view-controller paradigm for interacting with the device storage. The myPhotos web app hints at the some of the design patterns that can help adhere to model-view-controller design.

1. JavaScript objects can and should represent the domain model. This implies that all the localforage calls should be coded within in the JavaScript objects. An example in myPhotos is the Topic and Topics objects in topic.js. In fact, most of the code in the JavaScript objects is dealing with localforage calls.

2. The JavaScript files that activate the page represent the controller. There should be one activation JavaScript file for each view that interacts with the device storage.

The two standards above do not really answer all the requirements for the separation of concerns. Most obvious is that the JavaScript file activating the page constructs the page, interacts with the domain objects and manages synchronization. More abstraction is required to separate concerns, and perhaps some mixing of concerns is unavoidable. For example even for the model-view-controller that interacts with the database on the server uses custom tags to construct the views. But this is a better solution then using JavaScript to construct the view in the model-view-controller interacting with storage on the device. The primary coding hurtle is managing the asynchronous calls. Modern server models can hide most of the synchronization from the web developer working with the database on the server. Managing the data stored on the device, the web developer has only callbacks to manage asynchronous storage calls.

Async.js

There are a few JavaScript libraries that assist with asynchronous calls.

https://www.airpair.com/javascript/posts/which-async-javascript-libraries-should-i-use

We are interested in JavaScript libraries that can run the browser and handle both reading and writing. The libraries Async and Step are two libraries that can satisfy localforage call requirements.

https://github.com/caolan/async

https://github.com/creationix/step

Async is the more powerful than Step. The blog below illustrate simple examples.

https://blog.engineyard.com/2015/taming-asynchronous-javascript-with-async

I suspect that Async function calls cannot be nested. This might hinder Async's use.

More Thoughts