ngPhotos (Grails + AngularJS)

Robert Pastel, 9/12/2015, 9/13/2015, 9/14/2015, 9/15/2015, 10/18/2015, 12/24/2015, 3/19/2016
Stephen Radachy, 9/30/2015, 11/4/15, 11/11/15, 11/15/15

The goal of the ngPhotos project is to demonstrate the addition of offline functionality given an online Grails app. It is not intended as an example of a "complete" app. This is a modification of the myPhotos web app. myPhotos which was written by Adam Weidner during the Spring 2015 semester. Adam's project files (with some code corrections by Stephen Radachy) are located here:

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

My document describing Adam's code is at

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

It is called “Documentation.docx”

Adam developed his offline code using primarily jQuery. A major conclusion of Adam's project is that the offline code needed an appropriate JavaScript framework. Stephen Radachy's project was to determine the appropriateness of AngularJS as a framework for developing offline web applications.

https://angularjs.org/

Both projects suppose that a web app is first developed online using Grails for the web framework.

https://grails.org/

Although developing an offline web app directly without developing the online version first may be more efficient, developing the online web app first has several advantages for students at Michigan Tech. The web app will need a database, and the Grails framework using the Model View Controller (MVC) design pattern has an excellent Domain framework for the Model. The primary reason for first developing an online version is that most of our students are not familiar with web technology. Grails, a groovy based framework, is an excellent framework for introducing web technology. Minimum knowledge of JavaScript is required to develop a complete online web app. In addition, Grails is well supported, and has good plugins for authentication and any other needs for a web application. From a pedagogical perspective, students can work on the same project, initially introduced to web development learning groovy and later learn JavaScript to make the web app work offline.

There are disadvantages to first developing the online version. Typically "online first" development requires duplicating the views for offline use. Maintaining consistence between the online and offline views strains maintainability. Developing "online first" sacrifices some maintainability of the code for the pedagogical benefit to students learning web technology.

Online only myPhotos

The completed online only version of myPhotos is at BitBucket:

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

MyPhotos enable users to catalogue photos into "topics." Topics can be created in a database and photos can be uploaded to the database and assigned to topics.

This is a minimum web app. The following description of the code for the online version, assumes that the reader is acquainted with the Grails framework and web development. If not, please see my HCI course lecture notes and programming assignments

http://cs4760.csl.mtu.edu/

MyPhotos has only two domain classes, Topic.groovy and Photo.groovy, in the directory

grails-app/domain/myphotos/

Topic has many photos and photos belong to topics.

There are also only two controllers, TopicController.groovy and PhotoController.groovy, in the directory

grails-app/controllers/myphotos/

TopicController has actions for the CRUD database interface, creating topics and adding photos to topics. PhotoController only has actions for viewing a photo and deleting a photo. Viewing a photo requires reading from the database and rendering the data for the view.

The views for the online myPhotos are in

grails-app/views/

There are only topic views: addPhotos.gsp, create.gsp, edit.gsp, index.gsp and view.gsp. The views only use the grails-app/views/layouts/main.gsp.

Bootstrap is used for styling and JavaScript for the web app is only used by Bootstrap. The JavaScript libraries are in the directory,

grails-app/assets/javascript/

The configurations files are in the directory

grails-app/conf/

The web app uses primarily the default files provided by Grails. A H2 database in memory is specified in DataSource.groovy. MyPhotos only uses the default plugins including the asset-pipline plugin, which is standard for Grails 2.4. Note that the bootstrap plugin is not included in the BuildConfig.groovy file. Bootstrap is linked to the web-app using the asset-pipeline plugin.

The Bootstrap.groovy file builds the database by searching for photos on the development machine. Note that the code only works on Linux machines.

An important feature of the online myPhotos code is that it is all contained in the directory.

grails-app/

There is not a web-app/ directory. Studying the offline code will reveal that all the offline code is contained in the directory.

web-app/

This naturally separates the online and offline code. Note that the web-app directory is used for any resources that are not accessed by Grails. Resources in web-app directory can be accessed directly by the browser. The URLs for resources in the web-app directory begin with the app name and do not specify "web-app."

ngPhotos (offline)

AngularJS

The offline ngPhotos code uses AngularJS framework.

https://angularjs.org/

AngularJS is a Single Page Application (SPA) framework. Different views are presented in the same basic page. The different views are templates, typically called "partial views" because they only describe part of the view.

AngularJS uses a variant of the MVC design pattern that uses a View-Model (V-M) design pattern instead of a Model. The design pattern plays down the role of the model, and for lack of a better term we will call the design pattern VV-MC. The V-M is a stripped down model for the view, and its sole role is to provide the data interface between view and the controller. In other words, V-M does not provide an interface to the data store, like the Model in the MVC design pattern. In AngularJS the V-M is only a JavaScript object in memory called $scope. The JavaScript (meaning logic) specific to the view should be in the controller. The controller can control the data for the view by manipulating $scope. The view attaches the data using ng-model tags (or directives, more about directives later) and can access the data value using expressions ({{photo.id}} for example). Besides separating concerns and providing glue between the view and controller, VV-MC is agnostic to the data store. For small web apps, like ngPhotos, the controller can interface with the data store directly. Larger web app should have a domain object that could be an AngularJS service.

A very important feature of AngularJS for ngPhotos is automatic view update provided by the data-binding. Whenever $scope changes, AngularJS will automatically update the view to the new values in $scope. We will see that this automatic updating will trivialize the use of promises.

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

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

A popular feature of AngularJS is directives. AngularJS documentation for directives calls them marks in the DOM (http://www.w3.org/DOM/) element. AngularJS comes with useful directives and permits the coder to define their own html tags. The effect of directives is to make html more expressive while maintaining the declarative nature of the HTML documents. The end result is two give the web designers more power and control of the views.

AngularJS is designed to be modular. Modality is provided by the AngularJS Module and dependency injection (DI).

https://docs.angularjs.org/guide/module

https://docs.angularjs.org/guide/di

Dependency injection is more explicit then in Grails. The second parameter of Factories for Modules or Controllers is an array of strings listing all the dependencies that should be injected into the controller or module. Factories are JS objects that build the other JS objects. You can think of them as substitutes for object constructors.

Another important feature of AngularJS is routing configuration. The routing configuration associates controllers with views. Because AngularJS is a framework for SPA, the routing is specified by everything after the "#" (hash) in the URL. It will automatically load the controller with the view. The controller has its own $scope variable but can access variables that need to be global through $rootScope. Although $rootScope is considered to be “global” within AngularJS, controllers still have access to raw JavaScript global variables as well.

The above description of AngularJS is only intended to motivate it use for offline web app development. To learn more about AngularJS, I recommend the "Fundamentals in 60 Minutes" video

http://www.youtube.com/watch?v=i9MHigUZKEM

The "Official AngularJS Tutorial"

https://docs.angularjs.org/tutorial/index

and later, the "Developer Guide"

https://docs.angularjs.org/guide

and "API Reference"

https://docs.angularjs.org/api

Another great resource that Stephen used to learn AngularJS (which may be more appropriate for students who are less familiar with JavaScript and or MVC frameworks) is a couple of courses through a website called CodeSchool:

https://www.codeschool.com/courses/shaping-up-with-angular-js - This is free

https://www.codeschool.com/courses/staying-sharp-with-angular-js - a paid subscription is required, but a free-trial month is available.

The description of the offline code assumes that you have studied the fundamentals of AngularJS.

AngularJS Modules

In addition to the core AngularJS, the offline ngPhotos, also uses two AngularJS modules, angular-base64-upload and angular-indexedDB.

https://github.com/adonespitogo/angular-base64-upload

https://github.com/bramski/angular-indexedDB

The module angular-base64-upload is for encoding file inputs to base64. Encoding files into base64 strings works around an iOS indexedDB implementation bug.

https://gist.github.com/nolanlawson/08eb857c6b17a30c1b26

The angular-base64-upload module provides a directive to the input tag which automatically encodes single or multiple files input. It also makes a JS object of the file that can be easily stored into indexedDB. Our experience with angular-base64-upload is that it is stable and reliable.

The module angular-indexedDB simplifies working with the indexedDB API. You can use it to configure IndexedDB data stores and make an AngularJS Provider. It also provides convenient data store operations. Our experience with angular-indexedDB module is that it is stable and reliable

IndexedDB API

An offline web app intended to collect data to later upload to a database needs a temporary data store on the device. There are three HTML5 API that can provide data store on the device: Web Storage (LocalStorage), WebSQL and IndexedDB.

http://www.html5rocks.com/en/features/storage

https://w3c.github.io/webstorage/#the-localstorage-attribute

http://www.w3.org/TR/webdatabase/

http://www.w3.org/TR/IndexedDB/

LocalStorage is an object store intended for small storage. It is not appropriate for storing files. WebSQL is a SQL storage that is being phased out. It can handle large storage but is being faded out probably because it is too awkward to use. IndexedDB is an object store and is the only choice for myPhotos data store. IndexedDB is a very large API and is probably too difficult to use directly. We have experimented with LocalForage as interface for IndexedDB.

https://github.com/mozilla/localForage

LocalForage provides browser compatibility by falling back to Local Storage. Because of the bugs in iOS implementation bugs for IndexedDB, we have found that it falls back too much for our needs. Consequently, we use the angular-indexedDB module that only interfaces to indexedDB.

Because indexedDB is an object store, data is stored as a "key-value" pair. The "key" is the primary access to the "value" which is stored as an "object." The "object" is frequently modeled as a JSON. But as the name of the API implies, programmers can also create additional access via an "index." An index allows looking up records in an object store using properties of the values in the object stores records. The object store provides convince to the JavaScript programmer and creating the appropriate index makes data retrieval efficient.

JQuery

AngularJS is built on JQuery Lite.

http://www.informit.com/articles/article.aspx?p=2271482&seqNum=10

https://docs.angularjs.org/api/ng/function/angular.element

Online to Offline Development

The general process of adding offline capabilities to an online Grails web app require slight modification to the Grails app and adding the offline JavaScript and views. This section is intended to give a brief temporal outline of the process.

Below outlines the changes needed within each section of the ngPhotos project

Changes in grails-app/

1. Adding the Manifest

The online code must provide the manifest.

http://diveintohtml5.info/offline.html

for a more general introduction, and

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

The manifest is a text document specifying which resources the browser should cache. (See detail description of the "Application Cache and the Manifest" below; this is only a brief description of adding the manifest to the web.) A Grails controller, called ManifestController, is added to grails-app/controller/. ManifestController only renders the index action by setting the content type to "text/cache-manifest" The grails-app/views/manifest/index.gsp is a text file specify the cache manifest.

The browser learns of the location of the manifest by the manifest attribute of the html tag. Any web page that has the manifest attribute in the html tag will be cached weather or not it is specified in the manifest. (See the detail description in the "Application Cache and the Manifest" section below.) Not every page should be cached to the device, especially pages that are dynamically generated should not be cached.

Because the html-tag is in the Grails layout, adding the attributes implies modifying the current layout or adding an additional layout to the Grails app. It is possible to conditional add the manifest attribute to html tag depending on the body of the view, but it is a rather advance programming technique that requires detail knowledge of Grails layout manager. An easier approach is to add a second layout, **grails-app/views/layout/topic.gsp**, which has the manifest attribute in the html tag. Then pages that should have the manifest attribute use manifest layout and pages that should not be cached will use the original layout, grails-app/views/layout/main.gsp. In the case of the ngPhotos app only topic page uses the layout with the manifest attribute.

2. Online/Offline Detection

The web app will need to determine if it is on or offline. It is inappropriate to determine the online status by querying the server because this slows down and even breaks the app. A better approach is to use the browser's capability to know whether or not to fallback to an offline resource. (See the detail description in the "Application Cache and the Manifest" below.) Two JavaScript files are used two determine the online/offline status, offline.js and online.js. 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 in the layouts.

3. Add ng-view Tag

The offline views templated through the topic/index.gsp view. This is the location of templated views are specified by the <div ng-view> tags.

4. Hide and Show

Because the offline views are templated through the ngPhotos topic index page, grails-app/views/topic/index.gsp, we need to modify the layout, grails-app/views/layouts/topic.gsp. A short JQuery script is added to the bottom of the body which hides or shows contents based on a css classes, ".grails" or ".angular".

5. Upload a Topic

We add a new action to the TopicController, submitOfflineTopic(). It is the target of an AJAX call made by AngularJS in the viewTopic controller.

In the topic.js viewController the submitTopic, action makes an AJAX call to a specialized action within the TopicController called “submitOfflineTopic”. The TopicController action creates the online topic and photos by searching through the HTTP request parameters. The topic is created and the photos are added in by decoding the base64 strings into byte arrays which is how the source photos are stored within Grails.

6. Link to AngularJS

Installing AngularJS and modules is simple. Just add links to AngularJS and modules JavaScript files to the layouts. Also add the JavaScript files to web-app/js/

```
<g:external dir="js" file="vendor/angular.js"/>
 <g:external dir="js" file="vendor/angular-route.js"/>
 <g:external dir="js" file="vendor/angular-indexed-db.js"/>
 <g:external dir="js" file="vendor/angular-base64.js"/>
```

Above are the only additions required in the grails-app/ directory. The web app will now cache itself after the user visits the home page, the web-app can efficiently determine its online/offline status, the user only has access to online features when the web app is online, and the user can upload the topics stored on the device to the database server.

Additions in web-app/

Now the programmer needs to create the offline features of the web app. All this code is created in the web-app/ directory.

Only a few steps are required to complete the offline web-app. First, we need to specify the app, configure indexedDB, and configure the routing. Finally, we need to write the controllers and views. Because this is a small web app, many of the steps is coded in a single file, web-app/js/topic.js. A larger web app would use several files.

The topic.js file is an anonymous self-executing JavaScript function.

7. Specify the App

In topic.js, the app is created using the module function.

https://docs.angularjs.org/guide/module

https://docs.angularjs.org/api/ng/function/angular.module

The "app" Module is a container for all the parts of the web app. The module specifies the name of "app", "topic" and array of dependences. In this case the dependences are the other modules that app depends on: ngRoute for routing, indexedDB for interfacing to indexedDB, naif.base64 for string encoding files.

The "app" variable is run by chaining the "run" command, which sets the Boolean isOnline to false.

8. Configure IndexedDB

IndexedDBProvider is used to configure indexedDB in the config function

https://docs.angularjs.org/guide/providers

The topics and photos data store are created along with the indexes to access the objects in the data stores.

9. Configure the Routing

The routerProvide is used to specify the routing also in the config function. The configuration is simply chained "when" function-calls that specify the URL, the view (templateURL) and the controller name.

10. Make the Offline Views and Controllers

Generally, the controllers and views are written simultaneously. To study the code you'll probably want to look at the partial view and corresponding controller at the same time.

The offline partial views are located in the web-app/angular-views/topic/ directory. In general, they mirror the online topic views in grails-app/views/topic/ directory. Any g-tags (grails provided tags) are generally replaced with corresponding ng-tags (AngularJS provided tags).

The controllers are defined in the topic.js file. Each controller is created using a factory, a function called controller. Each controller factory call specifies the name of the controller, array of dependences and an anomalous function that defines any action of the controller. In ngPhotos the controller names mirror the view that they are associated with.

index.html and indexController

The index.html view and indexController show a list of topics stored on the device. The indexController needs to get the list of topics from indexedDB and pass them on to the index.html view. There are no controller actions in this view, but links in the list of topic will direct the browser to a view of the specific topic.

Parts of the view will only show if there are no topics. The controller adds the variables "noTopics" and “noServerTopics” to $scope, and the view can use the ng-hide and ng-show attributes to show and hide parts of the view.

If the app is online, the indexController uses $http to get JSON from the Grails controller to display Grails server topics within AngularJS.

NOTE: Stephen's initial approach for getting topics in the database server was to use Grails' controller and tags. Generally, this worked, but there is a bug. If the user uploads a new topic to the server database, the new topic will only show in the index view after reloading the app (i. e. closing the app window and returning to the app). This bug occurs because AngularJS is controlling loading of the page and the browser using the cached version of the index view, so the browser does request again the topics on the database. Returning to the index view, AngularJS will update its model, so the uploaded topic does not appear on list of topics stored locally. Consequently, a solution to the bug is to have the indexController in topics.js retrieve the topic list from the server.

The moral of the story is not to mix Grails and AngularJS dynamic content of a view. If using AngularJS on the page and then use AngularJS for all the dynamic content.

The indexController opens indexedDB, gets all the topics, and makes them available to the view by adding the list, "topics", to $scope. The view uses the ng-repeat attribute to list the topics.

A topic in topics has id and name properties. The links (a-tag) for each topic use the ng-href attribute. This enables the routing.

create.html and createController

The create.html view is a form for creating new topics offline. Only a name for the topic is required to create a topic, but photos can be uploaded to the new topic during creation. The create.html will need to pass the new topic name and photos to the createController, and the createController with then create the new topics and photo in indexedDB. A button, "Create," in the view invokes an action of the controller. Controller actions are specified as functions in $scope.

In the create.html view, the ng-model attribute in the input tag associates "topicName" with a property in $scope. The controller can then access "$scope.topicName."

In the create.html view, the photo are uploaded in the input tag with attribute type="file". The ng-model attribute associates "uploadPhotos" with a property of $scope. The controller will be able to access the file through $scope.uploadPhotos. The file upload uses the 'base-sixty-four-input" directive of the angular-base64-upload module. The directives will make a JS object with properties filetype and base64. The base64 is the string encoded content of the file.

In the create.html view, the ng-click attribute in the button tag associates the button with a function in the controller. In this case the createController should have a function name create.

The createController has an action that is associated with "Create" button on the view. The action is defined by the controller adding the "create" function to $scope. The create function generates a GUID. (See below in details.) Opens and inserts the topic in the topics data store. For each photo in the topic, the create function opens and inserts the photo into the photos data store. Promises are the functions defined in the "then" functions chained after the inserts. They do nothing. Instead the browser is redirected to the topic index view using $location.

https://docs.angularjs.org/api/ng/service/$location

view.html and viewController

The view.html and viewController show all the photos associated with a topic. Note that view.html is always called with parameter appended to the URL. The viewController use the parameter id to get the topic name and photos associated with the topic from indexedDB. The view.html has buttons for adding photos, editing the topic (changing the topic name) and deleting the topic. Also each photo has a delete button associated with the photo to remove the photos from indexedDB and the topic.

The viewController gets the "topicID" from $routeParams.id. It uses "topicID" to open the "topics" data store and add "topicName" to $scope in the promise. The view.html can access the topic name from $scope using the Angular expression, "{{topicName}}." Note that adding "topicName" to $scope is a promise (in "then" function chained with the indexedDB query). AngularJS will update the view when "topicName" changes. In this case that will occur when JavaScript returns from the asynchronous call to indexedDB.

When viewController is initialized (at the top of the viewController code), it gets the photos by querying the indexedDB data store. This is achieved by opening the “photos” data store and executing a search for each photo with the referenced topicID.. You must build a query to search the

datastore. This is done by using the “photos” parameter and calling “photos.query()” which substantiates a query object. Then the query is modified to search for the referenced topicID:

```
find = find.$index("topicID_idx").$eq($scope.topicID);
```

The $index() function parameters sets the index to “topicID_idx”, and the $eq() function parameter sets “$scope.topicID” as the topic id to search for. Lastly, you reference the query within an eachWhere call and set the returned content as the photos for the topic:

```
photos.eachWhere(find).then(function(e){ $scope.photos = e; });
```

In view.html, the buttons for adding photos and editing the topic are links to new views. So, they are not implemented by actions, rather they use ng-href in the anchor tag (a-tag) to direct the browser to the addPhoto.html or edit.html views. Note that the URL specified in ng-href has the topic id, "topicID," appended to it.

The button for deleting the topic is a controller action because the anchor tag has the ng-click attribute, ng-click="deleteTopic()". The viewController should add the function, "deleteTopic()," to $scope. the function deleteTopic, defined in viewController, uses "topicID" to find all the photos associated with topic and delete them from the photos data store. After deleting all the photos associated with the topic, the deleteTopic function deletes the topic from the topics data store and then redirects the browser to index page using $location

https://docs.angularjs.org/api/ng/service/$location

In view.html, the button for deleting a photo is an action. The ng-click attribute specifies the deletePhoto function with the photo ID as an argument. Note that the photo ID comes from a property in the photo object, photo.id. In viewController, the deletePhoto function uses photo.id to delete the photo from photos data store. Notice that deletePhoto function update view.html by retrieving all the photos associated with topic.

In view.html, the button for submitting a photo is an action. The ng-click attribute specifies the submitTopic function in the viewController. The submitTopic function works by opening the photos datastore and querying for all of the photos that need to be uploaded for the specified topic. This is done by using the “photos” parameter and calling “photos.query()” this substantiates a query object. Then the query is modified to search for the referenced topicID:

```
find = find.$index("topicID_idx").$eq($scope.topicID);
```

the $index function parameters sets the index to “topicID_idx” and the $eq function parameter sets “$scope.topicID” as the topicId to be searched for.. You reference the query within an eachWhere call. Next, you build a FormData object which will be submitted as the data for the AJAX post request. The topicName, and number of photos are appended to the object. The base64 encoded string of each photo is appended to the object with the name being “photo<index>”. The AJAX request is made to the submitOfflineTopic TopicController action. Lastly, the topic and associated photos are deleted and the web-app is redirected to the topic view using $location.

edit.html and editController

The view edit.html is a form that only updates the name of the topic. The view is only accessed from a topic view, and the link in the topic view will append the topic ID to the URL. The "Confirm" button specifies an action, ng-click="edit()"

In editController, the topic id is extracted from the URL and added to $scope and adds the edit function to $scope. The edit function then opens the topics data store and updates or inserts the topic name into the data store. The promise does not need to do anything. Finally the web-app is redirected to the topic view using $location and topicID

addPhoto.html and addPhotosController

The addPhoto.html view is a form only for adding photos to a topic. The view is only accessed from a topic view, and the link in the topic view will append the topic ID to the URL. The "Submit" button specifies an action, ng-click="submit()"

In addPhotosController, topicId is extracted from routing parameters and the submit function is added to $scope.

The addPhoto.html associates "uploadPhotos" with the uploaded files using ng-model attribute in the input tag. Note that the input tag also uses the base-sixty-four-input directive from the angular-base64-upload. In the submit function, the object to store into the data store is constructed using the model specified by ng-model in the view's input tag. The photos data store is opened and the photo object is inserted into the data store. Finally the web app is redirected to the view for the topic.

Detail Code Description

Recommended Development Process

Code description above was intended to introduce code required to add offline capability to the app. The above code description is not the process that a developer would actually use to implement the offline features. Below Stephen outlines the process that he would use to implement offline.

Tips: use Google Chrome in Incognito mode – it enables you to easily debug the HTML5 Cache and IndexedDB

1. Assess each of the views in grails-app folder. Specifically, pay attention to each of the views' controller actions, and variables you will need to craft your offline application.
    a) In the case of topic for ngPhotos
        i. Create.gsp, view.gsp, addPhotos.gsp, and edit.gsp were the important views
        ii. Adding/editing/deleting topics + adding/deleting photos were the important actions

iii. topicID and photoID are the important variables (also the photo data and topicname)

2. For each of the views, create an Angular route, controller, and template
   a) NOTE all JS+CSS must be in the web-app folder for offline apps
   b) Copy the important content over from the associated grails view into its' respective template
   c) Use expressions (e.g. {{topicID}}), $scope function calls using ng-click, and other AngularJS constructs to convert it over; see the views in web-app/angular-views/. The variable $scope refers to the current controller
   d) Using g-external tag include all of the AngularJS source files within the layout file (grails-app/views/layouts/main.gsp) and add the jQuery script within the layout file to disable all html elements with "grails" as a class when the application is offline. Include your angular module into the GSP that you desire to run your AngularJS app. Lastly, add in `<div ng-view></div>` where you want the angular content to display. In this case, I went with topic/index.gsp.
   e) You should now be able to manipulate the routing portion of the url to see everything properly
3. Implement each of the controllers using indexedDB as the model for our offline portion.
   a) https://github.com/bramski/angular-indexedDB
   b) Be sure to do dependency injection correctly, or Angular will complain like crazy in the javascript console - don't worry!
   c) At this point, you should now be able to properly use your application without any need from Grails for the backend!
4. Finish the offline implementation using the app cache
   a) Create the appcache manifest file. For this project, the appcache is associated with the /manifest route. You can view the appcache file for this project as a reference.
   b) For each static resource that you want to serve (your javascript and css files for example), add the url of that file into your appcache under the CACHE section. These files will be explicitly cached from now on.
   c) For each online route that you will have an offline component for, add a fallback route in the appcache. For example, for the /topic/create route in this example application, we add

```
FALLBACK:
${request.contextPath}/js/online.js ${request.contextPath}/js/offline.js
```

NOTE - use ${request.contextPath} for all components!

d) For everything else, it's OK to serve from the network, so add the following to the bottom of your appcache manifest:

NETWORK:

*

5. AngularJS Online/Offline content toggling

a) To complete the offline/online functionality, you should use the ng-show and ng-hide attributes with Angular templates and the "online" variable (use `$rootScope.isOnline = online;` within the run function) to toggle content based on app status. Be sure to inject the variable $rootScope into each controller you wish to reference "isOnline".

Layouts

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="${request.contextPath}/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.

Application Cache and the Manifest

This section is only a brief introduction to the app cache and describes the specifics techniques used by the ngPhotos 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

gairs-app/views/manifest/index.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:
# save these files locally for offline use
${request.contextPath}/js/vendor/angular.js
${request.contextPath}/js/vendor/angular-route.js
${request.contextPath}/js/vendor/angular-base64.js
${request.contextPath}/js/vendor/bootstrap.js
${request.contextPath}/js/vendor/jquery.js
${request.contextPath}/js/topic.js
${request.contextPath}/js/util.js
```

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:
# 'content-online' falls back to 'content-offline'
${request.contextPath} ${request.contextPath}/
${request.contextPath}/js/online.js ${request.contextPath}/js/offline.js
${request.contextPath}/topic/create ${request.contextPath}/topic/#/create
${request.contextPath}/topic/addPhotos ${request.contextPath}/topic/#/addPhotos
${request.contextPath}/topic/edit ${request.contextPath}/topic/#/edit
${request.contextPath}/topic/view ${request.contextPath}/topic/#/view
${request.contextPath}/topic/index ${request.contextPath}/topic/#/
```

Note that all the offline views are routed topic/index.gsp. Also the offline specify the specific partial after the hash.

The fallback can also be used for another reason. For example, the server would render "/ngphotos" and "/ngphotos/" 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
${request.contextPath} ${request.contextPath}/
```

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

Finally notice the following line in the fallback section

```
FALLBACK:
…
```

For telling if we are online or offline in the javascript
${request.contextPath}/js/online.js ${request.contextPath}/js/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

${request.contextPath}/topic/create ${request.contextPath}/topic/#/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

/${request.contextPath} ${request.contextPath}/

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:
...
${request.contextPath}/js/online.js ${request.contextPath}/js/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="${request.contextPath}/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

${request.contextPath}/topic/view/3

should not be cached, but the view ${request.contextPath}/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.

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

${request.contextPath}/topic/#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

${request.contextPath}/topic/#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 ngPhotos uses

${request.contextPath}/topic/#/views/3

The browser interprets the above URL as referencing

${request.contextPath}/topic/

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.

GUID

For convince all keys should be Global Universal Identifies (GUID). True GUIDs are hard to generate, but myPhotos and ngPhotos use 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.

Comparing myPhotos and ngPhotos Code

A difference between Adam's myPhotos code and Stephen's ngPhotos code is that the data stores are organized differently. The ngPhotos data stores are organized so that the photos have keys to their topics while the myPhotos data stores are organized so that topics have an array of foreign keys to the photos. Whenever I can (it is not always possible), I try to organize data into a tree. In the case of myPhotos and ngPhotos, the topics are parents of photos because the topics have many photos and photos only belong to a single topic. In this case, the data is clearly organized as a tree. I call the structure of the myPhotos data stores "top-down" because topic is the parent data store of the photos data stores and have the references to photos, so the parent data store is referencing the child data store. I call ngPhotos's structure "bottom-up" because the child data store, photos, has references to the parent data store. The top-down structure is natural, but for coding with JavaScript the bottom-up, structure is better for two reasons. The most obvious reason is that the

bottom-up structure does not require an array of references, only a single reference. Consequently, the data store is precisely know and "normalized." However, this advantage is nominal. The real advantage is dealing with JavaScript callbacks. Deleting a child using a bottom-up structure only requires a single asynchronous call to the child data store, while in a top-down structure deleting a child requires two layered asynchronous calls, one to the parent to delete the child's keys from the array and a second to delete the child. The bottom-up data store structure general eliminates one layer of callbacks from the code.

The astute reading will question the efficiency using a bottom-up data store structure when creating a view of parents composed of their children, for example when creating a topic view with all the photos. The astute reader would legitimately claim, that the code would have to search the entire child data store (in this example the entire photos data store) to find the associated children entries. That would be true if the data store API did not have an "indexing" feature. Fortunately, IndexedDB does have indexing, and programmers need to consider the indexing of their data stores in terms of the requirements of their applications. Indexing imposes a memory requirement; a second data structure is created associating parent keys to children keys. In a bottom-up tree data store structure, the requirements are obvious the children need to be indexed by the parent foreign key, a small memory requirement (a two-column table of integers).

Stephen's ngPhotos code differs from Adam's myPhotos code in two important aspects:

- ngPhoto's uses AngularJS to bind the data from the data store (IndexedDB) to the view
- ngPhotos's data store structures are "bottom-up."

Using AngularJs, eliminates one layer of callbacks from the JavaScript code because of its data binding. Using bottom-up data store structure eliminates another layer of callbacks from the JavaScript code. That is two layers of callbacks less than in ngPhotos code. In JavaScript coding, improving the readability and maintainability of code **is** eliminating callbacks.

In addition, AngularJS offers more, the ng-repeat directive eliminates many for-loop index dependences, meaning that coders do not have to worry about the closures in view loops. Using ng-repeat the array view elements are automatically maintain through editing, inserting and deleting view elements.

Summary

Using features provided by AngularJS eliminates much of the JavaScript code required by the developer. In particular, Angular JS data-binding makes promises an effective programming tool. The View View-Model Controller design pattern organizes the program logic into controllers. The simple structure of the View-Model ($scope and ng-model attribute) leads to natural controller programming of the view data. It is also agnostic to the data-store. This does burden the developers with programming their own version of the Domain, but this can be modularized by creating an AngularJS module.

Stephen's ngPhotos demonstrates the ease of adding offline capabilities to an online Grails web app using AngularJS. The online and offline code is nearly completely separated into grails-app/

directory for the online version and web-app/ directory offline version. The only cost is the duplication of the online views in the offline view.

Stephen's ngPhotos makes the JavaScript code clearer by eliminating two layers of callbacks. The ngPhotos code has only a single layer of callback that are natural to read for JavaScript programmers. This makes the code more readable and maintainable.

Discussion

Developers that are already proficient in AngularJS could theoretically develop the online and offline simultaneously and use ngResource to interface with the database, however that is outside the scope of this project.

https://docs.angularjs.org/api/ngResource

https://docs.angularjs.org/tutorial/step_11

This approach would make the both the online and offline version of the web app a SPA. It would eliminate many of the Grails controllers and views, but the web app would still make use of the Grails Domain classes.