AngularJS 2.0 Data Persistence Design Doc

*Status: Draft*

*Authors: Jeff Cross (*[*crossj@google.com*](mailto:crossj@google.com)*)*

*Ritchie Martori (*[*ritchie@strongloop.com*](mailto:ritchie@strongloop.com)*)*

*Anant Narayanan (*[*anant@firebase.com*](mailto:anant@firebase.com)*)*

## Objective

This document is concerned with the design of data-focused AngularJS v2.0 modules, considered to be part of the core distribution of Angular.

### End User Stories

These stories describe the kinds of apps that should be easy to build with Angular 2.0, focusing on application data, written from the end user’s perspective. Some features, like operational transformation, may not be supported by the core data persistence framework, but they should be made possible by mechanisms within the framework.

#### Offline & Realtime Synchronization (Github Client)

As a user of a github client I want to be able to instantly start the app and see the list of issues that were assigned to me. I want to be able to read the description of these issues and post comments. If for whatever reason my network connection is flaky or unavailable, I want to be notified ASAP that I'm working in the offline mode, but I want to be able to continue browsing the issues and post comments without any delays or interruptions. It should be clear to me which of my new comments were synchronized with the server and which were only stored in the local storage. I should also be able to add comments and other metadata to issues, which will eventually be persisted to other users when network connectivity is restored.

#### Offline Access Policies (Mobile Music Player)

As a music listener on my way to work, I'd like to use a web based audio player on my phone to play music I would enjoy. As I move from home wifi, to cellular data network, to coffee shop wifi, I want music to play without skipping a beat. The network in my home allows me to stream music without any issues, but even though I have full bars on my phone, it isn't fast enough to stream music without skipping a beat. The app should default to playing music that is available on the local disk and only stream songs when the network is capable. When I am driving, the app should make the choice for me so I don't crash into a pole. I should be able to skip songs and search my local library even when streaming music isn't possible. When the app isn't being intelligent, I should be able to force it to only use songs on the local disk.

#### Realtime Multi-User Presence (Github Client)

As a user of a github client I want to be able to look at an issue and if I'm online and some other team member opens the same issue I want to be visually notified of this. Similarly when that team mate moves to another issue, I should be able to know that they are not looking at this issue any more. If a change to the current issue is made by a team mate, I want to be able to see this change reflected in my view and possibly even get a notification (visual highlight) of the change for a limited amount of time.

#### Session-Persisted App State (Web Encyclopedia)

As a mobile user of an online encyclopedia, I want to be able to interact with UI widgets, like an accordion that contains different data for an entry, and be able to open my mobile browser later and see the page in the same state I left it. If the browser automatically reloads the page, I don’t want to see the initial state of the page; I want to see the page how I left it.

#### Offline Support with Proactive Caching (Mobile Sales Tool)

As a car salesman at a dealership, I’d like to use a web app on my tablet to make notes about my customers, including in what makes and models of cars they are interested. The app should anticipate that I have limited internet connectivity when I’m on the lot, although I know that I’ll have internet connectivity in-between customers when I’m back in the lobby.When the app has reliable internet connectivity, it can download updates about current inventory and offers. The app should also upload information about customers I’ve been helping, including interesting vehicles. The app should make sure to encrypt all data as it’s stored in DOMStorage and sent over the network, and clear data once it’s successfully been uploaded to our servers. I can securely access the user’s information from my home computer around dinner time, since they’re probably hoping I give them a follow-up call just as they’re sitting down together. I should also be able to open the app without an internet connection, and have some indication about how stale the inventory and offer data is.

#### Offline Archiving and Querying (Realtime Group Chat)

As a programmer who takes a subway to work, I want to be able to use my chat app to send group messages to other members of my team, and search for other messages in our group chat history, even when I’m deep underground without internet connectivity. I should be able to perform complex offline searches, like saying “show me messages from Frank within the past day that contain the word ‘unicorn.’” I should be able to respond to messages with witty jokes, and have subtle notifications from the app regarding the status of the message, such as “waiting for internet connection to send.” I can decide to cancel the message if too much time passes, and the joke would no longer be funny. If more messages arrived between the last message I see in history and the message I’ve just sent, the history of the group chat should indicate the timing of messages, with some UI indication that messages were spliced in after-the-fact.

#### High Frequency, Multi-Client Data Synchronization (Document Editor)

As a product manager working on writing a technical design doc in cooperation with other members of my team, I want to be able to use a web-based document editor to edit the document at the same time as all other members of my team, and see my teammates’ changes reflected in the same app instantly and automatically. I expect the app to be intelligent about how to merge changes together, even when multiple clients are experiencing connection issues and are editing a similar section of the document.

#### Multi-Client Shared State & Predictive Cache

As an avid music listener, I want to be able to listen to my music on my phone, tablet, or computer. I want a music app that can live in all these places, and show me what I’m listening to at any given time. Since I’m always listening to no more than one song, I should be able to control the playback and playlist from any client, regardless of which client is playing the audio. I should be able to browse artists and albums, even without an internet connection, because the app is intelligent enough to cache metadata of artists, albums, and tracks I’ve already looked at. The app should cache additional metadata of artists that I’d probably like, and should give me the option of downloading music to my device to play even when my device doesn’t have reliable internet access.

#### Multiplayer board game

As a gamer, I want to play board games with my friends in multiplayer over the web. As I play the game, I want to be able to chat with my friends, as well as make sure my moves are propagated in realtime to other players in the game. I want to be notified if the game goes offline and I shouldn’t be able to make any moves until the game state has caught up to the current state again.

#### Multiplayer RPG

As a PC gamer, I want an experience similar to PC games on the web. RPG games are graphics heavy and typically circumvent the DOM and render frames directly on <canvas>, WebGL or uses a framework like [famo.us](http://famo.us/). In addition, they are fast paced and require the game state to be synchronized with the server in realtime.

Editing medical record of a person

# Prior Art

Projects with some overlapping scope or challenge.

* [PouchDB](http://pouchdb.com/) - Interesting strategies for synchronizing changes from multiple sources
* [Racer](https://github.com/codeparty/racer) - Node lib for realtime sync, build on ShareJS
* [Meteor](https://www.meteor.com/) - DDP and strategies for sync
* [ShareJS](https://github.com/share/ShareJS) - Node.js operational transformation, inspired by Google Wave
* [Restangular](https://github.com/mgonto/restangular) - Alternative to $resource
* [EmberData](https://github.com/emberjs/data) - EmberJS’s ORM-like data persistence library
* [ServiceWorker](https://github.com/slightlyoff/ServiceWorker/blob/master/explainer.md)
* [IndexedDB Polyfill](http://nparashuram.com/IndexedDBShim/) - Polyfill IndexedDB with browsers that support WebSQL (Safari)
* [BreezeJS](http://www.breezejs.com/)
* [JayData](http://jaydata.org/)
* [Loopback](http://strongloop.com/mobile-application-development/loopback/)’s ORM
* [ActiveRecord](http://en.wikipedia.org/wiki/Active_record_pattern)
* [BackBone Models](http://backbonejs.org/#Model)
* [lawnchair](http://brian.io/lawnchair/?utm_source=twitterfeed&utm_medium=twitter) - Lightweight client-side DB lib from Brian Leroux
* [TogetherJS](https://togetherjs.com/) - Mozilla Labs’ multi-client state sync/collaboration lib.
* [localForage](https://github.com/mozilla/localForage) - Mozilla’s DOMStorage improvement library.
* [Orbit.js](https://github.com/orbitjs/orbit.js) - Content synchronization library
* [Angular-data](http://angular-data.pseudobry.com/) - EmberData-like analog for Angular.js

# Detailed Design

The data persistence modules will be built in phases, starting with the lower-level utilities. Starting from the bottom will allow designs for higher level modules to emerge organically as common patterns emerge from using the low-level services in real applications.

The phases may overlap, and modules may change priority based on needs that emerge as applications are developed using the new data persistence modules.

The below diagram, and the rest of the Detailed Design section of this document should serve as a guide to understand how different modules and services will relate to each other in Angular 2.0. No module or service should be considered final. This document only mentions as many APIs are as useful to understand how a module will be used, and leaves the bulk of the decisions for API design to developers to decide during the iterations in which a module is being developed.

## 

## Phase 1: Utilities

The first phase of development is focused on providing low-level utilities, and simple mid-level abstractions. As a result of the development in this phase, developers can build rich, realtime, offline apps, but will have to write a lot of marshalling code due to the absence of higher-level libraries that will provide more abstract utilities for working with persisted models.

Much of the first phase of development will be inspired by implementations of Angular 1, such as the $http library. However, community feedback on these implementations, as well as recent developments in security and browser capabilities should be considered.

### Phase 1.1: ngHTTP (Module)

The ngHTTP module will consist of low-and-mid-level utilities for communicating over HTTP and working with REST-like resources over HTTP. Although all the cool apps are using WebSockets (or HTTP fallback equivalent), applications being developed today still primarily use request/response HTTP as the data transport. Many applications that rely heavily on WebSockets will still use HTTP for all data operations except “read.”

#### $http

In general, the $http service will be a similar implementation to Angular 1. The Service will be written in ES6, and may provide more methods to more easily support different responseTypes (like $http.getBlob(), $http.getArrayBuffer()).

##### New Considerations

* Provide API to support FormData object

#### $resource?

The $resource factory will be replaced by the higher-level Model class in Angular 2. In addition to what $resource already provides in Angular 1, Model will add support for paging, querying and relating HTTP resources.

### Phase 1.2: ngStore (Module)

The ngStore module contains services that provide access to local “disk” storage provided by native browser APIs like localStorage, sessionStorage and IndexedDB.

The localStorage and sessionStorage APIs will be wrapped in Angular to take advantage of ES6 getters/setters, to provide a simpler API and input validation. Depending on the design of Change Detection in Angular 2, these wrappers could also facilitate automatic change notification as values are returned.

See Store API docs (includes sessionStorage and localStorage): <https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage>

See IndexedDB docs: <https://developer.mozilla.org/en-US/docs/IndexedDB>

#### $localStorage (Service)

The $localStorage service will provide a simple API for storing key/value pairs that should live beyond the current session. This service shouldn’t do much beyond what the native API does, but should tie nicely into other aspects of Angular.

Although this service will have a public API, it will also be used by other modules and services. For storing larger amounts of data, $localStorage should be avoided, since browsers will load the entire contents of an origin’s localStorage from disk each time the page is loaded. See: [The Performance of localStorage Revisited](http://www.nczonline.net/blog/2012/04/25/the-performance-of-localstorage-revisited/)

#### $sessionStorage (Service)

The sessionStorage API can add significant value to mobile apps in particular, because, unlike state data that only lives in memory, it persists data between page reloads/restores. Separate windows/tabs of an app can have their own state, and protect that state from wanted or unwanted page reloads. See the [Session-Persisted App State](#h.nq38a3ewys34) user story for more context.

#### $indexedDB (Service)

The $indexedDB service will be a thin wrapper for the indexedDB, but will also polyfill Safari in order to use the indexedDB API with WebSQL. This service will be utilized heavily by the Model service, which will support querying offline data.

#### $localDB (Service)

The $localDB service will be a higher-level, asynchronous service to provide a local data store to store queryable data, or key-value data. The service will be asynchronous, even though the $sessionStorage and $localStorage APIs are synchronous, to allow for optimizing storage underneath without requiring synchronous browser APIs.

In most cases, this should be the service used to persist data locally, rather than using the low-level utilities directly, in order to take advantage of performance optimizations. This service will be relied on heavily by the Model class, in order to automatically cache data.

##### Example methods:

* [get|set|update|unset]() - Sets a value as if using localStorage
* [get|set|update|unset]Temp() - Sets a value as if using sessionStorage
* [create|addTo|splice]Collection()
* [create|addTo|splice]TempCollection()
* find() - Queries a collection
* findTemp() - Queries a sessionStorage-based collection

All services within the ngStore module will support encryption, managed through the $localDB service. Details about the implementation can be found under the [Storage Encryption](#h.57girrphcre7) heading below.

### Phase 1.3: ngWebSocket (Module)

#### $ngWebSocket (Factory)

$ngWebSocket will be a factory to provide a thin wrapper around the HTML5 WebSocket API. The factory will provide convenient nuances not provided by the native API, such as automatic/configurable re-connection and automatic queuing of send requests. See the early design for ngWebSocket in Angular 1: <https://github.com/angular/ngSocket>

## Phase 2: Offline

### Phase 2.1: ngOffline (Module)

This module is concerned with providing state information and utilities to application developers to build offline-first applications.

As far as the Web Platform goes, the tools for supporting fully offline apps are still somewhat in flux. For example, ApplicationCache is hard to work with because it’s completely declarative and can’t be manipulated from client code, and the API for its more-imperative successor, ServiceWorker, is still being defined.

#### $serviceWorker (Service)

ServiceWorker is a proposed API, still being designed, that allows registration and installation of a script into a document to intercept and manage all network requests for that document (or for configurable path scopes within a domain).

This API is still early, but a sparse implementation should be provided to start being used by Angular 2’s dependency injection and $http. Because it’s not natively supported in any browser, Angular won’t be able to automatically intercept all network requests.

See [ServiceWorker Explainer](https://github.com/slightlyoff/ServiceWorker/blob/master/explainer.md)

#### $connection (Service)

Provides information about the current state of network connectivity for the application. Uses $serviceWorker to provide binary connectivity state, but may eventually provide more granular data about the quality of the connection, based on calculating how long resources take to download, and how many requests timeout.

#### $appCache (Service)

This service acts as a registry of resources and scripts to be cached, including documents, media, data, and scripts. The service should be loosely-based on the philosophy of HTML5 ApplicationCache, but with some added control to add resources to cache, and clear the cache.

In the absence of a native ServiceWorker implementation, developers will still need to maintain a cache manifest to load their base HTML page and some library/script to load their application offline. The services in ngOffline, in conjunction with Angular v2’s DI should be able to handle the rest of the logistics of supporting offline functionality. DI will not need to be concerned with whether a module is fetched over the network or from a localCache, as long as the $appCache service overrides the default ES6 Module Loader to provide a custom [fetch](http://wiki.ecmascript.org/doku.php?id=harmony:module_loaders#options.fetch_resolved_fulfill_reject_normalized_referer_metadata) implementation.

The configuration managed by the $appCache service will be stored using ngStore.

## Phase 3: Rich Data

### Phase 3.1: ngData (Module)

#### Model (Class)

The Model class acts as an extensible abstraction layer on top of low-level storage and network utilities to manage fetching , caching, and querying data. Behind the scenes, Model will create private caches of fetched data + buffers in ngStore, allowing querying, paging, and updating of models locally without necessarily requiring network access.

Properties of Model instances can be bound to inside views. Updating properties of Model instances should be managed through mutator methods. Views can bind to plain JavaScript objects as well as Model instances in Angular 2.

The Model API needs to be further defined, but the proposed approach is as follows:

* Model is an ES6 class, which can be instantiated directly or extended to define custom behavior. E.g. class Issue extends Model {}
* Model’s will maintain a “view” of underlying data, and optionally a buffer of additional data to be viewed in time. This solution should reduce the size of the first request for data, then request additional data (as configured), drastically increasing the speed at which additional data is displayed when asked for by views (like ng-repeat, infinite scrolling directives, etc).
* By default, Model instances keep state synced to ngStore, but can accept an implementation of IAdapter to easily support synchronization/replication with network-based persistence (although IAdapter doesn’t necessarily need to persist to a network location).
* Model will provide a limited set of operations on data, to allow for easy implementation of IAdapter. E.g. limit(), skip(), find(query), findById(), count(), order().
* Model will seamlessly fetch cached data, and will provide flexible strategies to let developers specify how to treat data in different network conditions.
* Model will support reading streams of data, or data that’s constantly changing. However, Model assumes that implicit model changes are one-way (reading), leaving the application responsible for explicitly writing data changes. In other words, local changes to models within views wouldn’t be implicitly written back to the server.

#### ChangeEvent (Class)

This type will be used to broadcast changes to models to other parts of Angular, in order to allow customization of how data changes are handled. This will also be used internally within the ngData model to provide metadata of a change to an underlying Adapter, in order for the Adapter to make more complex decisions about how to handle data. Ideally, the ChangeEvent should contain a series of changes to be applied to data, a copy of the data in its original state before the changes were applied, and some hint to the source of the event.

#### IAdapter (Interface)

The IAdapter interface specifies the methods that should be implemented by a participating Adapter in order to make synchronizing Models with a server Just Work. When instances of Model are created, they should be constructed with an implementation of IAdapter, unless the Model is never meant to be persisted to a server.

The IAdapter interface may allow adapters to provide more efficient and sophisticated means of synchronizing data, such as supporting patching.

##### IAdapter Example

class FirebaseAdapter implements IAdapter {...}

var fooAdapter = new FirebaseAdapter(‘http://foo’);

var myModel = new Model({adapter: fooAdapter});

# Caveats

The HTML5 FileSystem API has not been included in this design doc because Chrome is the only browser that supports the feature, and the API is not on a standards track.

As of right now, no module or service is concerned with knowing anything about authentication or authorization, because the problem is too broad and fragmented to solve at this point. Perhaps in time, patterns will emerge that could be incorporated into the scope of this document.

# Security Considerations

## Storage Encryption

Developers should be able to easily encrypt data that’s persisted to client devices, in order to be able to safely store sensitive data without worrying about the device falling into the wrong hands. The ngStore module should provide simple, predictable, flexible means of accomplishing this.

The proposed design is:

* Allow application developers to provide a single encryption key to the $localdb service once per in-memory session, which will be used to encrypt/decrypt data.
* The encryption key will be used to encrypt/decrypt keys and values persisted to underlying storage mechanisms if developers indicate that individual reads/writes should be encrypted at runtime.
* The encryption key will not be persisted beyond memory, unless an application developer chooses to store it somewhere else, e.g. a hard-coded constant.
* The module will use a default encryption algorithm, but can be configured to use any arbitrary encryption algorithm.
* Services will provide a third boolean argument to get/set methods to specify if data should be encrypted, default to false. If true, the service will wait until a key has been provided to resolve the get/set method. The default value is “false” because if it were “true,” the service could perpetually wait for a key to be provided before performing operations, which is likely to confuse developers who don’t care about encrypting local data.

Why one key?

* A use case hasn’t yet surfaced which justifies more than one key for client-side storage encryption. E.g. either all available data should be accessible by the user, provided they have the correct key.
* A much better developer experience is achieved by setting an encryption key one time.

Example Implementation (Based on Angular 1.x semantics):

|  |
| --- |
| myModule.run(function (myCryptKeyService, $localDB) {  /\*  \* It's up to the developer to determine how to get the key.  \* It could depend on authentication process with server,  \* or could be a hard-coded constant, etc.  \*/  myCryptKeyService.get().then(function (key) {  /\*  \* Synchronous method on $localDB.  \* Any $localDB method requesting encrypted  \* data will wait for the key to be set.  \*/  $localDB.setEncryptionKey(key);  });  }).  controller(‘UserPrefsController’, function ($scope, $localDB) {  /\*  \* Tell $localDB to get the userPrefs, encrypted=true  \* $localDB will wait until the encryptionKey has been set to resolve the  \* promise returned from getKey().  \*/  $localDB.get('userPrefs', true).then(function (prefs) {  $scope.prefs = prefs;  });  }); |

## Http Considerations

The ngHttp module will implement measures to ensure safe cross-site scripting similar to Angular 1, including but not limited to:

* JSONP support
* Receiving XSRF tokens, and passing to origin server as header or cookies.

# Performance Considerations

## Speed

The chief consideration for deciding how each module is implemented, is to provide the end user of an application with the most responsive experience possible. This isn’t necessarily a straightforward goal.

1. Initial load speed. Speed from page request to page being ready to use.
2. Data update speed. Such as when using a component like infinite scroll, or when viewing data that changes in real time.
3. Page change speed. Speed navigating to a subview of an application.

The most important place to balance these concerns well is in the ngData module, where the developer is most abstracted from the loading of data. Consider an example. A developer is building a contact list app, to display all of a user’s contacts from a remote service. The developer extends the Model class to fetch and cache all the contacts. The Model class could:

1. Err on the side of initial page load time, and could load only 10 contacts to show the initial view and be done with it until the user scrolls, at which point the model requests more contacts.
2. Err on the side of data update speed, and load all 1000 contacts from the server into memory, even though the user is only viewing 10 right now.
3. Make an initial request for only the required 10 contacts, and make a subsequent request to load the next 20 and previous 20 contacts.

The “C” approach is what will be most supported and encouraged throughout the design of Angular 2.0 data persistence, particularly in the higher levels of abstraction like ngData. The ngData module should provide some basic configuration to control the aggressiveness of pre-fetching for different models. The module should also be smart about detecting network capabilities, and throttling back on pre-fetching if the network isn’t capable, so that other assets won’t be bottlenecked by limited network resources.

Beyond what’s configurable, developers who would like to deviate from these patterns will have to use lower level utilities.

## Space

Since browsers limit available storage space differently, the ngStore services will rely on application developers to manage space responsibly, and handle write failures gracefully.

For higher-level abstractions, such as ngData, it is the framework’s responsibility to intelligently manage the amount of cached data in storage and in memory. How this will actually be implemented is TBD, but it will likely be a process of configurable, auto-expiring data (collections or objects that haven’t been interacted with in a while), with a garbage collection process that runs periodically as an app is being used.

This article provides useful tables and explanations of quotas of different browsers: <http://www.html5rocks.com/en/tutorials/offline/quota-research/>.

### Temporary Storage

Browsers previously had set limits of 5MB on DOMStorage and IndexedDB, but have since lifted limits. Our primary concern is with IndexedDB, since that is where all Angular ngStore APIs will be storing data underneath.

#### Chrome

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **App Cache** | **File system** | **IndexedDB or WebSQL** | **Local Storage** | **Session Storage** |
| **Chrome Android** | **33** | max quota | max quota | max quota | 10MB | 10MB |
| **Chrome iOS (Webview)** | **iOS 6,7** | 100MB? | N/A | 50MB WS | 5MB | 5MB |
| **Chrome Desktop** | **33** | max quota | max quota | max quota | 10MB | 10MB |

*(Table data from* [*html5rocks*](http://www.html5rocks.com/en/tutorials/offline/quota-research/) *via* [*Eiji Kitamura*](https://plus.google.com/+agektmr/posts)*)*

Chrome allows a portion (currently 50%) of the available disk space to be shared among web apps, where each web app can have no more than 20% of the shared pool. I.e. if 100GB is available on the user’s disk, a single app would have up to 10GB to use for its own temporary storage, (100\*.5\*.2) (everything but Filesystem storage is considered temporary by Chrome). Another way to look at it is that no single domain can exceed 10% of the total available disk space.

This policy applies to Chrome on Android. On iOS, Chrome is limited to the iOS WebView quota.

<https://developers.google.com/chrome/whitepapers/storage#temporary>

The Opera browser uses Blink, and so its quotas tend to be the same as Chrome.

#### Android Browser

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **App Cache** | **File system** | **IndexedDB or WebSQL** | **Local Storage** | **Session Storage** |
| **Android Browser** | **4.3** | Unlimited? | N/A | 200MB~ WebSQL | 2MB | Unlimited |

Since Angular 2.0 will be targeting Android 4.x, all solutions need to account for limitations of the stock Android browser and its WebView.

#### Firefox

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **App Cache** | **File system** | **IndexedDB or WebSQL** | **Local Storage** | **Session Storage** |
| **Firefox Android** | **26** | 5MB soft, unlimited | N/A | 5MB soft, unlimited | 10MB | 10MB |
| **Firefox iOS (Webview)** | **iOS 6,7** | 100MB? | N/A | 50MB WS | 5MB | 5MB |
| **Firefox Desktop** | **26** | 500MB, unlimited | N/A | 50MB, unlimited | 10MB | 10MB |

*(Table data from* [*html5rocks*](http://www.html5rocks.com/en/tutorials/offline/quota-research/) *via* [*Eiji Kitamura*](https://plus.google.com/+agektmr/posts)*)*

Firefox allows up to 50MB in IndexedDB Storage before prompting a user to allow more.

<https://developer.mozilla.org/en-US/docs/IndexedDB#Storage_limits>

This article suggests that Firefox on mobile prompts the user at 5MB instead of 50: <http://refcardz.dzone.com/refcardz/html5-indexeddb>

#### Safari

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **App Cache** | **File system** | **WebSQL** | **Local Storage** | **Session Storage** |
| **Safari iOS** | **iOS 6,7** | 300MB? | N/A | 5MB\*, 10, 25, 50 | 5MB | 5MB |
| **Safari Desktop** | **6,7** | unlimited? | N/A | 5MB, 10, 50, 100, 500, 600, 700... | 5MB | Unlimited |
| **iOS WebView** | **6,7** | 100MB? | N/A | 50MB | 5MB | 5MB |

*(Table data from* [*html5rocks*](http://www.html5rocks.com/en/tutorials/offline/quota-research/) *via* [*Eiji Kitamura*](https://plus.google.com/+agektmr/posts)*)*

*\*iOS 7 Safari currently has a bug that throws an exception when requesting more than 5MB when creating a database, requiring developers to not explicitly ask for permission, and letting Safari prompt the user.*

It seems that Safari imposes a soft limit on all local storage of 5MB, after which point it prompts the user to allow more space, which increases the limit to the hard limit of 50MB (although the data in the table says the limit can be progressively increased to a max of 50MB).

<http://stackoverflow.com/questions/10985959/what-is-the-maximum-size-of-a-web-sql-db-in-ios-safari-on-the-ipad-how-about>

#### IE11

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **App Cache** | **File system** | **IndexedDB** | **Local Storage** | **Session Storage** |
| **IE11 Desktop** | **6,7** | 100MB? | N/A | 10MB, 250MB, ~999MB | 10MB | 10MB |

*(Table data from* [*html5rocks*](http://www.html5rocks.com/en/tutorials/offline/quota-research/) *via* [*Eiji Kitamura*](https://plus.google.com/+agektmr/posts)*)*

It looks as though 250MB is the limit, and will throw if any app tries to write more than that.

<http://stackoverflow.com/questions/14717739/storage-limit-for-indexeddb-on-ie10>

### How to Gracefully Handle Limitations

There is a W3C [draft](http://www.w3.org/TR/2013/WD-quota-api-20131105/) to implement an API that allows querying the browser to get the currently-available quota for a domain, as well as its current usage, referred to as “Quota Management API”. The proposal also includes a means of requesting more quota. Unfortunately, Chrome appears to be the only browser with a (deprecated) implementation of this as of March, 2014 (further experiments will likely be [behind a flag](https://groups.google.com/a/chromium.org/forum/#!msg/blink-dev/ciicp7mJolQ/FzN4zzQ5o3EJ)). According to these [meeting minutes](http://www.w3.org/2013/11/11-webapps-minutes.html#item09), Mozilla is planning to implement this experimental API as well.

So how do we get more space? In short, we leave it up the browsers to prompt users to allow more space as the application requires it, and provide a way for application developers to handle aborted writes as they occur. On some platforms, such as iOS 7 and Windows, writes will need to be wrapped within a try/catch, to workaround bugs (features?) that throw errors when trying to write data above quotas. All utilities that are restricted by storage quotas should be designed and documented to encourage developers to assume that they will run into storage limits, and thus should design their applications to provide a good user experience when this occurs.

For the parts of Angular’s Data Persistence that automatically cache data, there should be configurable strategies to automatically clean up the cache while the application is running. For example, items in indexedDB could have properties with their timestamp and their expiration date, which could be examined once per session by a “cleanup” script. Alternatively, an item could have a “last used” timestamp, with a threshold of time after which it can be deleted from the cache.

### Memory

One of the goals of the ngData module is to allow templates and controllers to be built without having to be too concerned with how much data is in memory at any given time. I.e. a developer should have to think less about paging, skipping, and buffering of large collections of data when using it in a template with an ng-repeat or other directive. This becomes especially important on mobile devices which have less memory than desktop computers. This will largely be accomplished by the Model class, which will provide an abstract “View” or “Cursor” on top of underlying data, which will seamlessly provide data as it is needed by the view.

## Test Strategy

Modules in this design doc will provide a similar approach as Angular 1 in how developers are able to test their implementations with public APIs, by mocking native API behavior.

The Model class will specify an interface that underlying storage adapters will need to implement in order to let the Model read and write data with the adapter. To help ensure that the adapter is implemented properly, a generic suite of E2E tests will be provided to authors of adapters.

The ngOffline and ngStore modules will provide backend mocks for all of the APIs in order to remove the need to interact directly with cache and storage APIs. If unexpected calls are made to native APIs, an error will be thrown causing tests to fail.

It’s anticipated that some features within the scope of this document may be difficult to test, such as verifying that a sound was successfully loaded and played in a mobile browser (which isn’t as straightforward as one would think). As these difficulties arise, it will be necessary to leverage the knowledge of developers and test engineers on how to reliably test these behaviors with tools available on our CI servers. Fortunately, Google has internal groups willing to help solve these problems.

# Work Breakdown

The biggest priority is getting utilities for robust network communication and full offline apps in place before an Angular 2.0 beta release.

Here is an anticipated order of development work.

* Phase 1: ngHttp, ngStore, ngWebSocket
* Phase 2: ngOffline
* Phase 3: ngData

**More Jeff Notes**

* Think more about using SPDY to do hot pushes of updated assets (directives, providers, et), cooperating with DI. This would allow developers to push new features without ever requiring a refresh. It would also provide a means to automatically update stale assets that have already been loaded by $appCache. Providers would need to have a design that separates state from function, and all components would need to have some sort of versioning. James DeBoer’s idea is to base versions on something like commits/SHAs, where I can determine that one updated component requires another updated component by comparing commit history: certainly some complexity to be considered with that approach.
* Could HTTPBackend be simplified for testing?
* Consider pros/cons of supporting expected parsed type with http method names. E.g. $http.getArrayBuffer(‘http://buffer-src-url’) or $http.getBlob(‘<http://blob-url>’)
* Is it possible to partially load an ArrayBuffer into memory? If so, how about sharding bytearrays in IndexedDB?