Technical University of Košice

Faculty of Electrical Engineering and Informatics

# Atmosphere: Concurrency enabled data synchronization platform with HTML5/JS and Cocoa clients

**Appendix A: Developer's guide**

**2012** **Vojtech Riník**

Technical University of Košice
Faculty of Electrical Engineering and Informatics

# Atmosphere: Concurrency enabled data synchronization platform with HTML5/JS and Cocoa clients

## Appendix A: Developer's guide

| | |
|---|---|
| Study Programme: | Informatics |
| Field of study: | 9.2.1 Informatics |
| Department: | Department of Computers and Informatics (KPI) |
| Supervisor: | Assoc. Prof. Ing. František Jakab, PhD. |
| Consultant(s): | Ing. Ivan Klimek |

Košice 2012 Vojtech Riník

# A Developer's guide

## A.1 Introduction

Atmos2 is a library that adds synchronization to models in JavaScript applications.

Its goal is to make user interfaces very responsive by hiding the networking.

### A.1.1 Overview

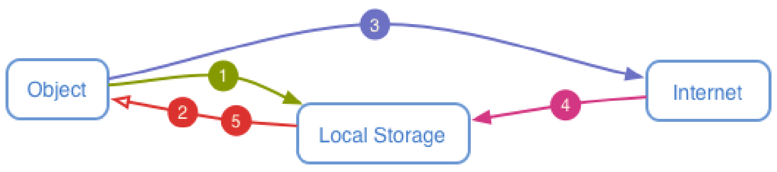


**Figure 1 – 1: Overview of Atmosphere functioning**

1. Object is saved to local storage.
2. User interface is immediately updated
3. Request is made to update the object on the remote side
4. Local version is updated from the response
5. User interface is updated according to changes received from the server

### A.1.2 API Example

```
MyModel.sync(remote: true)
```

**Listing 1: Fetching objects from remote source**

Code in Listing 1 fetches remote data and persists them in a local collection. Triggers an event on the model, so that the user interface could be updated.

```
myRecord.save(remote: true)
```

**Listing 2: Saving object**

Code in Listing 2 stores changes in local storage and makes remote request. When the request is done, it updates the local data and triggers event to update the user interface.

### A.1.3 Features

1. **API configuration**: Advanced options of configuring API. So far used with typical Rails app and Google Tasks API.
2. **Fetching and caching**: Objects can be fetched and cached in local storage.
3. **Sync and posting**: When object is changed, it's saved locally, then posted to the server.
4. **Offline usage**: The design allows for building offline applications.
5. **Live updating**: Comes with [atmos2-server](https://github.com/vojto/atmos2-server) which is lightweight Node.js proxy for updating objects in real time.

### A.1.4 Notes

The current implementation uses the model layer of Spine.js. All the code that works with Spine is encapsulated in the AppContext class, which is 80 lines of CoffeeScript.

## A.2 Getting started with the JavaScript library

Take existing Spine app or create a new one.

Install atmos2.

```
git clone git://github.com/vojto/atmos2.git node_modules/atmos2
```

**Listing 3: Installing atmos2 package**

### Adding modules to slug.json

Add these modules to your `slug.json`:

```
"dependencies": [

       "atmos2",
       "atmos2/lib/spine"
],
```

**Listing 4: Updating slug.json with atmos2 package**

### Setting up the model

Let's say this is your current Spine model:

```
class Task extends Spine.Model
  @configure 'Task', 'title', 'kind', 'selfLink'
```

**Listing 5: Current Spine model**

### Extending the model

All you have to do is require Atmosphere's Spine adapter, and extend model with it.

```
require('atmos2/lib/spine')

class Task extends Spine.Model
  @configure 'Task', 'title', 'kind', 'selfLink'
  @extend Spine.Model.Atmosphere
```

**Listing 6: Requiring Atmosphere's Spine adpater**

Atmosphere will automatically use the local storage.

### Setting up the synchronizer

Do this somewhere, where it will be executed before anything else.

```
Atmos = require('atmos2')

atmos = new Atmos
```

```
atmos.resourceClient.base = "https://www.googleapis.com/tasks/v1/users/
    @me"
```

**Listing 7: Setting up the synchronizer**

As you can see, this example will work with Google Tasks API. But first, Atmosphere needs more information.

```
atmos.resourceClient.routes =
  Task:
    index: "/lists"
atmos.resourceClient.addHeader "Authorization", "OAuth #{token}"
atmos.resourceClient.IDField = "id"
atmos.resourceClient.dataCoding = "json"
atmos.resourceClient.itemsFromResult = (result) -> result.items
```

**Listing 8: Setting up Synchronizer**

1. `routes` specifies path that will be hit on actions: index, create, update, delete. (TODO: Add others to the example.)

2. `addHeader` adds header to every request. In this case, we're adding OAuth Authorization, which we've taken care of someplace else, so we have a token.

3. `IDField` every retrieved object must have an ID. Some APIs expose this ID in field `id`, others `identifier`, so this settings lets you set it. If a record with empty ID will be retrieved, Atmosphere will throw an error.

4. `dataCoding` can be `form` or `json`, specifies in what format will be outgoing data encoding and sent. (Also, what `Content-Type` will be used)

5. `itemsFromResult` is a function that will be used to get the items from object decoded from response JSON. In this case, we receive a JSON that looks like this: `items: [...]`, so we need to tell Atmosphere how to look for actual records.

### A.2.1 Fetching objects

```
Task.sync(remote: true)
```

**Listing 9: Fetching objects**

This will first fetch data from local storage triggering the `refresh` event, then make the network request, update local data, and trigger `refresh` event again.

### A.2.2 Saving objects

```
task = new Task({title: "Task 2"})
task.save(remote: true)
```

**Listing 10: Sending objects**

Calling `save` will first save the object locally, then it will make network request to save it again. `create` action will be used.

If you call `save` on already saved object, `update` action will be used. Atmosphere keeps track of all objects you saved with `remote` flag to differentiate between objects that have been sent previously and those once that haven't.

## A.3 Getting started with the Cocoa library

### A.3.1 Installation

Atmosphere for Cocoa is available as CocoaPods package. In order to install Atmosphere, your first need to enable CocoaPods for your project. For installation instructions please see the CocoaPods website.

Add Atmos2 to your Podfile.

```
dependency 'Atmos2', {:git => 'git@github.com:vojto/atmos2-cocoa.git',
    :branch => 'master'}
```

**Listing 11: Adding Atmos2 to your Podfile**

### A.3.2 Setup

The next step is to setup the synchronizer. In order to do that, open the main entry point for your application (usually the app delegate) and include Atmos2 header file.

```
#import <Atmos2/Atmosphere.h>
```

**Listing 12: Including Atmos2 header file**

In your main class header file create a new instance variable that will hold reference to the synchronizer.

```
@property (strong) ATSynchronizer *sync;
```

**Listing 13: Adding instance variable for synchronizer**

The next step is to set up the synchronizer.

```
self.sync = [[ATSynchronizer alloc] initWithAppContext:self.
   managedObjectContext];
[self.sync setBaseURL:@"http://localhost:6001"];
[self.sync loadRoutesFromResource:@"Routes"];
[self.sync setIDField:@"_id"];
```

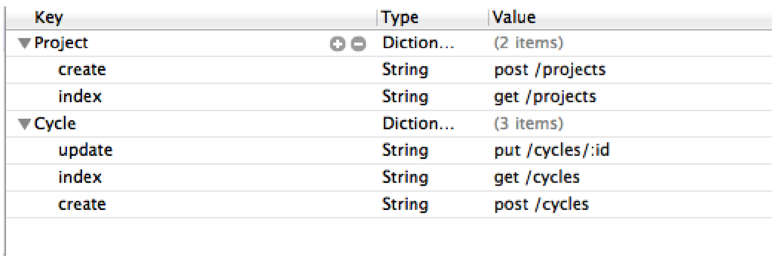
**Listing 14: Setting up the synchronizer**

The Listing **??** shows very simple setup that first initializes a new synchronizer with current application context, sets base URL of the REST source, loads routing table from `Routes.plist` and finally sets up the ID field for parsing responses.

The routes file should contain a dictionary for each entity with keys representing actions and values representing HTTP actions in `method /path` format. An example routes file is portrayed in Figure 1 − 2.

### A.3.3 Fetching objects

Listing 15 shows how to fetch objects for an entity Project.



| Key | Type | Value |
|---|---|---|
| ▼Project | Diction... | (2 items) |
| create | String | post /projects |
| index | String | get /projects |
| ▼Cycle | Diction... | (3 items) |
| update | String | put /cycles/:id |
| index | String | get /cycles |
| create | String | post /cycles |

**Figure 1 – 2: Example routes file**

```
[self.sync fetchEntity:@"Project"];
```

**Listing 15: Fetching objects for an entity**

Calling this command will fetch remote objects and store them locally. The application should be using data bindings or similar mechanism to update data when they change in Core Data store.

### A.3.4 Saving objects

Saving objects is done automatically by watching changes in Core Data context. When an object is saved locally such as code in Listing 16 shows, Atmosphere automatically sends it to the server.

```
PLProject *project = (PLProject *)[NSEntityDescription
    insertNewObjectForEntityForName:@"Project" inManagedObjectContext:
    self.context];
project.title = @"New Project";
[self.context save:&error];
```

**Listing 16: Saving objects locally**