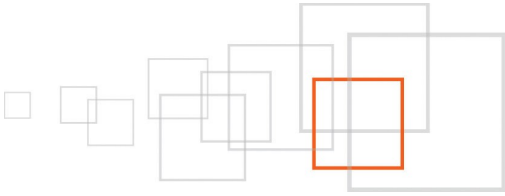
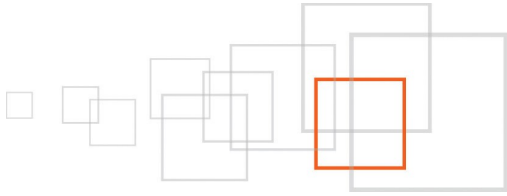
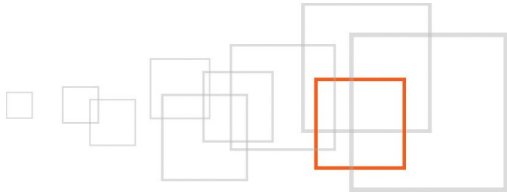


Building native mobile applications with the eZ Publish REST API

By Łukasz Serwatka
http://serwatka.net



Index



1 Goal description

eZ Publish is a Web Content Management System that provides a platform to publish content via any channel. Its powerful presentation engine enables you to create websites and pages that display your content in a variety of renderings. Its powerful API directly and simply integrates your content with any web-enabled application on any device, such as the iPad, iPhone, or an Android device, without ever interfering with, or impacting the platform itself.

At the end of this tutorial, you will have learnt the basics of mobile application development for both iOS and Android platforms, consuming content from eZ Publish. CMS-side adjustments for the mobile channel will be acquired too. This cheatsheet will help you leverage the multichannel capabilities of eZ Publish, and its REST API in future projects, in a more systematic fashion.

2 Introduction

This article is the second in a two-part series. The first article introduced the foundational concepts for mobile web development with eZ Publish by showing how to create a mobile siteaccess as well as basic apps that act as simple wrappers for that siteaccess. This document focuses on creating native mobile applications that communicate with eZ Publish via a REST interface.

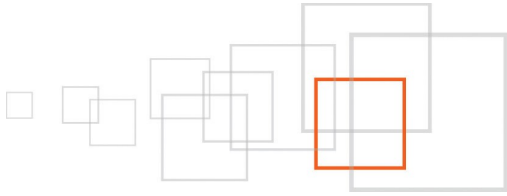
2.1 *Background*

Native applications are applications designed to work on a specific device such as an iPhone or BlackBerry. Users download these applications from an app store or through an internal enterprise program. The main advantage of native applications is that they offer access to built-in device features such as cameras or GPS functionality. Native applications often perform better than other types of apps, as they live on the device itself. These applications can also present unique marketing opportunities if they are distributed via an app store.

Hybrid applications access the mobile website (which is also accessible through a URL outside of the app) through an app, bringing together features of both web and native applications. They are downloaded via an app store and look and feel like a native application, although they are simply displaying the mobile website. The main advantage of a hybrid application over a native application is that a consistent user experience can be enjoyed by a wider audience (via the mobile URL) while still making the most of the marketing opportunities presented by app store distribution. Changes are made on the web application end rather than requiring the user to perform the update manually via the app store.

3 Pre-requisites and target population

This guideline is aimed at eZ Publish developers, partners, and integrators who need to build and native mobile applications that communicate with the eZ Publish REST interface. It outlines the steps required to create web service enabled native mobile applications for eZ Publish. The instructions described herein assume that you have already set up eZ Publish 4.5 or higher. It is recommended that you read the eZ Publish REST API documentation, as the focus of this article is not to explain all of the details of the interface, but to show how to connect a mobile application to it. You should also be familiar with some basics about setting up



an iOS or Android application, although anybody should be able to copy and paste the code examples to create a working application.

3.1 What do I need before I can start?

3.1.1 General requirements

- Web server with PHP installed
- Relational Database Management System
- Installed eZ Publish 4.5 instance
- Configured development environment for at least one platform: iOS (using Xcode) or Android (using Eclipse)

3.1.2 eZ Publish Requirements

- OS: Linux 2.6, Windows 2008 (IIS 7 only)
- Web Server: Apache 2.2.x (prefork mode), MS IIS 7 (with Microsoft URL Rewrite Module 1.1)
- DBMS: MySQL 5.0.x, MySQL 5.1.x, PostgreSQL 8.4, Oracle 11g
- PHP (mod_php) + PHP CLI + apache: PHP 5.2.1 -> PHP 5.2.17, PHP5.3.x
- PHP (FastCGI) + PHP CLI + IIS: PHP 5.3.x
- Graphic handler: ImageMagick >= 6.4.x, PHP extension GD2

3.2 General guidelines for eZ Publish REST based mobile clients

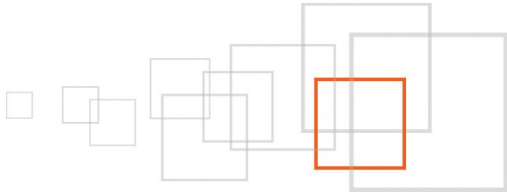
- List all the features necessary for the application
- Determine who your users are and define a usage context
- Do not block the main thread; always fetch data asynchronously
- When communicating over a network, write code as robust as possible when it comes to failure handling
- Minimize the amount of time spent transmitting or receiving data. Make use of pagination, and fetch only the data that is absolutely necessary.
- Alert users if they are using your application when their device is set to “airplane mode”

4 eZ Publish REST interface

In short, a REST-ful architecture supports server responses to client requests. The most popular and relevant model in our case is where the client makes an HTTP request to a REST web service URL.

The REST API for eZ Publish seeks to make the content and the features of an eZ Publish installation available to external clients. The first phase of development centers around providing read-only access to the data within the installation, and for ways to easily extend the REST interface with custom functionality.

The REST interface is bundled with eZ Publish 4.5.0. If you wish to use it, you need to make the the REST endpoint, index_rest.php, available via rewrite rules. Conversely, if you prefer to not expose this functionality,



you need to make sure that this file is not exposed in your rewrite rules, either via .htaccess files, or the virtual host configuration.

The installation process is well documented in the official eZ Publish documentation. Refer to the "Installation" section of the "Rest API" chapter.

4.1 Configuration

The eZ Publish REST interface supports various authentication mechanisms. For demonstration and development purposes, we will disable authentication in order to make the REST resources easily accessible. To do so, create the file settings/override/rest.ini.append.php with the following settings:

```
<?php /*

[Authentication]

RequireAuthentication=disabled

*/ ?>
```

You can then test that the API is accessible by going to http://www.example.com/api/ezp/content/node/2/list. The output at this URL should prompt you to download a JSON-encoded file with data about the child nodes under node 2 -- something we will use in our example applications.

For more information on the eZ Publish REST API URL format, see the documentation. Also see the list of services page for more details on what kind of data fetches can already be performed with the eZ Publish REST API.

5 Mobile applications

In this section we are going to build two simple mobile applications: one for iOS and one for the Android platform. These applications will perform like news readers, listing the content available and then showing the relevant full content when an item is selected. The applications will poll the eZ Publish REST web service and process the JSON response.

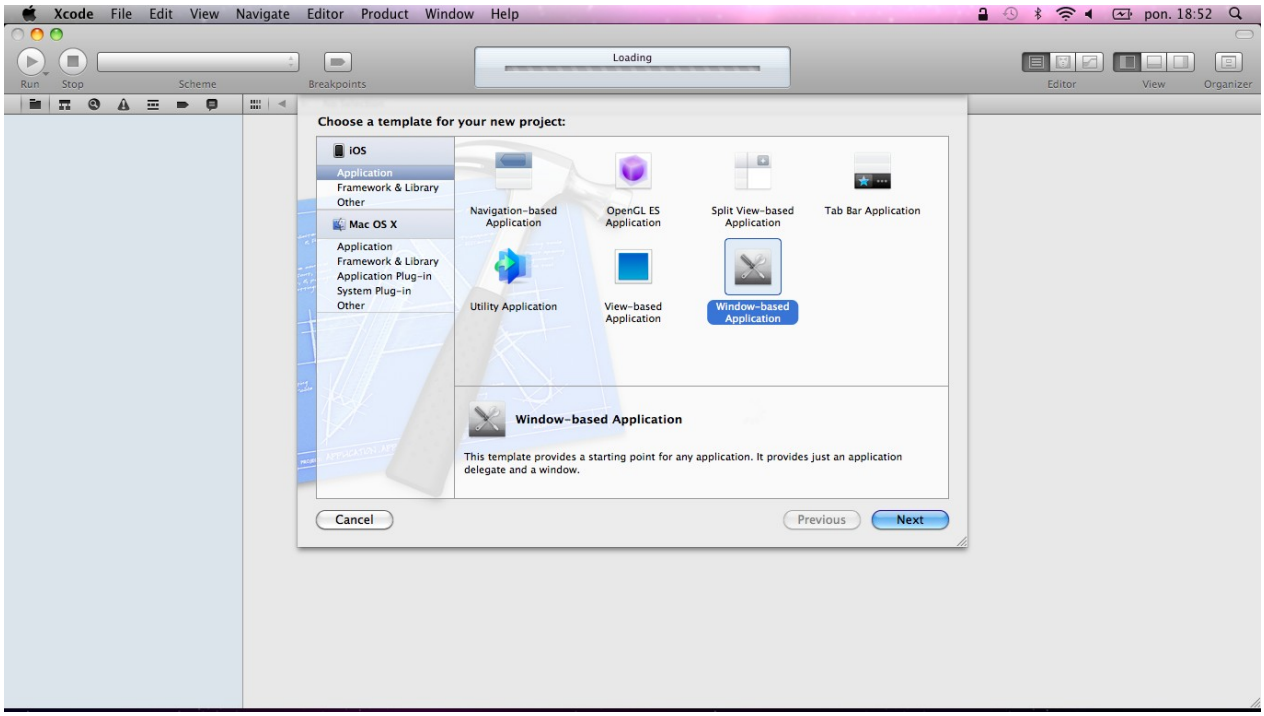
We assume that you already have a development environment for at least one platform ready; refer to the "iOS Developer Tools" or "Android SDK Installation Guide" for more information on setting up the appropriate development environment.

The sample applications built in this article are available for download on GitHub.

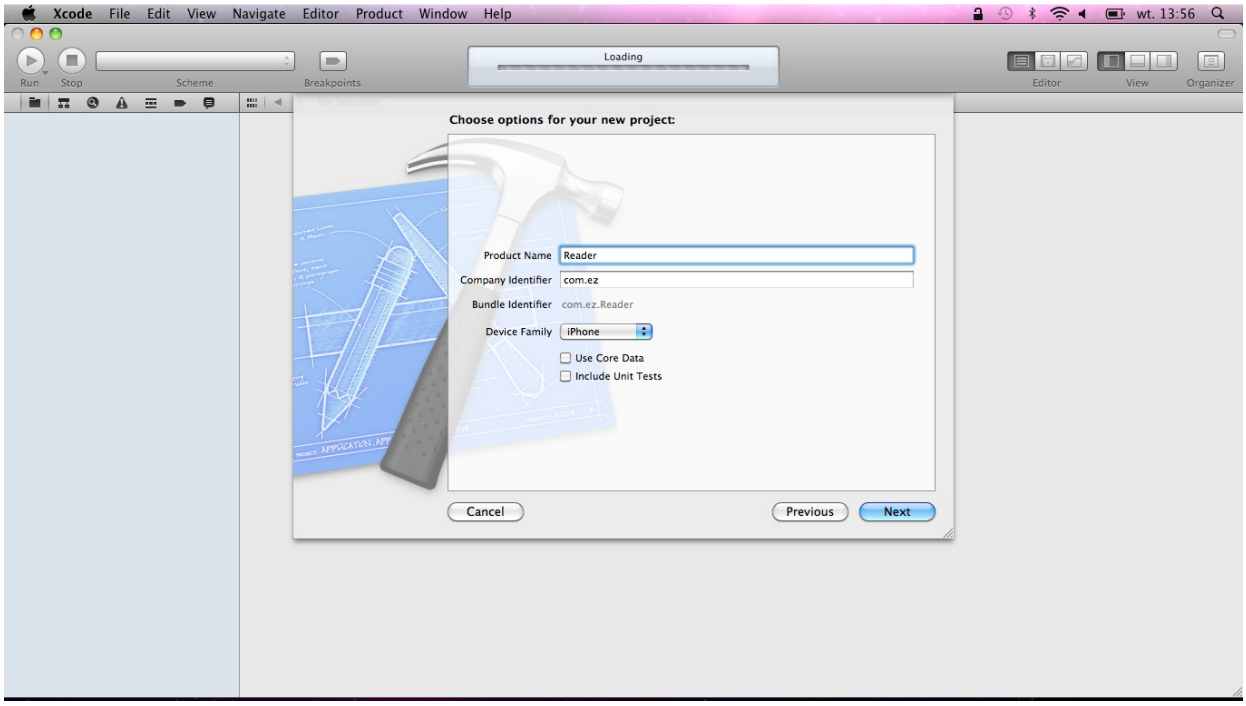
5.1 iOS sample application

Open Xcode, and on the first screen, select "Create a New Xcode project". Or, select "New Project..." from the "File" menu.

A window will appear giving you several application templates to choose from. Create a barebones Cocoa Touch application by selecting the “Window-Based Application” icon.



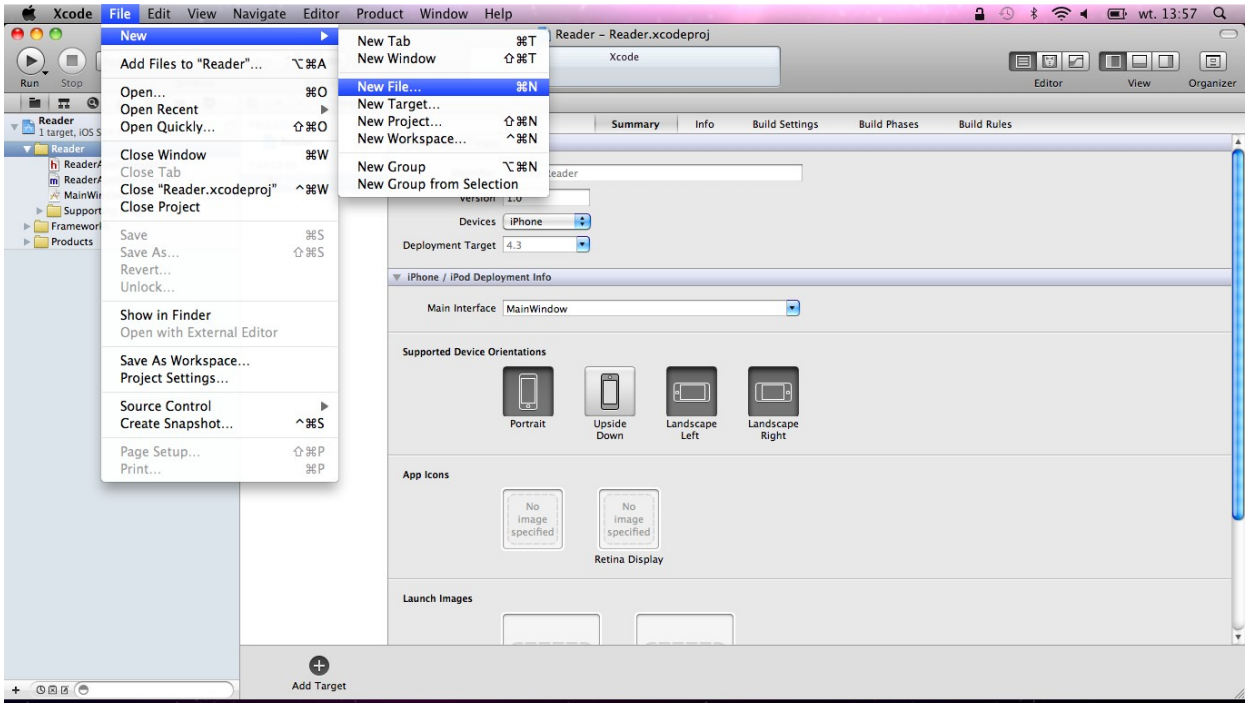
Click the “Next” button. On the next screen, enter “Reader” as the product name, “com.ez” as the company identifier, and select “iPhone” as the device family. Unmark the “Include Unit Tests” checkbox (as we are not going to write unit tests this time).



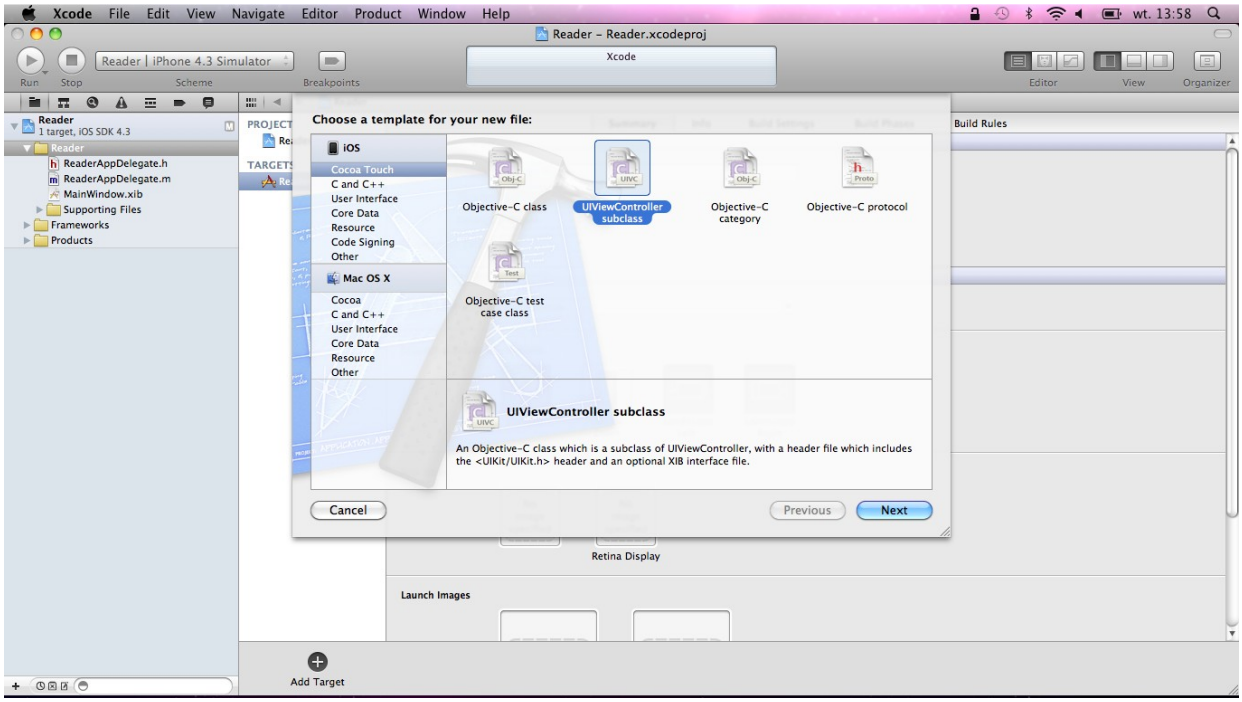
Click the “Next” button and select the location at which you want to save the project. Once the project is created, the project window will appear on your screen. Take a look at the contents of the “Project” table on the left side of the project window. In general, there are two types of files used to create an application: code and resources. Code is written in Objective-C, C, or C++. Resources are things like images and sounds that are used by the application at runtime. The groups in the project window are purely for organizing files. You can rename them to whatever you want.

5.1.1 Setting up the main view

In the next step, we are going to create a new view controller that will be responsible for loading our views with content served by eZ Publish. Select the “Reader” group on the left side and select “New -> New File…” from the “File” menu.

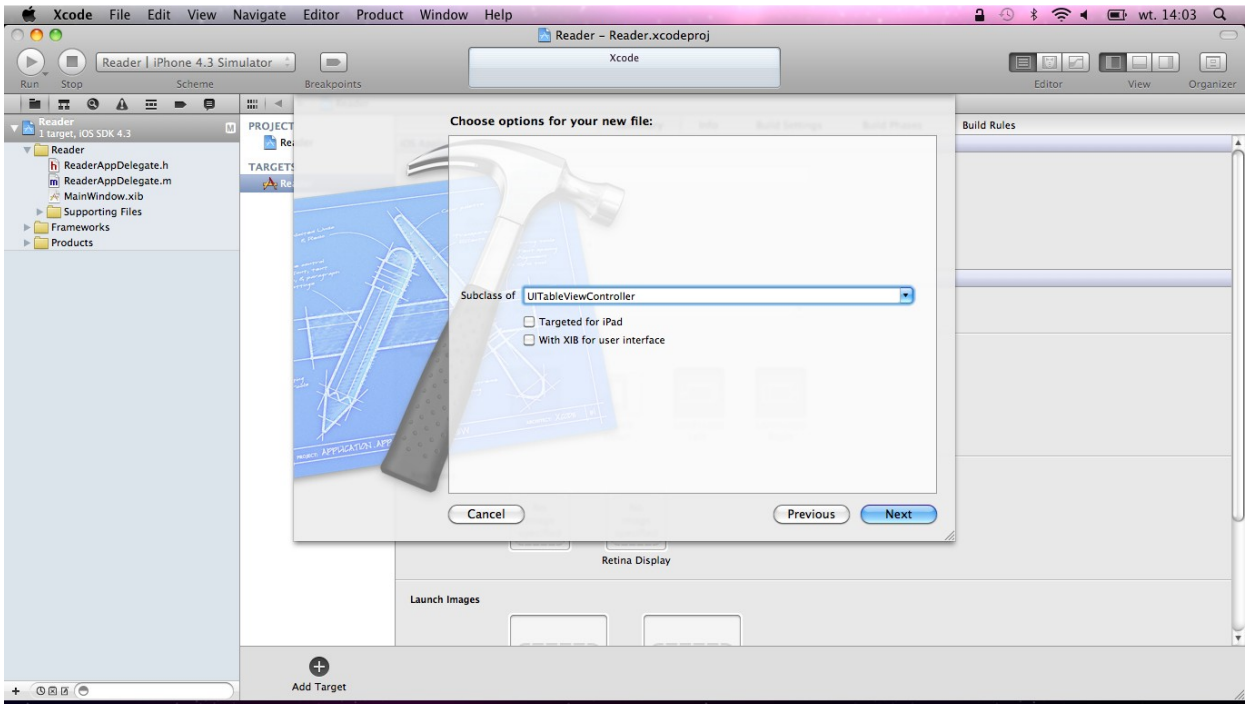


Then, select the "UIViewController" subclass from the "Cocoa Touch" group and click the "Next" button.



On the next screen, verify that the "Subclass of" field contains the "UIViewController" class, and that the checkbox titled "With XIB for user interface" is unmarked. (XIB stands for XML Interface Builder, which we

used to build a hybrid application in part 1 of this article series.) Then, click the "Next" button and name the file "JSONTableViewController".



Our application will present multiple screens of information: the main screen is a list view, and the other screens are details views. We will use a "UINavigationController" object, which will maintain a stack of those screens. The stack is an array of view controllers, and each screen is a view instance controller using the "UIViewController" class. When a "UIViewController" screen is on top of the stack, its view is visible. The "UINavigationController" object will be initialized with "JSONTableViewController" as its root view controller. Select the "ReaderAppDelegate.h" file and insert the following code:

```
#import "JSONTableViewController.h"
```

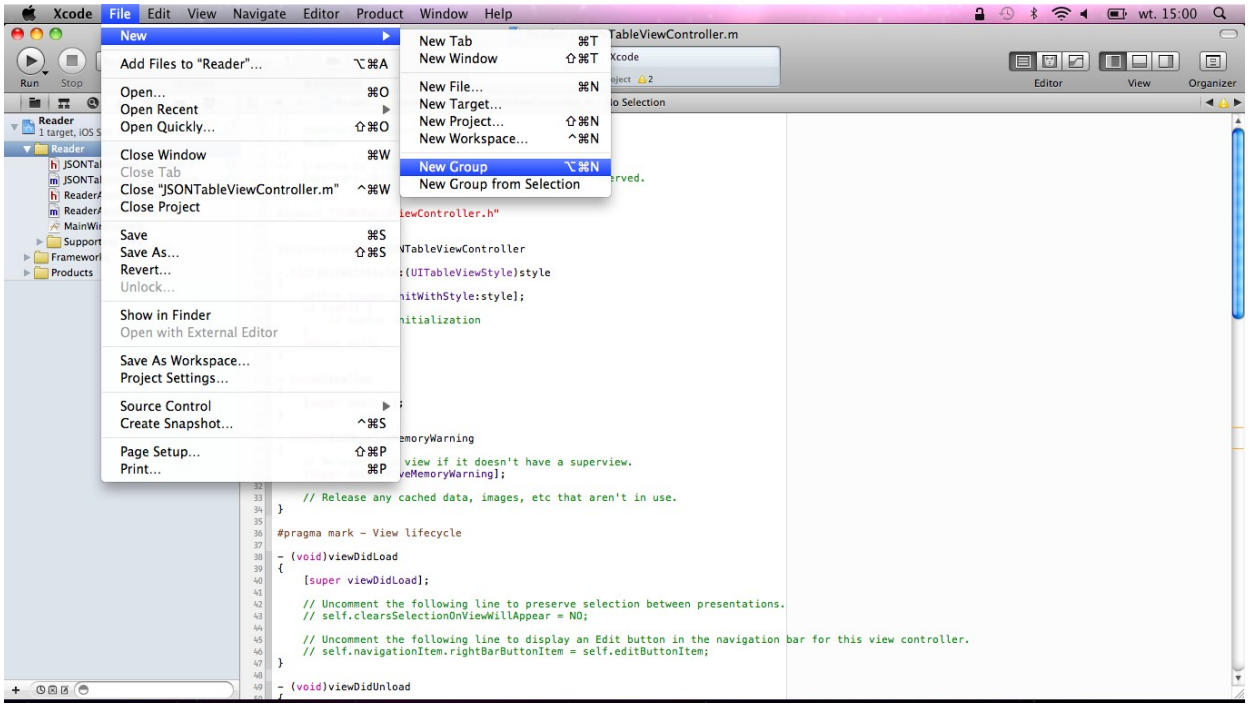
Next, in the "DemoAppAppDelegate.m" file, create the "UINavigationController" with "JSONTableViewController" and set it as the root view controller of the window:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions

{

    // Create the JSONTableViewController

    JSONTableViewController *jsonTableViewCtrl = [[JSONTableViewController alloc]
initWithStyle:UITableViewStylePlain];


    // Create an instance of a UINavigationController

    // Its stack contains only jsonTableViewCtrl

    UINavigationController *navigationController = [[UINavigationController alloc]
initWithRootViewController:jsonTableViewCtrl];

    [jsonTableViewCtrl release];
```

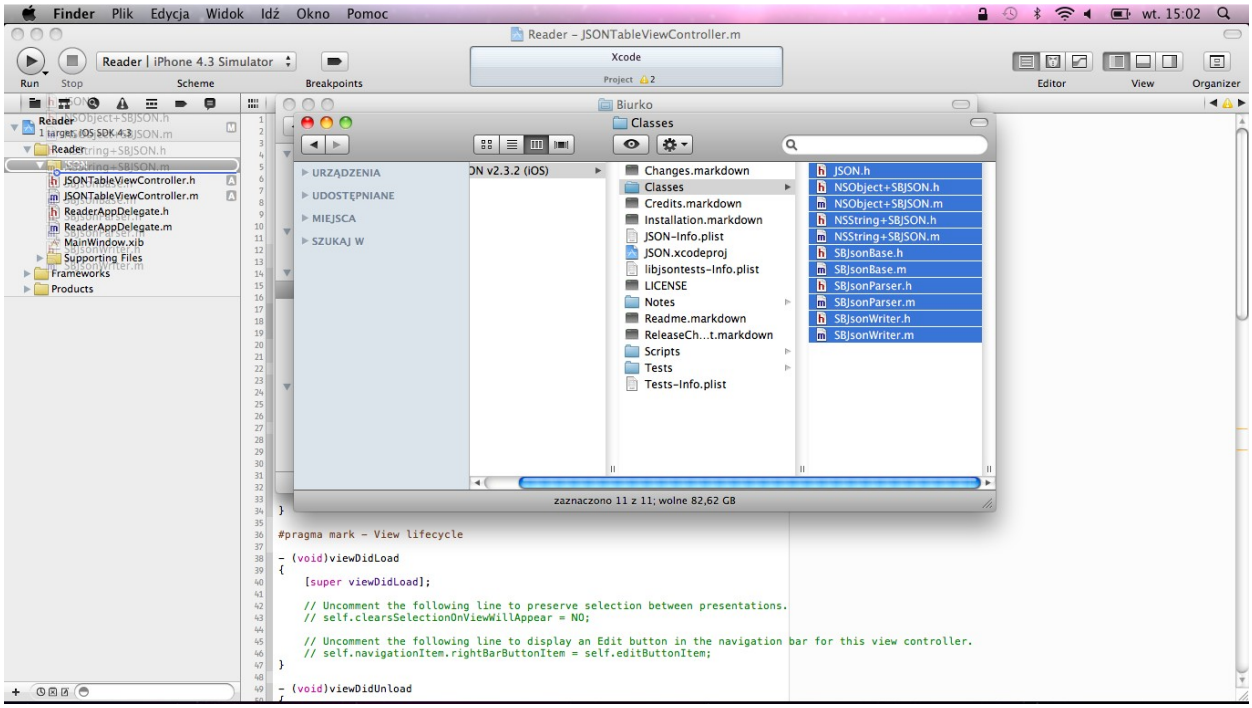
```
    // Place navigation controller's view in the window hierarchy
    [self.window setRootViewController:navigationController];
    [navigationController release];

    [self.window makeKeyAndVisible];
    return YES;
}
```

Build and run the application to verify that everything works. At this point, you should see only empty table rows with a blue navigation toolbar at the top. Next, we are going to load data from the eZ Publish REST web service and display it within the table view. In order to do so, we will need to include an open source JSON parser written in Objective-C, freely downloadable from GitHub. When this article was published, JSON framework 2.3.2 was the latest stable version available.

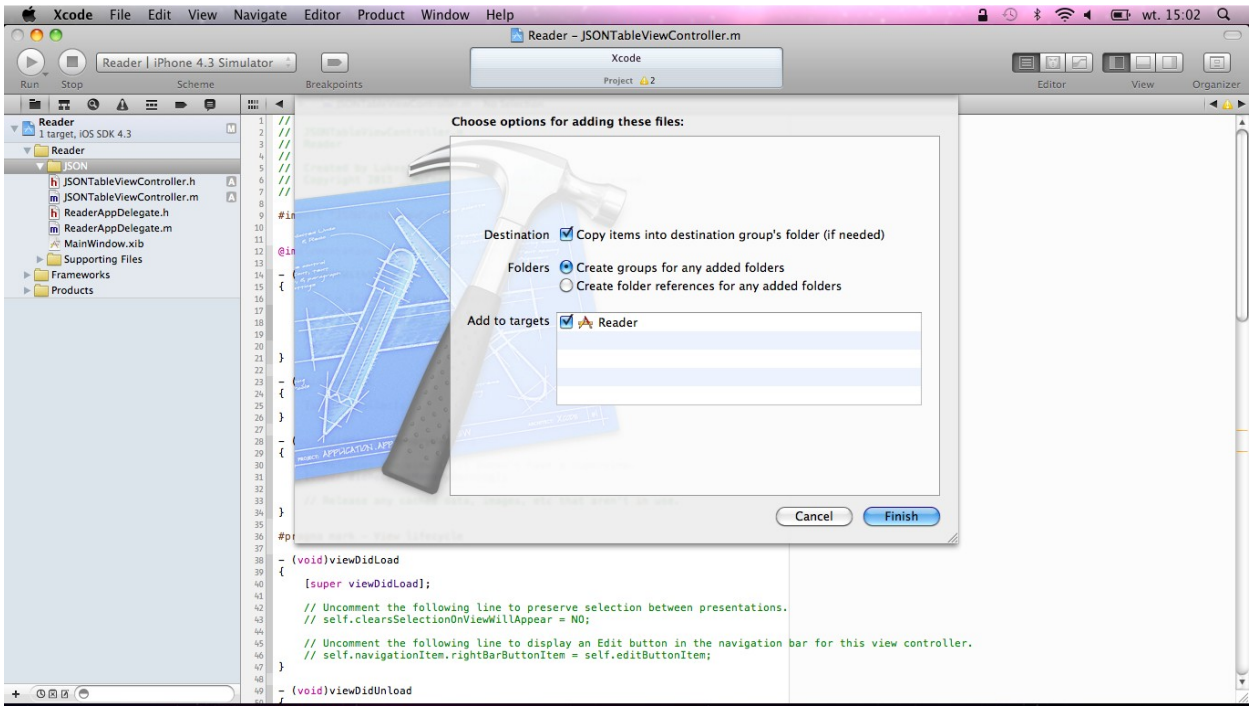
Select the "Reader" group in the project pane, then select "New Group" from the "File" menu. Xcode will create an empty group, which you should name "JSON".



Next, drag all of the files from the "classes" folder from the downloaded JSON framework into the JSON group in Xcode:



A window should appear, presenting some options for the new files. Mark the “Copy items into the destination group’s folder (if needed)” checkbox, then click the “Finish” button.



Next, add the following line to “JSONTableViewController.h”:

```
#import "JSON.h"
```

We have successfully added the JSON framework into our "Reader" project. Now we can use it to read the data returned by the eZ Publish REST web service.

Earlier, we created "JSONTableViewController", which will be responsible for fetching the node list, storing the nodes, and presenting each node's name in the table view. In "JSONTableViewController.h", add an instance variable for the storage array:

```
@interface JSONTableViewController : UITableViewController {
    NSMutableArray *nodes;
}
@end
```

Next, in "JSONTableViewController.m", override the designated initializer "initWithStyle:" to instantiate the nodes.

```
- (id)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        nodes = [[NSMutableArray alloc] init];

        self.title = @"Reader";
    }
    return self;
}
```

Notice that we have also set the title for this controller to "Reader".

Since we are going to operate on nodes, we need to create a business object that will hold node-related information. Select the "Reader" group from the left pane, then select "New File..." from the "File" menu. Choose "Objective-C class", then click the "Next" button. Make sure that the newly created class is a subclass of "NSObject", then click the "Next" button. Give it the name "Node". This class will be a definition for our node business objects.

Add the following line to "JSONTableViewController.h":

```
#import "Node.h"
```

Next we need to implement the data source methods to return cells displaying text strings:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:
```

```
(NSInteger)section
{
    // Return the number of rows in the section.
    return [nodes count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
{
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:CellIdentifier] autorelease];
    }

    Node *node = [nodes objectAtIndex:[indexPath row]];

    [[cell textLabel] setText:node.name];

    return cell;
}
```

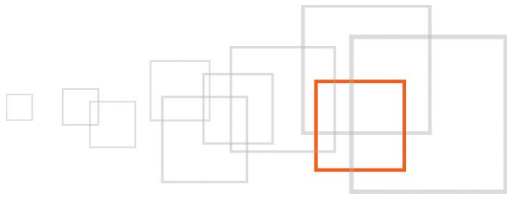
Build and run the application. You should see an empty table view on the screen.

Now we are going to fetch some data from the eZ Publish REST web service. To help with this, there are three classes: “NSURL”, “NSURLRequest” and “NSURLConnection”. Each of these classes has an important role in communicating with a web server.

Open “JSONTableViewController.h” and add two instance variables and a method declaration:

```
@interface JSONTableViewController : UITableViewController {
    NSMutableArray *nodes;
    NSMutableData *jsonData;
    NSURLConnection *connectionInProgress;
}

- (void)loadData;

@end
```

“NSURLConnection” instances can communicate with a web server in two ways: synchronously and asynchronously. We will perform an asynchronous connection.



In "JSONTableViewController.m", we need to implement the "loadData" method to create an "NSURLRequest" within which we will make a connection to the eZ Publish REST web service. "NSURLRequest" will poll http://www.example.com/api/ezp/content/node/2/list for information about the child nodes of node 2 in JSON format, using "NSURLConnection" to make the connection.

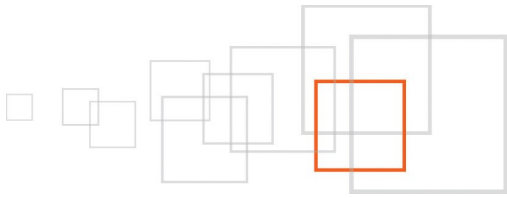
```
- (void)loadData
{
    [nodes removeAllObjects];
    [[self tableView] reloadData];

    NSURL *url = [NSURL
URLWithString:@"http://www.example.com/api/ezp/content/node/2/list"];

    NSURLRequest *request = [NSURLRequest requestWithURL:url
                                            cachePolicy:NSURLRequestReloadIgnoringCach
eData
                                        timeoutInterval:30];
    if (connectionInProgress)
    {
        [connectionInProgress cancel];
        [connectionInProgress release];
    }

    [jsonData release];
    jsonData = [[NSMutableData alloc] init];

    connectionInProgress = [[NSURLConnection alloc] initWithRequest:request
                                                           delegate:self
                                                   startImmediately:YES];
}
```

We want the connection to be made whenever the "JSONTableViewController" table view appears on the screen; to do this, we will override "viewWillAppear" in "JSONTableViewController.m"

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self loadData];
}
```

Build the application to make sure there are no syntax errors. We shouldn't see any data on the screen yet, as we need to implement some of the delegate methods for "NSURLConnection" to actually use the JSON data returned from the request.

Implement the following method in "JSONTableViewController.m" to put all of the data received by the



connection into the instance variable "jsonData".

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [jsonData appendData:data];
}
```

When a connection has finished retrieving all of the data from a web service, it sends the message "connectionDidFinishLoading:" to its delegate. Implement this method in "JSONTableViewController.m" in order to translate JSON data into node business objects.

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSString *jsonString = [[[NSString alloc] initWithData:jsonData
                                                  encoding:NSUTF8StringEncoding]
autorelease];

    for (NSDictionary *childNode in [[jsonString JSONValue]
objectForKey:@"childrenNodes"])
    {
        Node *node = [[Node alloc] init];
        node.id = [[childNode objectForKey:@"nodeId"] intValue];
        node.name = [childNode objectForKey:@"objectName"];

        [nodes addObject:node];
        [node release];
    }

    [[self tableView] reloadData];
}
```

Build and run the application. The table should now display the node names received from the web service.

Now let's implement a view to display actual node content. First, we need to create a new "DetailViewController". Select "New" -> "New File..." from the "File" menu. Next, choose "UIViewController subclass" and click the "Next" button. On the next screen, verify that the "Subclass of" field contains the "UIViewController" class, and that the "With XIB for user interface" checkbox is marked. Then, click the "Next" button and name the view "DetailViewController.m".

Our detail view requires some additional information passed from the table list view. We will implement the custom object initializer "initWithNode:" and pass a Node object as an argument.

Add the following node instance variable and custom object initalizer method to "DetailViewController.h":

```
#import "Node.h"

@interface DetailViewController : UIViewController {
```

```
    Node *node;
}

@property (nonatomic, retain) Node *node;

- (id)initWithNode:(Node *)object;

@end
```

Next, add the object initializer implementation by replacing the "initWithNibName:bundle:" method with the following in "DetailViewController.m". Notice the "self.title" parameter, which will correspond to the title of the browsing window.

```
- (id)initWithNode:(Node *)object
{
    self = [self initWithNibName:@"DetailViewController" bundle:nil];
    if (self) {
        self.node = object;
        self.title = object.name;
    }
    return self;
}

- (void)dealloc
{
    [node release];

    [super dealloc];
}
```

Then, add the following code to "JSONTableViewController.h":

```
#import "DetailViewController.h"
```

Next, we need to handle the action when a table view row is selected. This action is handled by the table view delegate method "tableView:didSelectRowAtIndexPath:":

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath
{
    Node *node = [nodes objectAtIndex:[indexPath row]];

    DetailViewController *detailViewController = [[DetailViewController alloc]
initWithNode:node];
```

```
    [self.navigationController pushViewController:detailViewController animated:YES];
    [detailViewController release];
}
```

Build and run application. When you tap on a table row, the detail view should appear. Since navigation is handled by "UINavigationController", you can tap the "Reader" button to go back to the table list view. Recall that "Reader" comes from the value of "self.title" defined a bit earlier in the "DetailViewController.m" object initializer implementation.

Notice how the title bar for the detail view changes according to the selected node item from the table list view.

5.1.2 *Populating the detail view*

The next step is to implement the communication functions to get the actual content for a selected node. We will re-use the same concept and code for an asynchronous connection to the eZ Publish REST web service.

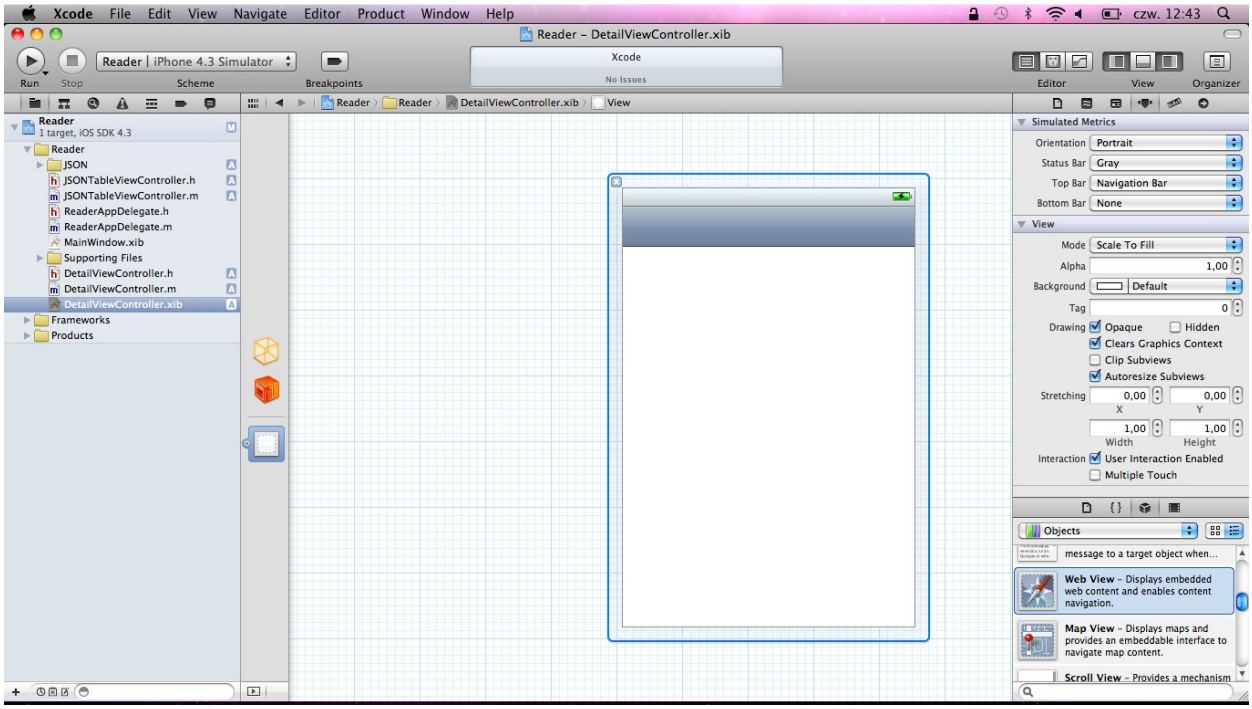
Open DetailViewController.h and add three instance variables and a method declaration:

```
@interface DetailViewController : UIViewController {
```

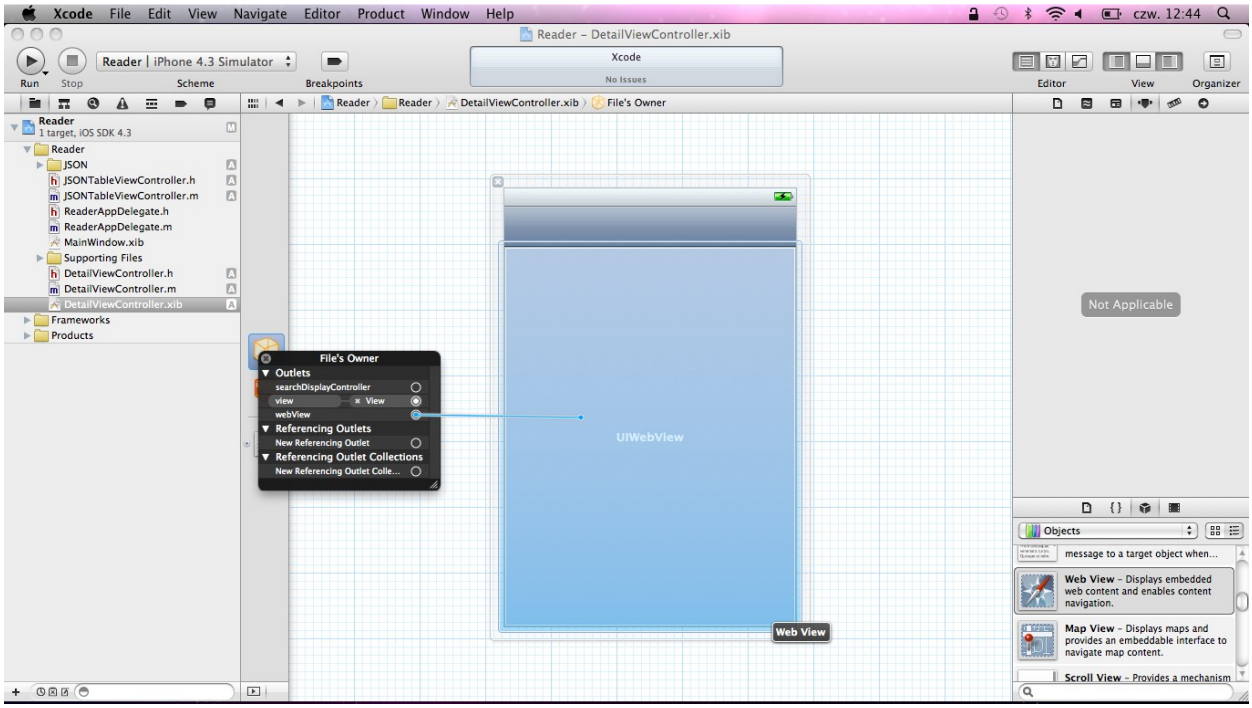
```
    Node *node;
    IBOutlet UIWebView *webView;
    NSMutableData *jsonData;
    NSURLConnection *connectionInProgress;
}

@property (nonatomic, retain) Node *node;

- (id)initWithNode:(Node *)object;
- (void)loadData;

@end
```

"IBOutlet" in front of the instance variable should clue you into the fact that we are going to use the Interface Builder to lay out the interface for the "DetailViewController" view. When we created "DetailViewController", an XIB file of the same name was created and added to the project. Open "DetailViewController.xib" now.

In the right pane, find the "UIWebView" control in the object library and drag it into the view.

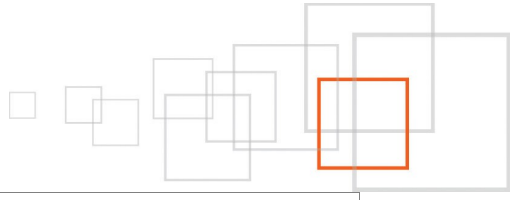


Next, make a connection from the file's owner to that object, as shown below:



In "DetailViewController.m", we need to implement the "loadData" method to create an "NSURLRequest" to connect using the eZ Publish REST API. "NSURLRequest" will poll http://www.example.com/api/ezp/content/node/NODE_ID?OutputFormat=xhtml for information about the node NODE_ID in JSON format. The "NSURLConnection" instance makes the connection. (Note that the "xhtml" value for the "OutputFormat" GET parameter corresponds to the format of the value of the "renderedOutput" element, not the format of the returned data as a whole; see this page for more information.)

```
- (void)loadData
{
    NSURL *url = [NSURL URLWithString:[NSString
stringWithFormat:@"http://www.example.com/api/ezp/content/node/%i?OutputFormat=xhtml",
node.id]];

    NSURLRequest *request = [NSURLRequest requestWithURL:url
                                             cachePolicy:NSURLRequestReloadIgnoringCach
eData
                                         timeoutInterval:30];
    if (connectionInProgress)
    {
        [connectionInProgress cancel];
        [connectionInProgress release];
    }
```



```
    [jsonData release];
    jsonData = [[NSMutableData alloc] init];

    connectionInProgress = [[NSURLConnection alloc] initWithRequest:request
                                                           delegate:self
                                                   startImmediately:YES];
}
```

Override “viewWillAppear:” in “DetailViewController.m” to load the data whenever the “DetailViewController” view appears:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self loadData];
}
```

Implement the following method in “DetailViewController.m” to put all of the data received by the connection into the instance variable “jsonData”:

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data
{
    [jsonData appendData:data];
}
```

As before for the table list view, implement the “connectionDidFinishLoading:” method in “DetailViewController.m”. In this case, we will get the XHTML output for the requested node and pass it to the web view for rendering.

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSString *jsonString = [[[NSString alloc] initWithData:jsonData
                                                  encoding:NSUTF8StringEncoding]
autorelease];

    [webView loadHTMLString:[[jsonString JSONValue] objectForKey:@"renderedOutput"]
baseURL:nil];
}
```

When the “DetailViewController” view gets unloaded, its subviews will still be retained by “DetailViewController”. They need to be released and set to “nil” in “viewDidUnload” by overriding this method in “DetailViewController.m”:

```
- (void)viewDidUnload {
   [super viewDidUnload];
```

```
    [webView release];
    webView = nil;
}
```

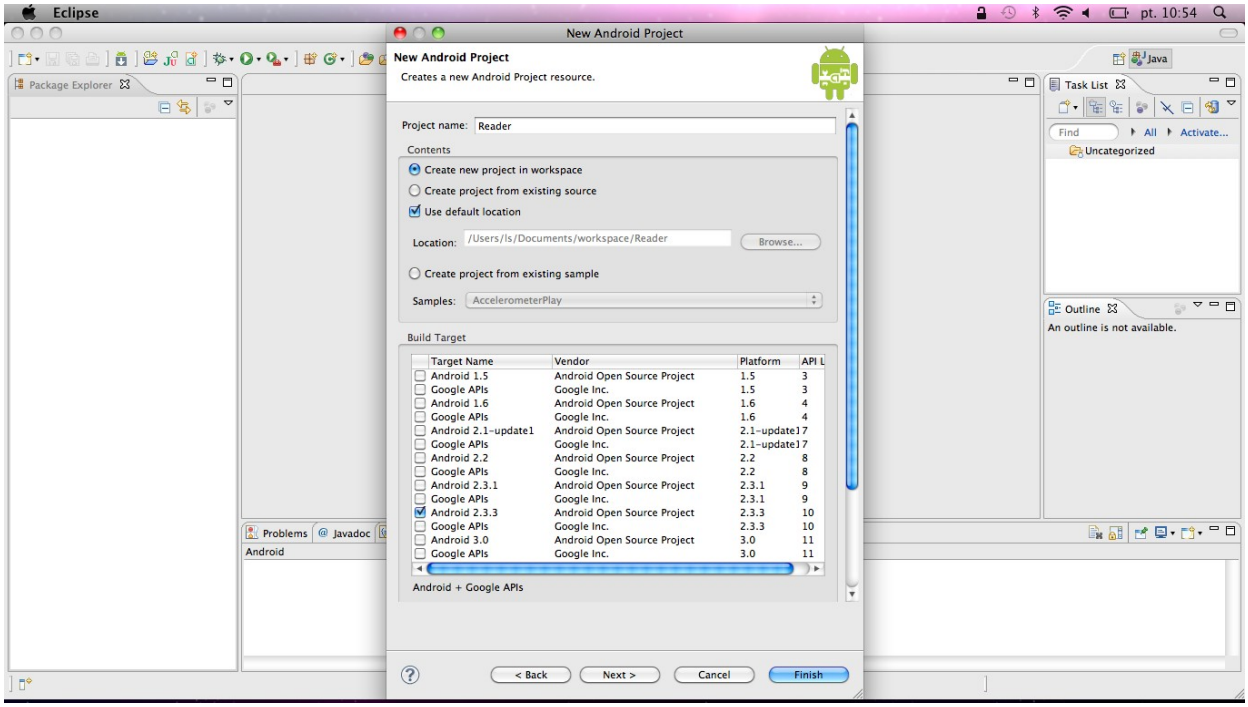
Finally, we need to update the "dealloc" method, which frees up the object's memory and disposes of any resources it holds, including ownership of any object instance variables:

```
- (void)dealloc {
    [node release]
    [webView release];
    [jsonData release];
    [connectionInProgress release];

    [super dealloc];
}
```

That's all! Build and run the application to verify that everything works.

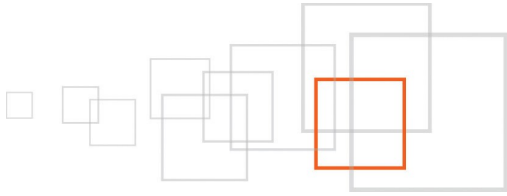
5.2 Android sample application

Open Eclipse and select "New -> Android Project" from the "File" menu.



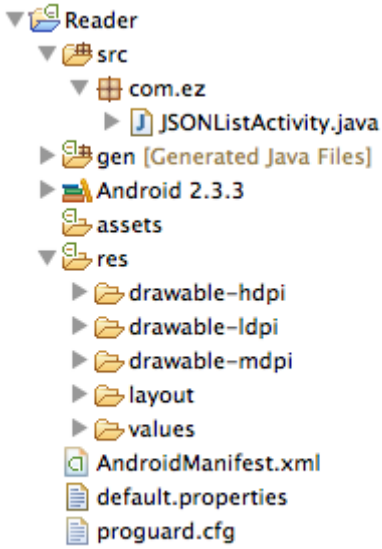
Enter the following information for the project:

- Project name: "Reader"

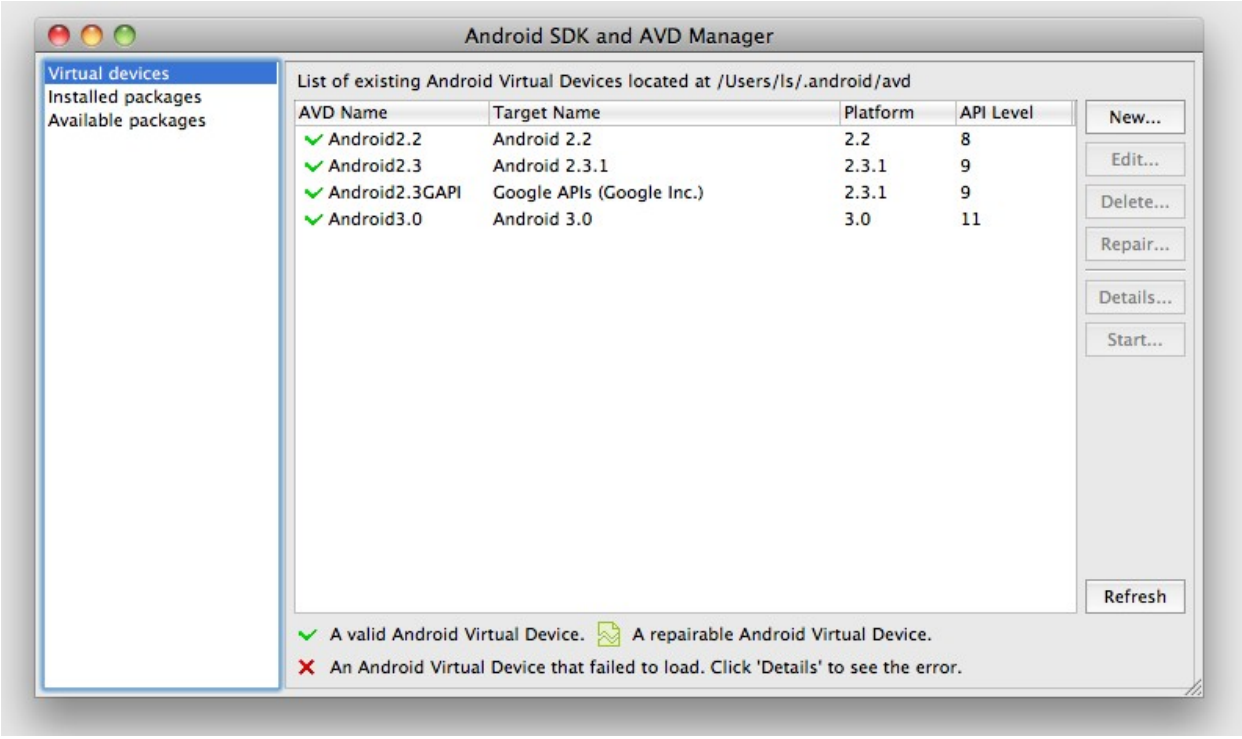


- Build target: “Android 2.3.3”
- Application name: “Reader”
- Package name: “com.ez”
- Create activity: “JSONListActivity” (and make sure that the checkbox is marked)
- Min SDK version: “10”

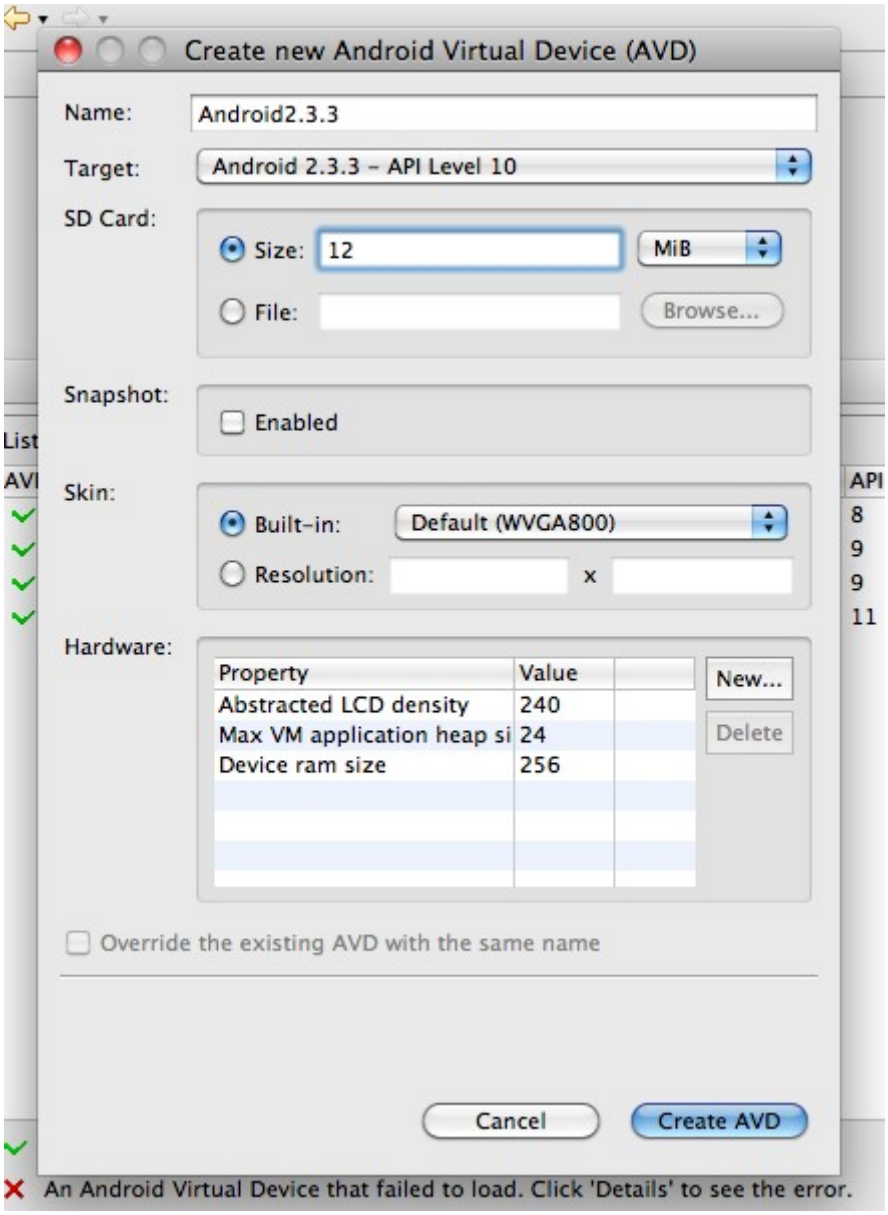
Then, click the “Finish” button. You can inspect the project structure in the left pane.



Next, select “Window” -> “Android SDK and AVD Manager”. In the resulting dialog, select “Virtual devices” from the left panel and click the “New…” button.



Enter a name for your device, and choose an SDK target and screen resolution. Set the SD Card size to 12MiB (this is just to specify a card size for the Android emulator).



5.2.1 *Setting up the main view*

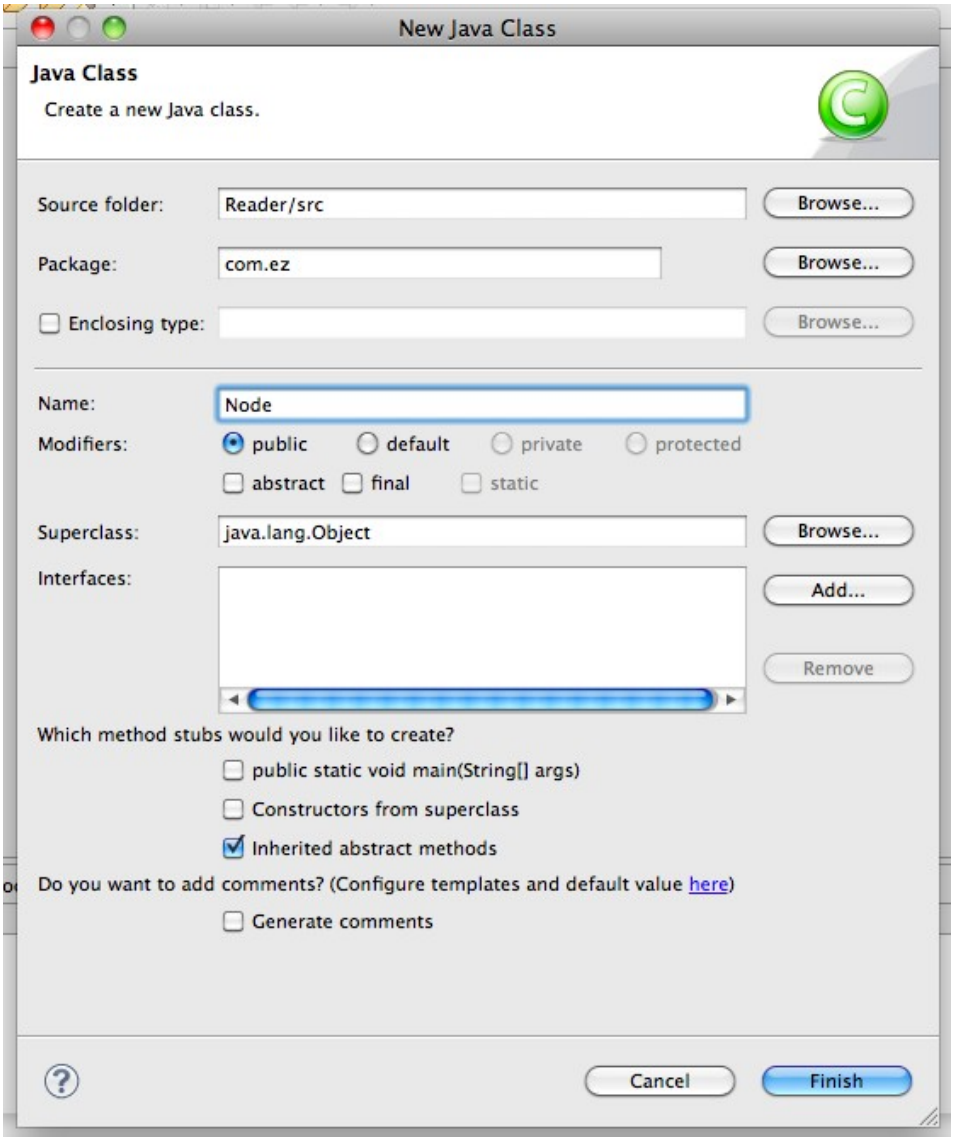
Our application will present multiple screens of information, one with a list view and the second with a details view. The Android SDK provides a convenient way to quickly display a list of data using a superclass called "android.app.ListActivity". This activity already provides a list and content view, ready to use and populate with data.

Open "JSONListActivity.java" and change the superclass of "JSONListActivity" to extend "ListActivity", removing the setting of the "ContentView" from the "onCreate" method.

```
public class JSONListActivity extends ListActivity {
    /** Called when the activity is first created. */
    @Override
```

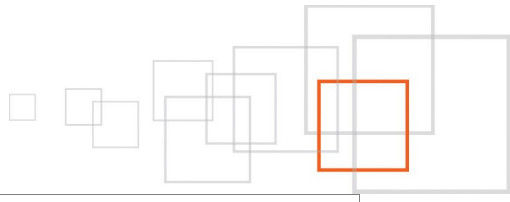
```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

Since we are going to operate on nodes, we need to create a business object that will hold node-related information. Select "New -> Class" from the "File" menu in the "src/com.ez" package. Enter "Node" as the class name, then click the "Finish" button.



Because we are going to pass node objects into the detail view, we need to implement a "java.io.Serializable" interface and implement the "serialVersionUID" property (this is a unique value used to recognize a serialized object; most IDEs can generate this for you, or you can create one yourself):

```
package com.ez;
```

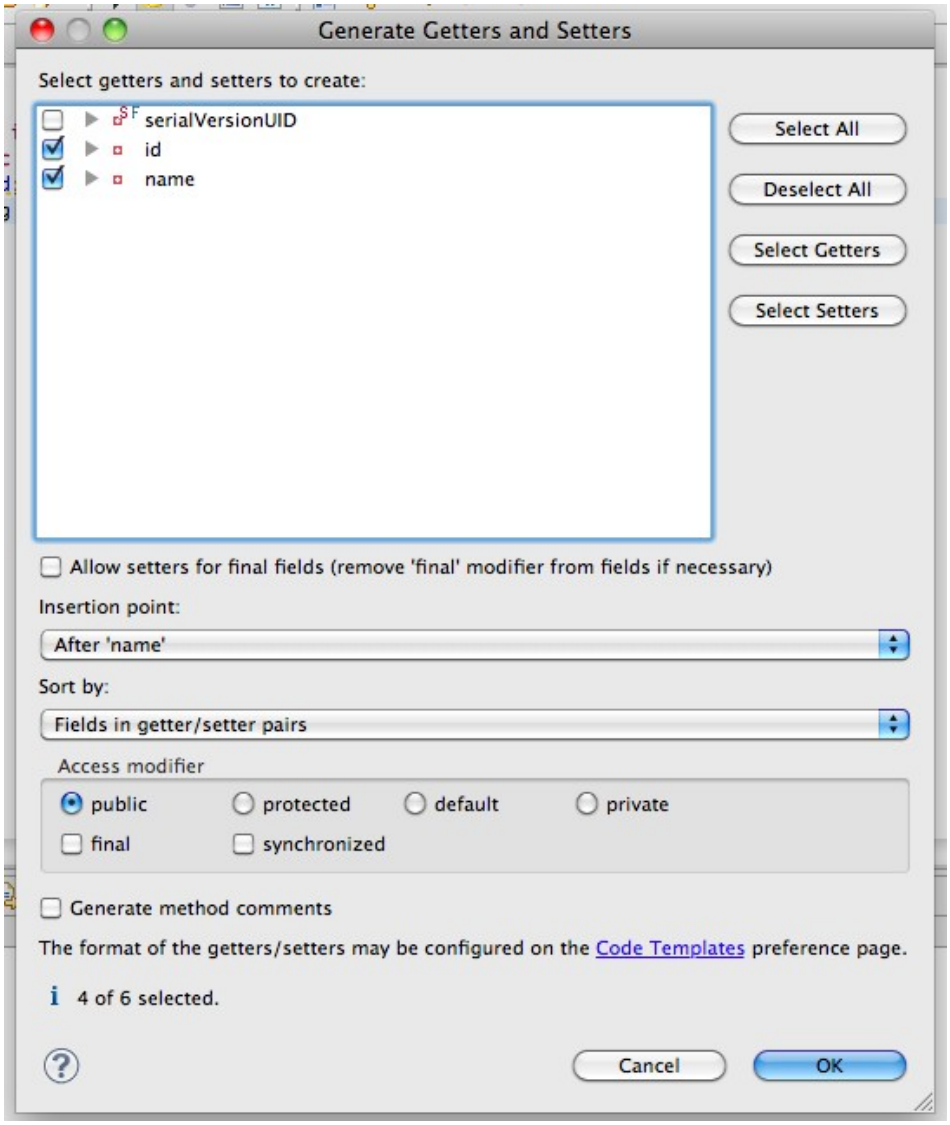


```
public class Node implements java.io.Serializable {
	private static final long serialVersionUID = 432509327611624427L;
}
```

We will store the eZ Publish node ID and object name within the Node business object, thus we need to add two properties:

```
package com.ez;

public class Node implements java.io.Serializable {
	private static final long serialVersionUID = 432509327611624427L;
	private int id;
	private String name;
}
```

Next, we need to add getter and setter methods. Eclipse provides a very convenient way to automate this process. Select “Generate Getters and Setters...” from the “Source” menu. In the resulting window, mark the checkboxes for the “id” and “name” properties, then click the “OK” button.



Finally, we need to implement an override of the "toString()" method, which is going to be used to display the node name in the list view:

```
package com.ez;

public class Node implements java.io.Serializable {
    private static final long serialVersionUID = 432509327611624427L;
    private int id;
    private String name;

    public void setId(int id) {
        this.id = id;
```

```
    }

    public int getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

Now we are going to fetch some data using the eZ Publish REST API. To help with this, we will use the "AsyncTask" class, which provides a simple, convenient mechanism for moving time-consuming operations into a background thread, and provides event handlers to communicate the progress and results of that thread.

However, before we will start implementing "AsyncTask", we need to allow our application to access the Internet connection. Open "AndroidManifest.xml" and add the following line before the closing </manifest> tag:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

To create a new asynchronous task, we need to extend "AsyncTask", but first we need to add a new Java class to our project. Select "New -> Class" from the "File" menu in the "src/com.ez" package. In the resulting window, enter the name "NodeListAsyncTask", then click the "Finish" button. Enter the following as the code for the class:

```
public class NodeListAsyncTask extends AsyncTask<String, Void, Node[]> {
    private ProgressDialog progressDialog;
    private Context context;

    public NodeListAsyncTask(Context context) {
        this.context = context;
    }

    @Override
```

```
    protected void onPreExecute() {
        super.onPreExecute();

        // [... Display progress bar, or other UI element ...]
    }

    @Override
    protected Node[] doInBackground(String... requestURL) {
        Node[] nodes = null;

        // [... Perform background processing task ...]

        return nodes;
    }

    @Override
    protected void onPostExecute(Node[] result) {
        super.onPostExecute(result);

        // [... Report results via UI update, Dialog or notification ...]
    }
}
```

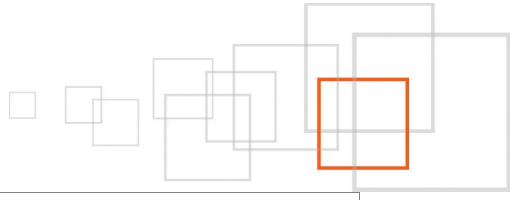
An "AsyncTask" object takes a string as an input parameter: this will be the eZ Publish REST resource URL. For the other parameters, we will use "Void" for progress reporting and a node array for the execution result. In the constructor, we will pass the context in which the asynchronous task was performed.

In the "onPreExecute()" method, we will display a progress bar to the user:

```
@Override
protected void onPreExecute() {
    super.onPreExecute();

    this.progressDialog = ProgressDialog.show(this.context, "Please wait ...",
            "Acquiring data", true);
}
```

In the "doInBackground()" method, we will implement the actual connection to the eZ Publish REST web service and parse the JSON response, translating it into business objects:

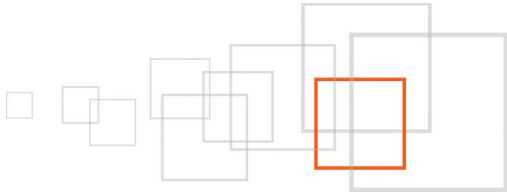
```
@Override
protected Node[] doInBackground(String... requestURL) {
```



```
    Node[] nodes = null;
    String requestString = requestURL[0];

    try {
        DefaultHttpClient httpClient = new DefaultHttpClient();
        HttpGet request = new HttpGet(requestString);
        HttpResponse response = httpClient.execute(request);

        String result = EntityUtils.toString(response.getEntity());
        JSONObject root = new JSONObject(result);
        JSONArray childrenNodes = root.getJSONArray("childrenNodes");

        nodes = new Node[childrenNodes.length()];

        for (int i = 0; i < childrenNodes.length(); i++) {
            JSONObject childNode = childrenNodes.getJSONObject(i);

            Node node = new Node();
            node.setId(childNode.getInt("nodeId"));
            node.setName(childNode.getString("objectName"));
            nodes[i] = node;
        }

    } catch(Exception e) {
        e.printStackTrace();
    }

    return nodes;
}
```

In the "onPostExecute()" method, we will dismiss the progress dialog and notify the view that data processing is finished:

```
@Override
protected void onPostExecute(Node[] result) {
    super.onPostExecute(result);

    this.progressDialog.dismiss();

    ((JSONListActivity)this.context).onNodeListResult(result);
}
```

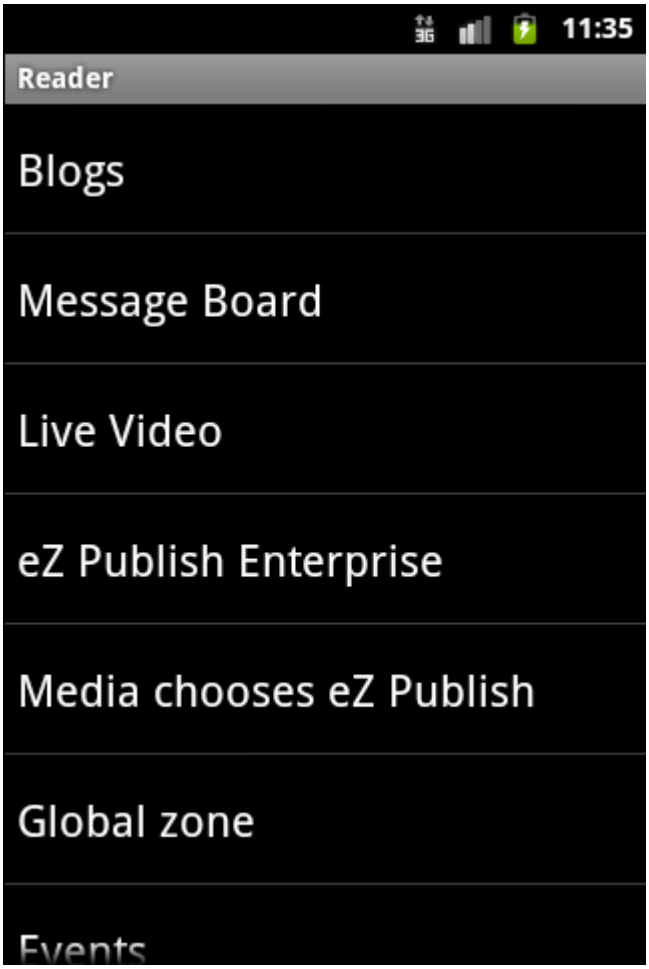


Now that data access is implemented, we need to update the view upon a finished asynchronous task by using the following methods in "JSONListActivity.java":

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);


    new
NodeListAsyncTask(this).execute("http://www.example.com/api/ezp/content/node/2/list");
}


public void onNodeListResult(Node[] nodes) {
    this.setListAdapter(new ArrayAdapter<Node>(this,
android.R.layout.simple_list_item_1, nodes));
}
```

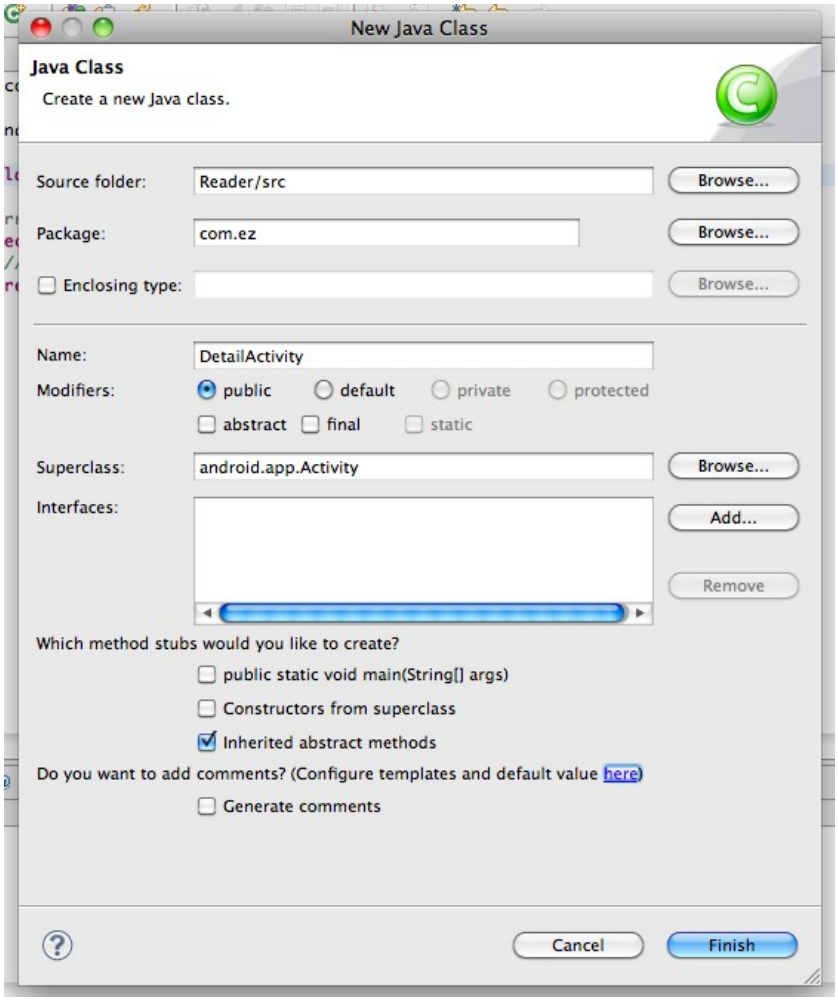
Notice that we are executing "NodeListAsyncTask" by creating a new instance of it with the context parameter, and calling its "execute()" method with the eZ Publish REST URI resource (which should return information about the children of the node with an ID of 2).

Build and run the application. You should see the following result:



5.2.2 *Populating the detail view*

Next, we are going to implement the full view of a page, shown when users tap a row in the table list view. In order to do so, we need to create a new activity class. Select “New -> Class” from the “File” menu in the “src/com.ez” package. In the resulting window, enter the name “DetailActivity” and the superclass “android.app.Activity”. Then, click the “Finish” button.

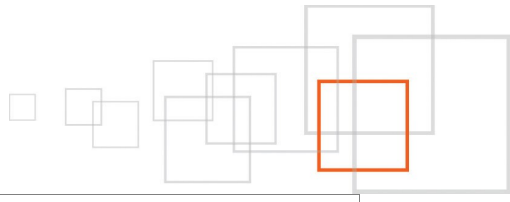


Once a new activity class is created, we need to register it in the application. In order to do so, open "AndroidManifest.xml" and paste the following code before the </application> closing tag:

```
<activity android:name="DetailActivity"
          android:screenOrientation="portrait"/>
```

Next, we need to create a layout file for "DetailActivity". In "res/layout", create a "detail.xml" file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <WebView
        android:id="@+id/webview"
        android:layout_width="fill_parent"
```



```
        android:layout_height="fill_parent"
    />
</LinearLayout>
```

Then, we need to implement the "onCreate" method, which is called when the activity is first created. In "DetailActivity.java", add the following:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.detail);
}
```

Now that we have implemented the base code for "DetailActivity", we can load it whenever the user taps on a table list view row. Implement the following method in "JSONListActivity.java":

```
@Override
public void onListItemClick(ListView l, View v, int position, long id) {
    super.onListItemClick(l, v, position, id);

    Intent detailIntent = new Intent(v.getContext(), DetailActivity.class);
    detailIntent.putExtra("Node", (Node)this.getListAdapter().getItem(position));
    startActivityForResult(detailIntent, 0);
}
```

Build and run application to verify that there are no code errors. The detail view does nothing at the moment. We need to implement data access similar to what we did for the table list view by using "AsyncTask". Select "New -> Class" from the "File" menu in the "src/com.ez" package. In the resulting window, enter the name "NodeAsyncTask", and the superclass "android.os.AsyncTask<String, Void, String>, then click the "Finish" button. Paste the following code:

```
package com.ez;

import android.os.AsyncTask;

public class NodeAsyncTask extends AsyncTask<String, Void, String> {

}
```

We then need to implement three familiar methods: "onPreExecute()" (which shows a progress dialog), "doInBackground()" (which grabs the data), and "onPostExecute()" (which clears the progress dialog and passes the data result to "DetailActivity").

```
public class NodeAsyncTask extends AsyncTask<String, Void, String> {
    private ProgressDialog progressDialog;
```

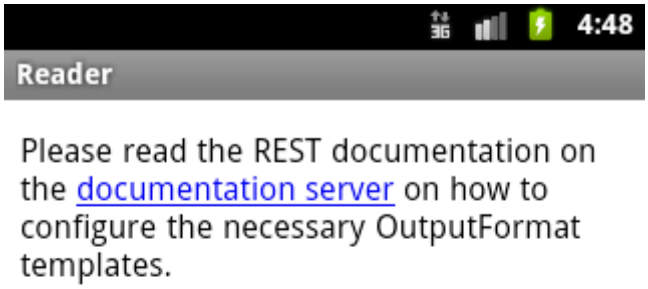
```
private Context context;

public NodeAsyncTask(Context context) {
    this.context = context;
}

  @Override
  protected void onPreExecute() {
        super.onPreExecute();

        this.progressDialog = ProgressDialog.show(this.context, "Please wait ...",
                    "Acquiring data", true);
  }

  @Override
  protected String doInBackground(String... requestURL) {

    String renderedOutput = null;
    String requestString = requestURL[0];

    try {
        DefaultHttpClient httpClient = new DefaultHttpClient();
        HttpGet request = new HttpGet(requestString);
        HttpResponse response = httpClient.execute(request);

        String result = EntityUtils.toString(response.getEntity());
        JSONObject root = new JSONObject(result);

        renderedOutput = root.getString("renderedOutput");
    } catch(Exception e) {
        e.printStackTrace();
    }

        return renderedOutput;
  }

  @Override
  protected void onPostExecute(String result) {
        super.onPostExecute(result);
```

```
            this.progressDialog.dismiss();

            ((DetailActivity)this.context).onNodeResult(result);
        }
}
```

Now that we have implemented "AsyncTask" for getting detailed node data, it is time to use it. Add the following code into "DetailActivity.java":

```
public class DetailActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.detail);

        Node node = (Node)getIntent().getExtras().get("Node");

        new
NodeAsyncTask(this).execute(String.format("http://www.example.com/api/ezp/content/node/
%d?OutputFormat=xhtml", node.getId()));
    }

    public void onNodeResult(String result) {
        WebView webview = (WebView) findViewById(R.id.webview);
        webview.loadData(result, "text/html", "utf-8");
    }
}
```

Build and run the application. Test the detail view by tapping on a row in the table list view. We should see the following output.

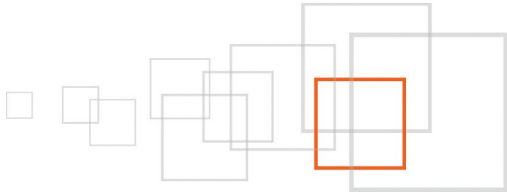


6 Resources

- Sample code for the iOS and Android applications in this article
- eZ Publish REST API documentation
- JSON parser written in Objective-C
- Android Developer Center
- iOS Developer Center
- Building mobile and hybrid applications with eZ Publish

7 About the author : Łukasz Serwatka

Łukasz joined eZ Systems in March 2005. During his 8 years of experience with eZ Publish he created many eZ Publish based solutions and projects. He is also an author of many print and online publications about eZ Publish, as well as eZ Community supporter. He recently became a Mobile Lead Engineer at eZ Engineering,



focusing on the mobile solutions dedicated to eZ Publish sharing his 2 years experience gained in the native applications development.

Reviewers

We would to particularly thank Peter Keung for his thorough review, and Nicolas Pastorino for the last review and web publishing.

8 License choice