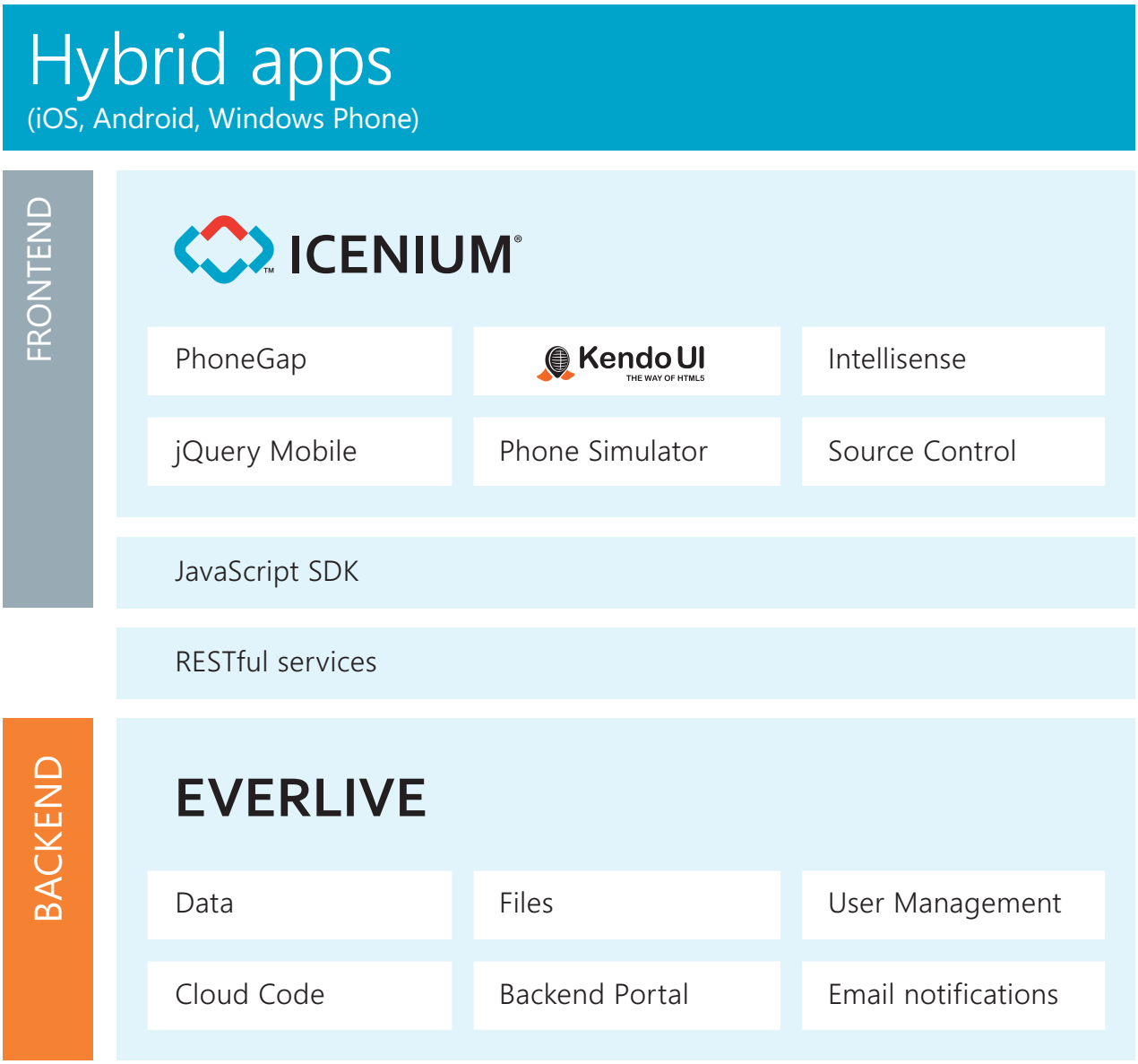
Build cross-platform mobile applications

with Telerik tools

A publication of telerik

Telerik provides a set of tools for creating cross-platform mobile applications and additionally exposes a robust set of cloud services for storing data. Icenium provides the environment for developing and deploying the application, the Kendo UI Mobile framework provides the means to structure the application and a set of common user interface components, and Icenium Everlive provides the cloud storage for data via a set of REST services.

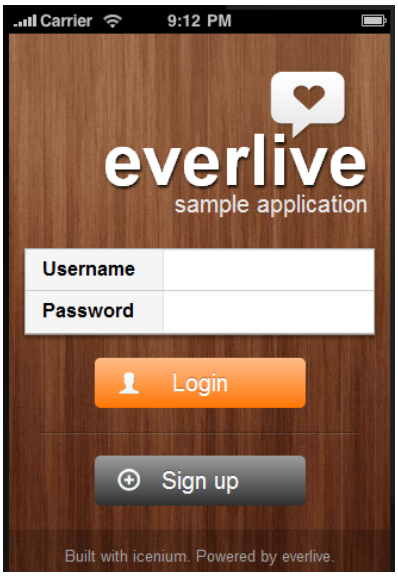
This tutorial will demonstrate how to build a simple application using these three tools.

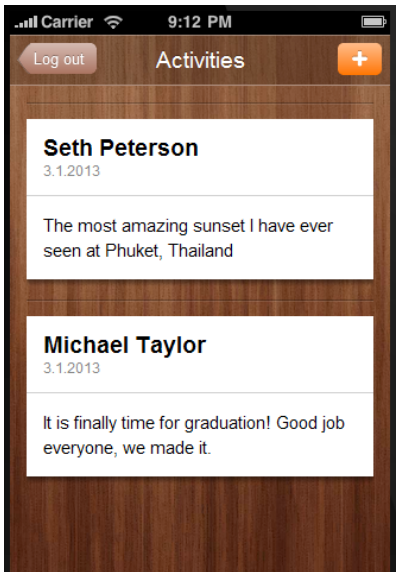


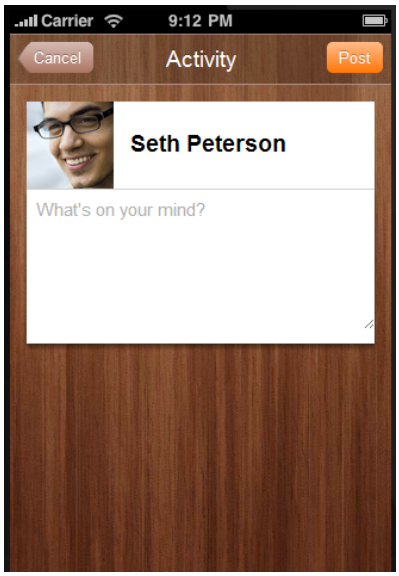
Sample application

TOOLS PROVIDED BY TELERIK

Icenium Everlive provides a sample application that demonstrates the usage of its REST services together with the Kendo UI Mobile framework. The application is a social app that allows a user to post short messages and to view messages of other users. The sample is available for download on this address. Here are images of the app:







This tutorial will demonstrate how to build such application using the tools provided by Telerik.

Contents

Required Knowledge	5
Creating an Icenium project	5
Creating the first view	5
Adding the application logic	7
Creating your backend app in Everlive	8
Initializing the Everlive SDK	9
Adding login functionality	10
The First ViewModel	11
Binding the ViewModel	12
Testing the app so far	12
Creating a new content type for storing user activities	13
The Activities view	14
The Activities model	14
Binding the Activities	17

2 HOURs

Approximate time to complete.

Required Knowledge:

A basic understanding of HTML and JavaScript will be helpful,
as well as a general understanding of Kendo UI's Model View ViewModel (MVVM) framework.

We will take advantage of the Kendo UI MVVM framework by separating our app in Views, ViewModels and Models. The advantages of this approach are not the focus of this tutorial; the focus is simply on enabling you to quickly build a mobile app. There are two really good blog posts explaining in detail how to best structure your code with the Kendo UI Mobile framework and provide a general guide to MVVM for JavaScript developers. I recommend getting familiar with these two valuable resources:

http://codingwithspike.wordpress.com/2012/11/30/making-a-well-structured-kendo-ui-mobile-app-in-icenium-with-require-js/

http://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/

Creating an Icenium project

We start our app by creating a new Cross-Platform Device Application (Kendo UI Mobile) in Icenium. This type of project is used because it provides a base project template containing all the resources needed to build an application. The project comes with sample views in index.html and JavaScript code in scripts/hello-world.js, but because we'll write our own views and functionality, they can safely be removed.

Creating the first view

A Kendo UI mobile application needs a Layout situated in the index.html file and at least one View in the same file. Let's open the index.html file and remove all the existing views, the content of the layout and the script that initializes the Kendo UI application; it will be placed in a separate JavaScript file. Also, go ahead and rename the layout to "default" by setting the "data-id" attribute. The body of the HTML document should look like this:

```
    <body>
        <div data-role="layout" data-id="default">
        </div>
    </body>
```

Add a view named "welcome" and point it to the "default" layout:

```
<div data-role="view" id="welcome" data-layout="default">
</div>
```

This view is the initial screen for our application and it needs a login form. It's also nice to have a title for this view. The file should look like this:

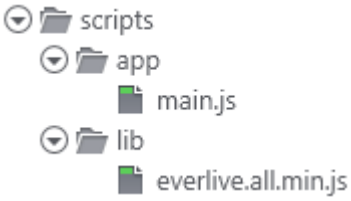
```
<!DOCTYPE html>
<html>
	<head>
		<title></title>
		<meta charset="utf-8" />
		<script src="cordova.js"></script>
		<script src="kendo/js/jquery.min.js"></script>
		<script src="kendo/js/kendo.mobile.min.js"></script>
		<script src="http://maps.google.com/maps/api/js?sensor=true"></script>
		<script src="scripts/hello-world.js"></script>

		<link href="kendo/styles/kendo.mobile.all.min.css" rel="stylesheet" />
		<link href="styles/main.css" rel="stylesheet" />
	</head>
	<body>
		<div data-role="layout" data-id="default">
		</div>
		<div data-role="view" id="welcome" data-layout="default">
			<h1>
				<b>everlive</b> sample application
			</h1>
			<form id="login-form">
				<ul data-role="listview" data-style="inset">
					<li>
						<label for="loginUsername">Username</label>
						<input type="text" id="loginUsername" />
					</li>
					<li>
						<label for="loginPassword">Password</label>
						<input type="password" id="loginPassword" />
					</li>
				</ul>
			</form>
			<div>
				<a data-role="button">Login</a>
			</div>
		</div>
	</body>
</html>
```

Adding the application logic

So far we've only used HTML. To make this a real application, however, some JavaScript code is needed. By default the project comes with a scripts/hello-world.js file and a reference to it in the index.html. We don't need this file, so we'll remove it along with its reference from the index.html. Our script files will still be placed in the "scripts" folder but they will be divided by their role:

- The files containing the necessary libraries, such as the Everlive SDK JavaScript file, will be in the "lib" subfolder.
- The files containing the application logic will be in the "app" subfolder.

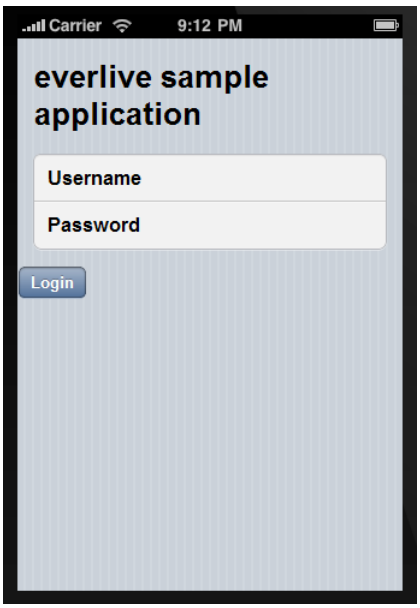
In the "app" subfolder we'll create a new file: main.js. In the "lib" subfolder, we will add a file containing the Everlive JavaScript SDK: everlive.all.min.js. The SDK file can be obtained by navigating to the Everlive backend and opening the Downloads section. Click the "Hybrid & Web" tab and then select the JavaScript SDK for hybrid mobile apps and websites link. Unzip the downloaded file and look for everlive.all.min.js in the "min" folder.



Don't forget to add references to the JavaScript files in the index.html page like this:

```
<script src="scripts/lib/everlive.all.min.js"></script>
<script src="scripts/app/main.js"></script>
```

We'll write the application logic inside the main.js file. In order to keep from polluting the global namespace, all the code is wrapped in an immediately-invoked function expression (IIFE) and a single entry point is provided for our data and functionality through a global variable called "app":




```
var app = (function () {
    return {};
}());
```

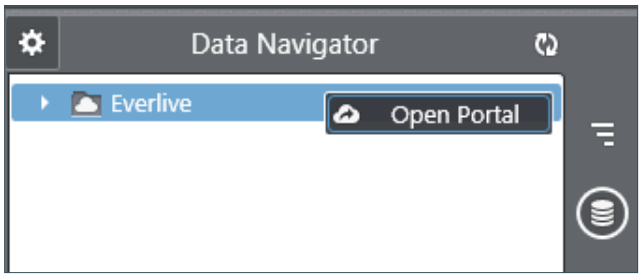
Since the script that initializes the Kendo UI application was removed from the index.html in the previous step we need to add it back here:

```
var mobileApp = new kendo.mobile.Application(document.
body, { transition: 'slide', layout: 'default' });
```

Now if you click the "Run in simulator button" in Icenium, you will see the "welcome" screen.

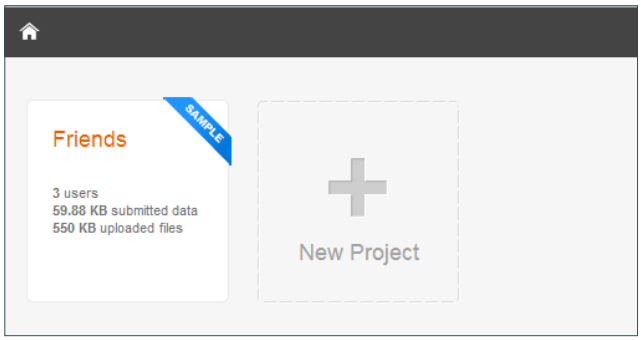
Creating your backend app in Everlive

Our application will use the Icenium Everlive cloud services to store its data. To create the backend for the app, you can either navigate the Everlive Portal at https://www.everlive.com in your browser, or open the Data Navigator pane, right-click Everlive and click Open Portal.

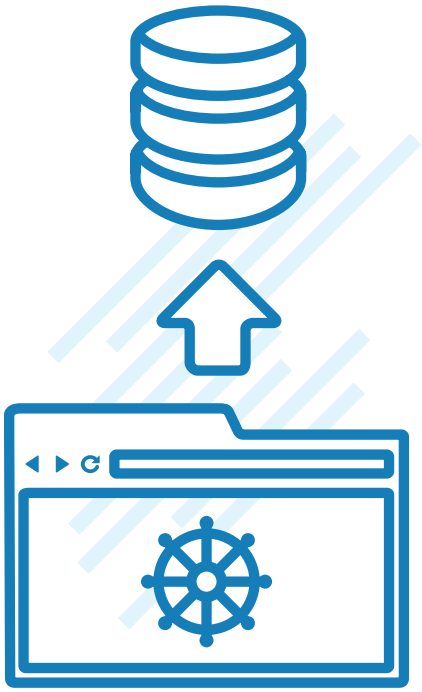


The backend defines the structure of your app in terms of content types and fields. It also lets you explore and manage the generated data from the mobile apps. In Everlive, you can define the following custom fields: Text, Number, DateTime,
Yes/No, File, GeoPoint, Single Relation, Multiple Relation, and Array. Every app also comes with predefined content types enabling you to store your Users, Roles and Files.

By default, you already have an existing Friends sample that you can explore. For the purpose of this article, however, we will go ahead and create a new Everlive app that will serve as our backend.

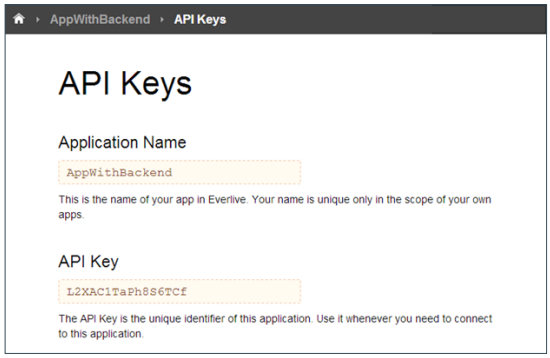


Click on "New Project", and enter a name for your backend app. Select "Start from scratch", and your backend app will be created in seconds.

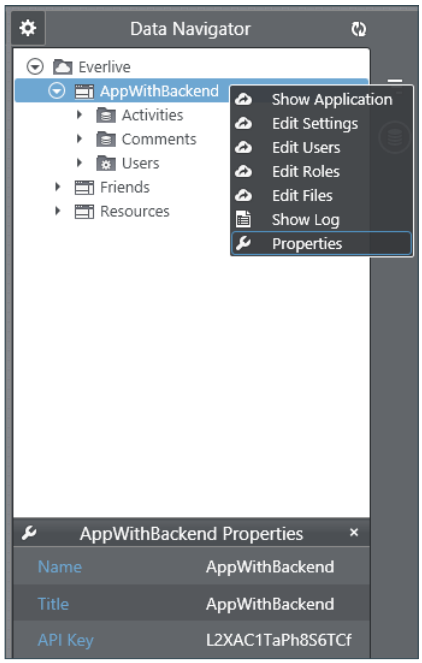


Initializing the Everlive SDK

To connect your Icenium app to the backend services, you have to obtain the API Key of your Everlive app and pass it to the JavaScript SDK so it knows which app to connect to. In order to initialize the Everlive SDK, you have to provide an API key of an existing app in Everlive. Since we already created our first app in Everlive, you can obtain its API Key by navigating to the API Keys section of your app within Everlive. →



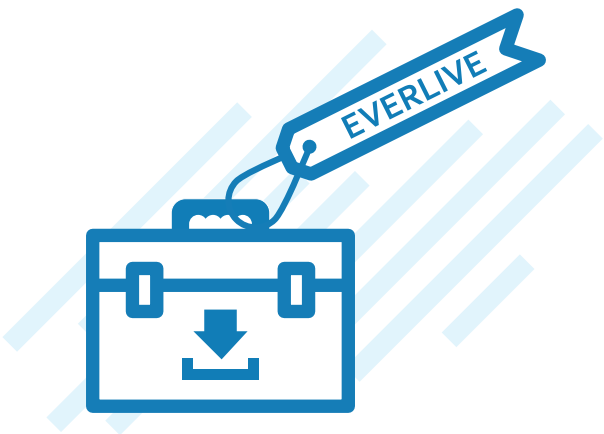
Alternatively, you can obtain the API Key of an app straight from Icenium. You can open the Data Navigator in Icenium, right-click your app, and click Properties. The Properties panel shows the API Key of your app as shown on the right →



Add the following code in the main.js:

```
var el = new Everlive('your API Key');
```

This code initializes the Everlive SDK and sets a reference of it to the "el" variable. The SDK provides convenient ways for querying the cloud data services. If you don't keep a reference you can still access the initialized SDK instance through the Everlive.$ property.



Login functionality

IT IS VERY SIMPLE

Adding login functionality

Every Everlive app has a built-in Users content type which is used to store information about your application's users - like username, password, email, etc. We will use this Users type in our new login functionality.

The "Login" button in the "welcome" view still doesn't do any real work. It needs an event handler that will gather the username and password of a user and send them to the cloud for authentication. The following is the sample code for the event handler:

```
var login = function () {
    var username = $('#loginUsername').val();
    var password = $('#loginPassword').val();
    el.Users.login(username, password)
    .then(function () {
        alert('Authentication successful');
    },
      function (err) {
          alert (err.message);
      }
    );
};
```

It is very simple. It takes the values of the input fields for username and password and sends them to Everlive by calling "el.Users.login". The "el" variable is the reference to the SDK instance and has a predefined property "Users" that provides basic CRUD operations as well some specialized functions for authenticating, registering and logging out a user. You can find more information about the API of the SDK on the Documentation page in the Everlive backend.

Notice that the "login" function is chained with the "then" function, which means that the "login" function returns a promise because callbacks are not provided directly. The code that will be executed when the request is over is defined by passing anonymous functions to the "then" method. You can view this slide show for more information about the concept of promises in JavaScript. If the login succeeds the app displays a popup saying "Authentication successful." If it fails, you will see a popup with an error message.

The First ViewModel

We need a way to expose that event handler to the views. It is a job for a ViewModel and we need to create one. The ViewModel for the "welcome" view is an object that holds a reference to the "login()" event handler. In the main.js file we declare a variable "loginViewModel" to hold the ViewModel and expose it to the application via the "app" global variable:

```
var app = (function () {
       // ...
       var loginViewModel = (function () {
              var login = function () {
                     // ...
              };
              return {
                     login: login
              };
       }());
       return {
        viewModels: {
            login: loginViewModel
        }
    };
}());
```

The "app" variable contains the ViewModels as an associative list named "viewModels." It allows accessing and binding to them in the views.

Binding the ViewModel

The event handler is exposed and it needs to be invoked when someone clicks the "Login" button. In Kendo UI, setting the "data-model" attribute of the view element connects views and ViewModels. In the index.html file we add this attribute to the "welcome" view and set its value to "app.viewModels.login":

```
<div data-role="view" id="welcome" data-layout="default" data-model="app.viewModels.login">
```

A "data-bind" attribute is added to the "Login" button in order to attach the event handler to it:

```
<a data-role="button" data-bind="{click: login}">Login</a>
```

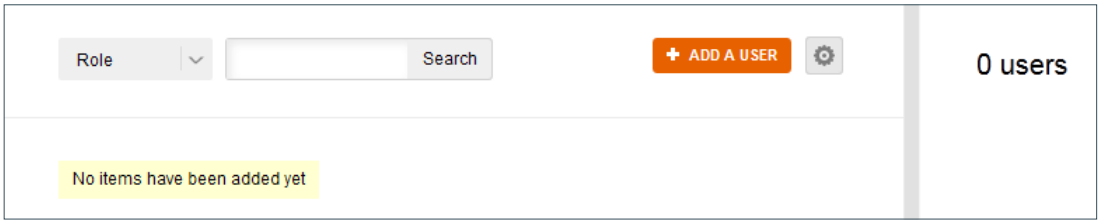
The value of the "data-bind" attribute says that the "data-click" attribute should be added to the element and its event should be the function specified by the "login" property of the ViewModel. It happens to be our authentication function. The code above is equivalent to:

```
<a data-role="button" data-click="app.viewModels.login.login">Login</a>
```

We'll stick to the former way for resolving attributes values.

Testing the app so far

Click the "Run in simulator" button to see how the application looks. Since the application does not have a signup screen we'll create a new user by utilizing the backend of Everlive (if you really want to have a signup screen before ending this post then check out the code of this sample hybrid application). Open the "Data Navigator" tab, expand your app from the treeview, and right-click on the Users type. From the menu select "Show Items." It will open the page with the data of the "Users" content type. After it is opened you will see an empty grid with the ability to add a new user.



Click the "Add a user" button and create a new user, e.g. "John Doe." Now return to the window with the Icenium simulator and type the username and password of the new user. If you click the "Login" button and everything works as expected, you will see a popup saying "Authentication successful." If there is no popup, then check the console of the simulator for any errors.

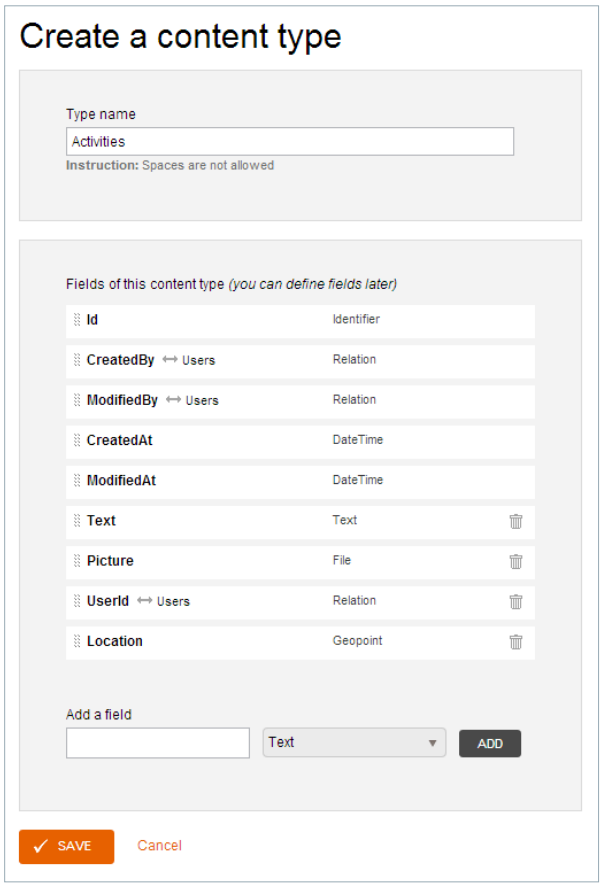
User activities

Creating a new content type for storing user activities

Let's create a new content type called "Activities" that will be used to store check-in information by our users. The Activities content type stores short user posts with a picture. It has four fields: "Text" – the text of the post, "Picture" – the id of the file that is used for a picture, "UserId" – the id of the user that created the activity, and "Location" – a GeoPoint specifying the coordinates where the activity took place. The Activities type also contains system fields that are common for every content type. Fields like CreatedAt, which specify the date and time when an item was created.

Open your app in Everlive, and click on "Create a content type." A dialog for creating a new content type in Everlive will open. Name your new content type "Activities," and enter your 4 custom fields as shown on the right →

Open the content type you have created and enter a couple of activities that we will retrieve in our Icenium app.



The Activities view

We will now add another view to our application that will consume our new Activities content type we just defined in Everlive. We'll place the markup of the new view in a separate html file named "activitiesView.html" and put that file in a new folder "views." Kendo UI does not require the view files to be valid HTML documents so we'll add only the necessary elements to it:

```
<div data-role="view" id="view-all-activities" data-layout="default">
       <header data-role="header">
              <div data-role="navbar">
                     Activities
              </div>
       </header>
       <ul id="activities-listview" data-style="inset" data-role="listview"></ul>
</div>
```

In this view we have a header with a navigation bar and a title. We also have an unordered list that will hold the contents of the activities.

The Activities model

The new view needs a ViewModel from which it can retrieve the list of activities. The ViewModel in turn needs to know what the structure of the activities content type is. This is done by creating a model for an activity. The ViewModel and the model will be placed in the main.js file next to the login ViewModel. The model is defined first:

```
var activitiesModel = (function () {
        var activityModel = {
            id: 'Id',
            CreatedAtFormatted: function () {
                return AppHelper.formatDate(this.get('CreatedAt'));
            },
            User: function () {
                var userId = this.get('UserId');
                var user = $.grep(usersModel.users(), function (e) {
                    return e.Id === userId;
                })[0];
                return user ? {
                    DisplayName: user.DisplayName
                } : {
                    DisplayName: 'Anonymous'
                };
```

```
            }
        };
        var activitiesDataSource = new kendo.data.DataSource({
            type: 'everlive',
            schema: {
                model: activityModel
            },
            transport: {
                // required by Everlive
                typeName: 'Activities'
            },
            sort: { field: 'CreatedAt', dir: 'desc' }
        });
        return {
            activities: activitiesDataSource
        };
    }());
```

What's interesting here is that the model is kept in a Kendo UI data source. Technically speaking, the data source is more related to the ViewModel, but the data that the data source will retrieve from the cloud may be used in more than one ViewModel. A Kendo UI widget like the ListView works great with data sources and does not require additional plumbing.

The "activitiesDataSource" is configured to work with the Everlive REST services by setting "type: 'everlive'" option. It also needs the "schema.model" and "transport.typeName" options to be defined in order to function properly. The "activityModel" that is used as a "model" for the data source has two calculated fields: "CreatedAtFormatted" and "User." We'll use these fields in the activities view to display a more user friendly representation of the fields CreatedAt and UserId. You might notice that the calculated fields use some objects which we haven't yet defined: "AppHelper" and "usersModel." "AppHelper" holds utility functions while the "usersModel" represents the Users data. Here is the missing code:

```
    var AppHelper = {
        formatDate: function (dateString) {
            var date = new Date(dateString);
            var year = date.getFullYear().toString();
            var month = (date.getMonth() + 1).toString();
            var day = date.getDate().toString();
```

```
            return day + '.' + month + '.' + year;
        }
    };

    var usersModel = (function () {
        var usersData;
        var loadUsers = function () {
            return el.Users.get()
            .then(function (data) {
                usersData = new kendo.data.ObservableArray(data.result);
            })
            .then(null,
                  function (err) {
                      alert(err.message);
                  }
            );
        };
        return {
            load: loadUsers,
            users: function () {
                return usersData;
            }
        };
    }());
```

The data of users is kept in an ObservableArray so it can be bound easily to a view. It first needs to be retrieved from the cloud by executing the "loadUsers" function. It can be done after a user authenticates so they are authorized to see the other users in the app. She can be redirected afterwards to the new view. We'll change our "login" function to achieve this functionality:

```
        var login = function () {
            var username = $('#loginUsername').val();
            var password = $('#loginPassword').val();
            el.Users.login(username, password)
            .then(function () {
                return usersModel.load();
            })
            .then(function () {
                mobileApp.navigate('views/activitiesView.html');
            })
            .then(null,
```

```
                    function (err) {
                        alert(err.message);
                    }
                );
            };
```

The ViewModel for the Activities view will only expose the Activities model for now, so it is quite simple:

```
var activitiesViewModel = (function () {
        return {
            activities: activitiesModel.activities
        };
    }());
```

We expose the activitiesViewModel with app.viewModels:

```
return {
        viewModels: {
            login: loginViewModel,
            activities: activitiesViewModel
        }
    };
```

Binding the Activities

Now the activitiesViewModel can be bound to the activitiesView. We have already done the same for the "welcome" view:

```
<div data-role="view" id="view-all-activities" data-layout="default" data-model="app.
viewModels.activities">
```

Next, the activities data source is bound to the ListView widget by using the same "data-bind" attribute we used for the "Login" button. This time the "data-source" attribute of the ListView will be evaluated:

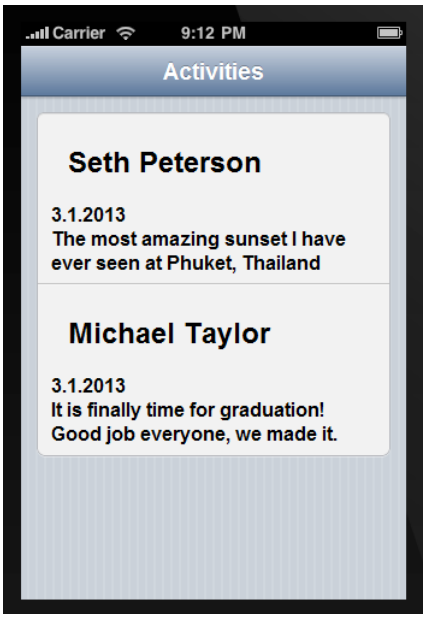
```
<ul id="activities-listview" data-style="inset" data-role="listview" data-
bind="source: activities" data-template="activityTemplate"></ul>
```

That's it!

The "data-template" attribute specifies the HTML that will be generated for each item in the data source and the attribute value is the id of a script element containing the template. You can refer to the Kendo UI demos for more information on templates and how they work. The following is the template we'll use for this ListView:

```
<script type="text/x-kendo-template" id="activityTemplate">
    <div>
        <div>
            <h2>${User().DisplayName}</h2>
            <time class="timeSpan">${CreatedAtFormatted()}</time>
        </div>
        <div>${Text}</div>
    </div>
</script>
```

It shows the DisplayName of the user that posted the activity, the time the activity is created and the text of the activity. That's it! If you click "Run in simulator" you should be able to login with the user we created earlier and see a list of predefined activities.



What's Next?

What we've built is a small mobile app with two screens. It lacks:

- Detailed views of our master view (the list with activities).
- Screens for registering new users and adding activities.
- CSS styling and images.
- A global error handler so that if an unexpected error occurs, the app does not break. Every mobile app built with the Kendo UI should have this global error handler.
- A touch widget that supports tap events. Using the click event in a mobile app is not very responsive. Kendo UI provides a Touch widget that supports tap events.

The complete app featuring all of the above functionality is available in Icenium accessible by simply creating a new project based on the Icenium Everlive Cloud Services template. Your resulting new project will feature a fully developed social app called Friends.

Also, be sure to check the comprehensive documentation of the Everlive SDK by visiting the Everlive backend of your application.

ABOUT THE AUTHOR

Ivan Pelovski

Ivan Pelovski is a member of the Everlive team. He is a developer with experience in building JavaScript and .NET applications. He likes to have fun and learn new things while he's on his quest to write perfect code.

ABOUT TELERIK

Telerik is a market-leading provider of end-to-end solutions for application development, automated testing, agile project management, reporting, and content management. Telerik's award-winning software development products enable enterprises and organizations of every size to generate tangible productivity gains, reduce time-to-market, and stay on time and under budget. With tens of thousands of users in more than 90 countries around the world, Telerik's customers include numerous Fortune 2000 companies, academic institutions, governments, and non-profit organizations.