
Build an Ajax application with the Dojo Toolkit

Skill Level: Intermediate

[Joe Lennon](#)

Product Manager

Core International

01 Mar 2011

The Dojo toolkit is a JavaScript library that makes the process of building large JavaScript-based Rich Internet Applications (RIAs) much simpler. With a wide range of features—from DOM querying and manipulation, Asynchronous JavaScript and XML (Ajax) request handling, excellent object-orientation support, and a full user interface widget library (Dijit)—Dojo is an excellent library to use to build a dynamic and interactive web application. In this tutorial, learn about many of the concepts of Dojo and the Dijit widget library through the development of a fully featured sample application, a contact manager system. This application lets a user browse, create, edit, and remove contacts (and contact groups) from a MySQL database. PHP is used on the server side to communicate with the database, with Dojo and the Dijit component library providing a rich, Ajax-powered user interface. The final result is a powerful web application that you can use as a foundation for your own RIAs.

Section 1. Before you start

This tutorial is for web application developers who are interested in utilizing the Dojo Toolkit to create visually impressive RIAs with relative ease. The tutorial guides you through the process of developing a full web application using Dojo, and although you will learn a lot about Dojo through it, it is recommended that you familiarize yourself with the basics of Dojo before proceeding. You should also be familiar with HTML and CSS, and you should be comfortable with JavaScript development. You don't need to be an expert, but the more knowledge you have of these topics, the easier the tutorial will be to follow. For a detailed introduction to Dojo, please see [Resources](#) for links to some introductory articles.

About this tutorial

The Dojo toolkit is a powerful JavaScript framework that provides all of the building blocks required to build impressive desktop-style RIAs. At its base, the framework includes various useful features that make the process of writing JavaScript applications easier, including DOM querying and manipulation, effects and animations, Ajax event handling, and more. Where Dojo stands apart from many other libraries, however, is in its one-of-a-kind widget system, Dijit, which includes a powerful parser that lets you use Dijit components as if they were regular HTML elements. Finally, Dojo also includes an extensive suite of extensions in DojoX that provide additional features, including support for a wide range of data stores, data grids, additional Dijit components, and much more.

In this tutorial, you will implement each of these three parts of the toolkit in a sample application that resembles a desktop contact manager application. This application will allow you to manage your contacts by adding new ones as well as editing or deleting existing ones. You can arrange your contacts into groups, moving them between groups as required. The groups themselves can also be added, modified, and deleted. The idea is that this tutorial will show you how to create a fully functional, dynamic, database-driven application with relative simplicity and much less code than you might expect. The application you create in this tutorial can be easily extended or modified to create projects of your own.

Prerequisites

To follow this tutorial, you will need the following:

- A web server that supports PHP, such as Apache
- PHP, version 5 or later
- MySQL, version 4 or later (earlier versions may work)
- Dojo Toolkit 1.5 or later (can be loaded using the Google Content Delivery Network. Alternatively, you can download the toolkit locally and run it from your development machine.)

The PHP scripts and MySQL tables used in this application are extremely straightforward, so they should be relatively easy to convert to other server-side languages such as Java™ code, C#, Python, and so on, and databases such as DB2®, Oracle, and SQL Server. The majority of the work in this application is done by Dojo, with PHP and MySQL simply used for persistence.

See [Resources](#) for links to these tools.

Section 2. Setting up

This tutorial guides you through the development of a feature-rich Dojo web application. This section explains the various concepts of the Dojo Toolkit that are covered in this tutorial as well as giving a tour of the sample application this tutorial will teach you to develop.

Dojo concepts used in this tutorial

As you work through this tutorial, you will be putting into practice various features of the Dojo toolkit. The following sections outline the features from the different parts of the toolkit (Base, Dijit, and Dojox).

Base

At its core, the Dojo Toolkit provides a base set of features common to many JavaScript libraries. These features typically include DOM querying and manipulation functions, features that enable developers to make cross-browser-friendly Ajax requests, event handling, data APIs, and more. In this tutorial, you will use many of the features of Dojo Base, including:

- DOM querying
- DOM manipulation
- `XmlHttpRequests` (XHR/Ajax)
- Event handling (`dojo.connect`)
- Read/Write data stores

Dijit

One of the most distinctive aspects of the Dojo Toolkit is the Dijit component library, which lets you create powerful and visually stunning user interfaces with minimal effort and fuss. What's more, Dijit lets you use JavaScript components in a declarative fashion, thanks to Dojo's powerful widget parsing capabilities. This tutorial implements a wide variety of Dijit components, including:

- Layout components (`BorderContainers` and `ContentPanels`)
- Menus (Application and Context)

- Tree
- Dialogs
- Forms, ValidationTextBoxes, TextBoxes, FilteringSelects, and Buttons

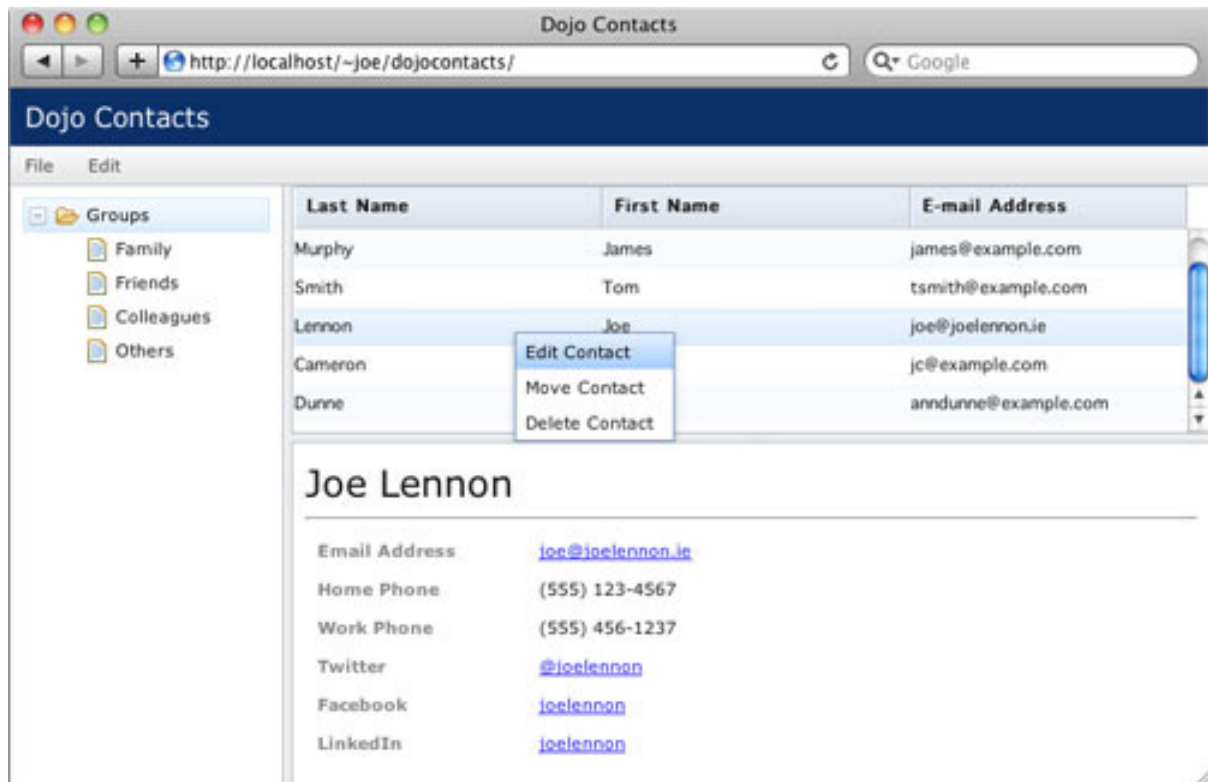
DojoX

Although much of the functionality required for most web applications is included in the other parts of the Dojo Toolkit, sometimes you need additional features that aren't available out of the box. With other libraries, you'd often need to look for third-party plug-ins to gain access to such features. Dojo's community-driven DojoX extension library means that you can access many stable and experimental additional components without needing to look elsewhere. In this tutorial, I use the following DojoX component: DojoX.

The sample applicationa basic contact management application

The focus of this tutorial is on the creation of a full-featured sample application that includes various CRUD (Create, Update, Delete) operations to resemble a potential real-world application. The end result is a web-based contact management system that has desktop-like performance. Figure 1 shows how the final product will look in your web browser.

Figure 1. Final application main application



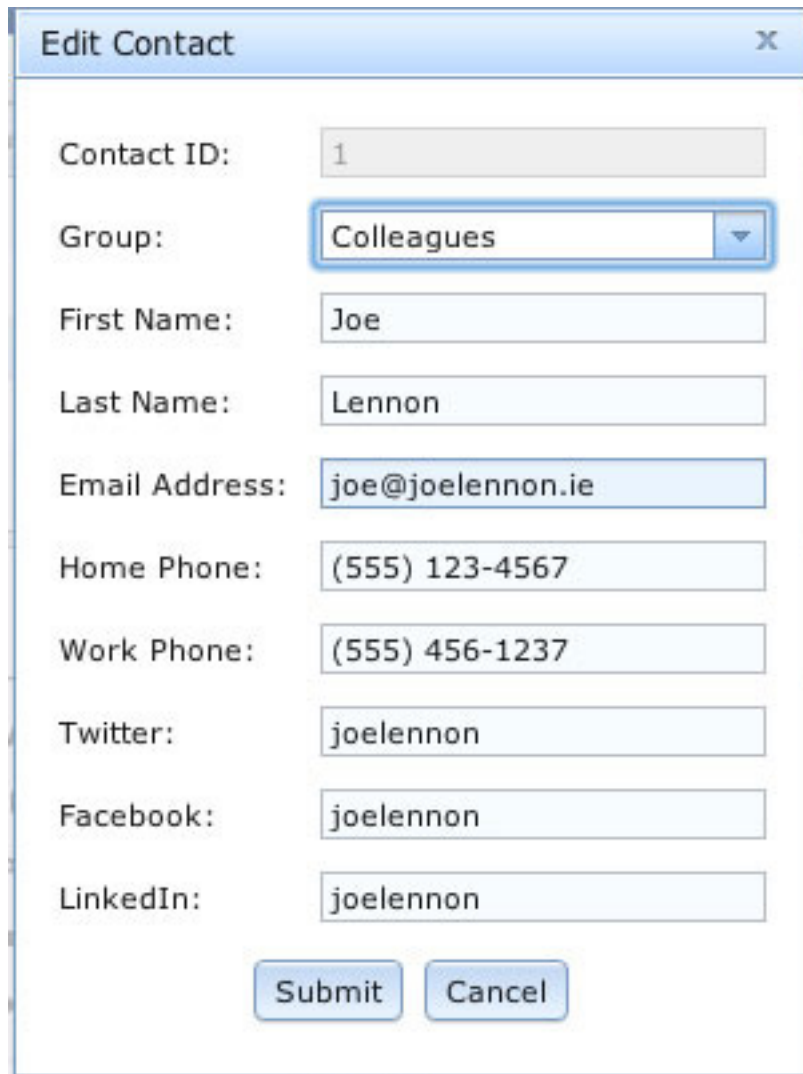
At the top of the application you have an application menu, with File and Edit options. These include sub-menus for the various operations you can perform, including adding new groups and contacts, and editing and deleting existing items.

On the left side of the application is a tree that displays the different groups available. Creating a new group results in the group being added to this tree. A context menu is available on the tree, so when you right-click an item in the tree you get options to rename or delete that group. Selecting (left-clicking) a group loads the contacts from that group on the right side. If you select the root node (Groups, which cannot be deleted), every contact, regardless of their group, is displayed on the right.

The right side of the application is split into two views, with the top section displaying a grid view of the contacts in the currently selected group, and the bottom half showing the details of the currently selected contact. Again, the grid features a context menu, which lets you edit, move, or delete a contact by right-clicking the contact and choosing the appropriate option.

The application makes extensive use of pop-up windows to provide additional functions, particularly in terms of creating, editing, and deleting items. An example of this is shown in Figure 2.

Figure 2. Dialog pop-up example: Editing contacts



Edit Contact [X]

Contact ID:

Group:

First Name:

Last Name:

Email Address:

Home Phone:

Work Phone:

Twitter:

Facebook:

LinkedIn:

The pop-up windows in the application fade out the main application in the background and display the pop-up window in focus over the application window. These windows typically take the shape of a form with components, including text boxes (with built-in "required" validation) and auto-complete drop-down lists. When deleting items, a confirmation window is shown that asks the user to confirm the deletion of the item. Pressing **OK** causes the item to be deleted; otherwise, the process is canceled.

I'm sure you'll agree that this application looks quite like a typical desktop application. Other features are also included, such as the ability to resize the split between the left and right side of the application, and the grid and preview panes on the right side. When you resize your browser window, the application resizes to take up the entire window. You should be able to take this sample application and build on its structure to create impressive looking results.

Section 3. Server-side setup: Database and PHP scripts

In this section, you will set up a new MySQL database to store the data for the contact manager application. You will also set up a PHP script that will manage the connection to this database, as well as a series of PHP APIs that will take the data from MySQL and translate it into a JavaScript Object Notation (JSON) format that can be easily read by Dojo's XHR (Ajax) function callbacks.

Getting started: MySQL database and server-side configuration

Before getting started with Dojo, you need to set up the server-side part of the application. The first thing you need to do is create the MySQL user, database, tables, and data that will form the data layer for the application. If you have root access to MySQL (or at least the ability to create users and databases), you can simply run the `install.sql` script included with the source code of this tutorial. Assuming the MySQL bin directory is on your system path and you are in the source code folder, you can run this script as follows:

```
$ mysql -u root -pMyPassword < install.sql
```

.

Obviously, you should replace "MyPassword" with your actual MySQL root password. This creates a new database user, database, tables, and some sample data. The contents of `install.sql` are shown in Listing 1.

Listing 1. `install.sql` script for setting up MySQL database

```
create user 'dojo'@'localhost' identified by 'somepass';
create database dojocontacts;
grant all privileges on dojocontacts.* to 'dojo'@'localhost' with grant option;
use dojocontacts;

create table groups(
  id          int(11) auto_increment primary key,
  name        varchar(100) not null
) engine=INNODB;

create table contacts(
  id          int(11) auto_increment primary key,
  group_id    int(11) not null,
  first_name  varchar(100) not null,
  last_name   varchar(100) not null,
  email_address varchar(255) not null,
  home_phone  varchar(100),
  work_phone  varchar(100),
  mobile_phone varchar(100),
  twitter     varchar(255),
```

```
        facebook      varchar(255),
        linkedin      varchar(255),
        foreign key(group_id) references groups(id) on delete cascade
    ) engine=INNODB;

insert into groups(name) values('Family');
insert into groups(name) values('Friends');
insert into groups(name) values('Colleagues');
insert into groups(name) values('Others');

insert into contacts(group_id, first_name, last_name, email_address, home_phone,
work_phone, twitter, facebook, linkedin)
values(3, 'Joe', 'Lennon', 'joe@joelennon.ie', '(555) 123-4567', '(555) 456-1237',
'joelennon', 'joelennon', 'joelennon');

insert into contacts(group_id, first_name, last_name, email_address, home_phone,
work_phone, twitter, facebook, linkedin)
values(1, 'Mary', 'Murphy', 'mary@example.com', '(555) 234-5678', '(555) 567-2348',
'mmurphy', 'marym', 'mary.murphy');

insert into contacts(group_id, first_name, last_name, email_address, home_phone,
work_phone, twitter, facebook, linkedin)
values(2, 'Tom', 'Smith', 'tsmith@example.com', '(555) 345-6789', '(555) 678-3459',
'tom.smith', 'tsmith', 'smithtom');

insert into contacts(group_id, first_name, last_name, email_address, home_phone,
work_phone, twitter, facebook, linkedin)
values(4, 'John', 'Cameron', 'jc@example.com', '(555) 456-7890', '(555) 789-4560',
'jcameron', 'john.cameron', 'johnc');

insert into contacts(group_id, first_name, last_name, email_address, home_phone,
work_phone, twitter, facebook, linkedin)
values(4, 'Ann', 'Dunne', 'anndunne@example.com', '(555) 567-8901', '(555) 890-5671',
'ann.dunne', 'adunne', 'dunneann');

insert into contacts(group_id, first_name, last_name, email_address, home_phone,
work_phone, twitter, facebook, linkedin)
values(1, 'James', 'Murphy', 'james@example.com', '(555) 678-9012', '(555) 901-6782',
'jmurphy', 'jamesmurphy', 'james.murphy');
```

Connecting to the database with PHP

With your MySQL database created, you can now create the PHP scripts that will insert, update, and delete data in these database tables.

In this sample application, the PHP scripts will merely be called using Ajax or XHR requests, and will always return a response to the request object in JSON format, with the exception of contact.php, which actually returns an HTML snippet that is inserted into the DOM. All of the PHP scripts are located in a subdirectory named data. Following are the scripts that are required for this project:

- contact.php (Retrieves the contact detail for a given contact)
- contacts.php (Retrieves a JSON representation of the contacts in the database)
- database.php (Takes care of connecting to MySQL)

- delete_contact.php (Deletes a contact from the database)
- delete_group.php (Deletes a group from the database)
- edit_contact.php (Adds/Edits a contact in the database)
- edit_group.php (Renames a group in the database)
- groups.php (Retrieves a JSON representation of the groups in the database)
- move_contact.php (Moves a contact to a different group)
- new_group.php (Adds a new group to the database)

Due to space restrictions, I can't list the contents of all of these files in this tutorial, so for full source code listings, see the accompanying [source code](#) for this tutorial. To give you an idea of how these PHP scripts work, however, I will discuss a few of them.

The database.php file is used by all of the other PHP scripts in this tutorial to open a connection to the MySQL database. This file is very straightforward; its contents are shown in Listing 2.

Listing 2. data/database.php file contents

```
<?php
$db_host = "localhost";
$db_user = "dojo";
$db_pass = "somepass";
$db_name = "dojocontacts";

//Connect to MySQL
$conn = mysql_connect($db_host, $db_user, $db_pass);
//Select database
mysql_select_db($db_name, $conn);
?>
```

Creating JSON-formatted API functions in PHP

There are two main types of operations performed in this application: retrieving data and manipulating data. Let's take a look at an example of each of these. First up, groups.php takes care of retrieving a list of groups from the database and represents it using the Dijit Identity API structure so it can be read by the Dijit Tree widget. Listing 3 shows the contents of this file.

Listing 3. data/groups.php file contents

```
<?php
include_once("database.php");
```

```
//SQL statement to get all groups
$sql = "SELECT id, name, 'node' as type FROM groups ORDER BY id";
$result = mysql_query($sql, $conn) or die("Could not load groups");

//Always show "Groups" as root element
$data = array(
    'identifier' => 'id',
    'label' => 'name',
    'items' => array(
        array(
            'id' => 0,
            'name' => 'Groups',
            'type' => 'root'
        )
    )
);

//Retrieve groups and add to array
if(mysql_num_rows($result) > 0) {
    while($row = mysql_fetch_assoc($result)) {
        $data['items'][0]['groups'][] = array('_reference' => $row['id']);
        $data['items'][] = $row;
    }
}

//Output $data array as JSON
header('Content-Type: application/json; charset=utf8');
echo json_encode($data);
?>
```

The Dijit Identity API requires that the JSON data structure take a particular format, with the properties `identifier`, `label`, and `items` at the top level. In Listing 3, I have added a root `Groups` item by default. This does not represent a particular group, but will always be displayed at the top of the tree (even if there are no groups). I have then looped through the result of the SQL `SELECT` statement and added each of these items as a child element.

First, each item is added to the `items` array, and the root item has a `groups` array added to it, with a collection of objects with a `_reference` property that refers to the child item's ID. Finally, the script takes the `$data` array that it builds up and outputs it as JSON. Later in this tutorial, you will create a data store to connect to this PHP script, which will take the JSON output and use it accordingly in a Dijit Tree.

The second example I discuss here is `move_contact.php`, which updates a contact record, setting the `group_id` value as required. Listing 4 shows the code for this file.

Listing 4. `data/move_contact.php` file contents

```
<?php
include_once("database.php");
//Get form values
$contact_id = $_POST['move_contact_id'];
$group_id = $_POST['move_contact_new'];
//Perform update
$sql = "UPDATE contacts SET group_id = ".mysql_real_escape_string($group_id, $conn)." WHERE
```

```

id = ".mysql_real_escape_string($contact_id, $conn);
$result = mysql_query($sql) or die("Could not move contact in database");
//Check if performed via Ajax
if(mysql_affected_rows() > 0 && $_POST['move_contact_ajax'] == "0") {
    header("Location: ../index.html");
} else {
    //If Ajax, return JSON response
    header('Content-Type: application/json; charset=utf8');
    $data = array();
    //If rows affected, change was successful
    if(mysql_affected_rows() > 0) {
        $data['success'] = true;
        $data['id'] = $contact_id;
    } else {
        $data['success'] = false;
        $data['error'] = 'Error: could not move contact in database';
    }
    //Output array in JSON format
    echo json_encode($data);
}
?>

```

In Listing 4, two `POST` parameters are required, the contact ID and the new group ID that the contact should be moved to. These are then plugged into an SQL `UPDATE` statement. When the request is made through Ajax, a parameter is used to indicate that it was sent using Ajax and not using a regular HTML `POST` command. If it was the latter, the page is refreshed (to allow for a standard HTML fallback that you can implement if you want). If it was indeed performed using Ajax, a response structure is constructed and returned in a JSON format.

Most of the PHP server-side APIs work in a similar manner to the above code listings. As I said previously, for full code listings of all the scripts, see this tutorial's accompanying [source code](#).

With the server-side part of the application configured, you are now ready to use Dojo to create a snazzy front end to the application. The next section explains just how simple it is to declaratively add Dijit components to your HTML document.

Section 4. Building the user interface with Dijit

Dojo's rich component library, Dijit, lets you build full user interfaces with minimal coding and minimal fuss. Dijit components can be used in two ways: either programmatically using JavaScript or declaratively using HTML-style elements with a `dojoType` attribute. In this tutorial, I create all of the Dijit widgets used declaratively. Any JavaScript logic behind these widgets will be done in a separate JavaScript source file, however.

The main application code is contained in three files. The layout and widget

declarations are done in the main index.html file. A subdirectory named css contains a single file, style.css, which contains all the stylesheets required for this particular application. Finally, all JavaScript logic is stored in the script.js file in the js subdirectory.

For the code for the style.css file, see the source code that accompanies this tutorial. This code is all basic CSS syntax and should be relatively self-explanatory.

In this section, I explain how to build the user interface for the application using HTML and Dojo widgets.

The basic structure

Start the application by creating the basic HTML structure. This includes any relative stylesheets, setting the Dijit theme to be used, and loading the JavaScript source files required. In this application, I am loading Dojo from Google's CDN for convenience. Of course, this means that you can only use the application if you have an Internet connection available. If you want to use the application offline, you can simply download the Dojo library and load it locally.

Listing 5 shows the starting point for the application's main index.html file.

Listing 5. Basic index.html structure

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Dojo Contacts</title>
    <link rel="stylesheet"
href="https://ajax.googleapis.com/ajax/libs/dojo/1.5/dijit/themes/claro/claro.css">
    <link rel="stylesheet"
href="https://ajax.googleapis.com/ajax/libs/dojo/1.5/dojox/grid/resources/Grid.css">
    <link rel="stylesheet"
href="https://ajax.googleapis.com/ajax/libs/dojo/1.5/dojox/grid/resources
/claroGrid.css">
    <link rel="stylesheet" href="css/style.css">
  </head>
  <body class="claro">
    <!-- Content goes here -->
    <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs
/dojo/1.5/dojo/dojo.xd.js" djConfig="parseOnLoad: true"></script>
    <script type="text/javascript" src="js/script.js"></script>
  </body>
</html>
```

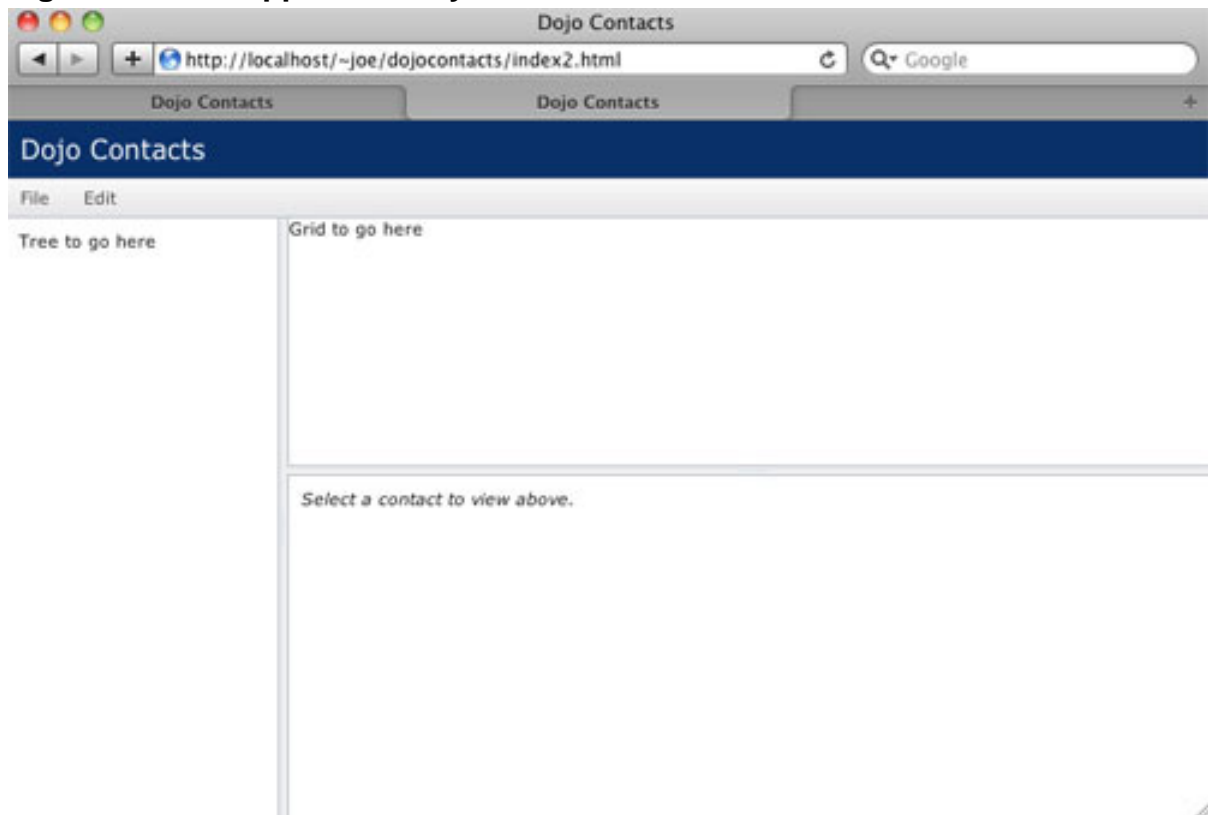
As you can see, the application loads four stylesheets. Three of these are from Dojo itself: the first one is the main theme CSS file for the Claro Dijit theme, and the other two are used for the Data Grid DojoX extension. An important point to note is that I have added a class `claro` to the main `<body>` element of this document. This tells the Dojo parser that it should use the Claro theme. Without this, the application will

not look right, so don't leave it out. I have loaded the `<script>` blocks for the application at the bottom of the file, just before the closing `</body>` element. This is the generally recommended way of including JavaScript source files, as it will not prevent the rest of the page from loading before the scripts have finished loading. I have loaded the main Dojo script from Google's CDN and the application's script file, which you will create later in this tutorial.

Defining the application layout

Next, create the main application layout. The goal for now is to achieve the appearance shown in Figure 3. Basically, by the end of this section, you want to be able to load the file in your web browser and see something like this screenshot.

Figure 3. Basic application layout



To use the Dijit components used in the application, you need to tell Dojo to load these widgets using the `dojo.require` JavaScript function. Create a file named `script.js` and save it in a subdirectory named `js` in your application folder. Add the contents of Listing 6 to this file.

Listing 6. Loading the Dijit layout components

```

dojo.require("dijit.dijit");
dojo.require("dijit.layout.BorderContainer");
dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.MenuBar");
dojo.require("dijit.PopupMenuBarItem");
dojo.require("dijit.Menu");
dojo.require("dijit.MenuItem");

```

This tells Dojo that it should load the `BorderContainer` and `ContentPane` layout widgets, as well as various `Menu` widgets that will be used in the application's menu bar. With these items loaded, you can now add them to your application's `index.html` file. Replace the comment `<!-- Content goes here -->` in the file with the contents of Listing 7.

Listing 7. Adding the layout components to the application

```

<div dojoType="dijit.layout.BorderContainer" design="header" gutters="false"
liveSplitters="true" id="borderContainer">
  <div dojoType="dijit.layout.ContentPane" region="top" id="topBar">
    <h1>Dojo Contacts</h1>
    <div dojoType="dijit.MenuBar" id="navMenu">
      <div dojoType="dijit.PopupMenuBarItem">
        <span>File</span>
        <div dojoType="dijit.Menu" id="fileMenu">
          <div dojoType="dijit.MenuItem"
jsId="mnuNewContact">New Contact</div>
          <div dojoType="dijit.MenuItem"
jsId="mnuNewGroup">New Group</div>
        </div>
      </div>
      <div dojoType="dijit.PopupMenuBarItem">
        <span>Edit</span>
        <div dojoType="dijit.Menu" id="editMenu">
          <div dojoType="dijit.MenuItem" jsId="mnuEditContact"
disabled="true">Edit Contact</div>
          <div dojoType="dijit.MenuItem" jsId="mnuMoveContact"
disabled="true">Move Contact</div>
          <div dojoType="dijit.MenuItem" jsId="mnuRenameGroup"
disabled="true">Rename Group</div>
          <div dojoType="dijit.MenuSeparator"></div>
          <div dojoType="dijit.MenuItem" jsId="mnuDeleteContact"
disabled="true">Delete Contact</div>
          <div dojoType="dijit.MenuItem" jsId="mnuDeleteGroup"
disabled="true">Delete Group</div>
        </div>
      </div>
    </div>
    <div dojoType="dijit.layout.BorderContainer" region="center" gutters="true"
liveSplitters="true" id="mainSection">
      <div dojoType="dijit.layout.ContentPane" splitter="true" region="left"
id="treeSection">
        Tree to go here
      </div>
      <div dojoType="dijit.layout.BorderContainer" design="header" gutters="true"
liveSplitters="true" id="mainContainer" region="center">
        <div dojoType="dijit.layout.ContentPane" region="top" splitter="true"
id="gridSection">
          Grid to go here
        </div>
        <div dojoType="dijit.layout.ContentPane" id="contactView"
jsId="contactView" region="center">

```

```

        <em>Select a contact to view above.</em>
      </div>
    </div>
  </div>
</div>

```

As shown in Listing 7, you add Dijit components to your page using standard HTML elements with a special `dojoType` attribute. These widget declarations also use some distinctive attributes that are specific to that widget type, for example the `region` attribute is used to define where a child element of a `BorderContainer` should display in the context of the container. There are quite a few components used in Listing 7, so let's take a look at the basic structure for the components:

- `dijit.layout.BorderContainer`
 - `dijit.layout.ContentPane` (top)
- `h1`
- `dijit.MenuBar`
- `dijit.PopupMenuBarItem`
 - `dijit.Menu`
- `dijit.MenuItem`
- `dijit.MenuItem`
- `dijit.PopupMenuBarItem`
 - `dijit.Menu`
- `dijit.MenuItem`
- `dijit.MenuItem`
- `dijit.MenuItem`
 - `dijit.layout.BorderContainer` (center)
- `dijit.layout.ContentPane` (left)
- `dijit.layout.BorderContainer` (center)
- `dijit.layout.ContentPane` (top)
- `dijit.layout.ContentPane` (center)

At the top level there is a `BorderContainer`, which lets you place child components in one of five regions: top, center, left, right, and bottom. Below this are two components, a `ContentPane` at the top and a `BorderContainer` at the center. Because there are only two components used, the center component will

take up all space not used by the top component. In the top section, there is a standard HTML `h1` heading and a Dijit Menu structure. In the center section, there is a nested `BorderContainer` component, with a `ContentPane` on the left, and another `BorderContainer` in the center (which takes up any space not used by the left component). The left `ContentPane` is where you will put your Tree control. The `BorderContainer` in the center has two child components — both `ContentPanes` — in the top and center regions.

At this stage, you should be able to save your file and load it in your web browser to get a result like the one you earlier saw in [Figure 3](#).

Adding the Tree component

Next, add a `Tree` component to the left side of the application. Dijit Trees are complex components, and require a couple of support components to ensure that they work appropriately. First, they require a data store that adheres to Dojo's Identity API, connected to some JSON data source. In the sample application, this data is generated by the PHP script `data/groups.php`, so you can connect the tree's data store to this API. Next, the `Tree` needs a model component to define the structure of the data store and what data it should read. You then add your `Tree` component and connect it to this model.

To get started, you first need to tell Dojo the components it should load so that they are available to the application. Add the two lines of code shown in Listing 8 to the `script.js` file.

Listing 8. Loading tree-related components

```
dojo.require("dojo.data.ItemFileWriteStore");
dojo.require("dijit.Tree");
```

You might be wondering why a writable store is being used. Dojo's `Tree` component is quite complex to work with and actually cannot be directly refreshed asynchronously (unless you put it in a `ContentPane` and refresh the entire `ContentPane`), so to add/update/delete items from the tree dynamically it needs to connect to a writable store.

Next, find the "Tree to go here" section of your `index.html` file and replace it with the code in Listing 9. This defines the data store, the tree model, and the tree itself and puts it in the left side of the application. It also adds a context menu to the tree, although the items will be disabled right now. You will implement the context menu later.

Listing 9. Adding the tree to the application


```

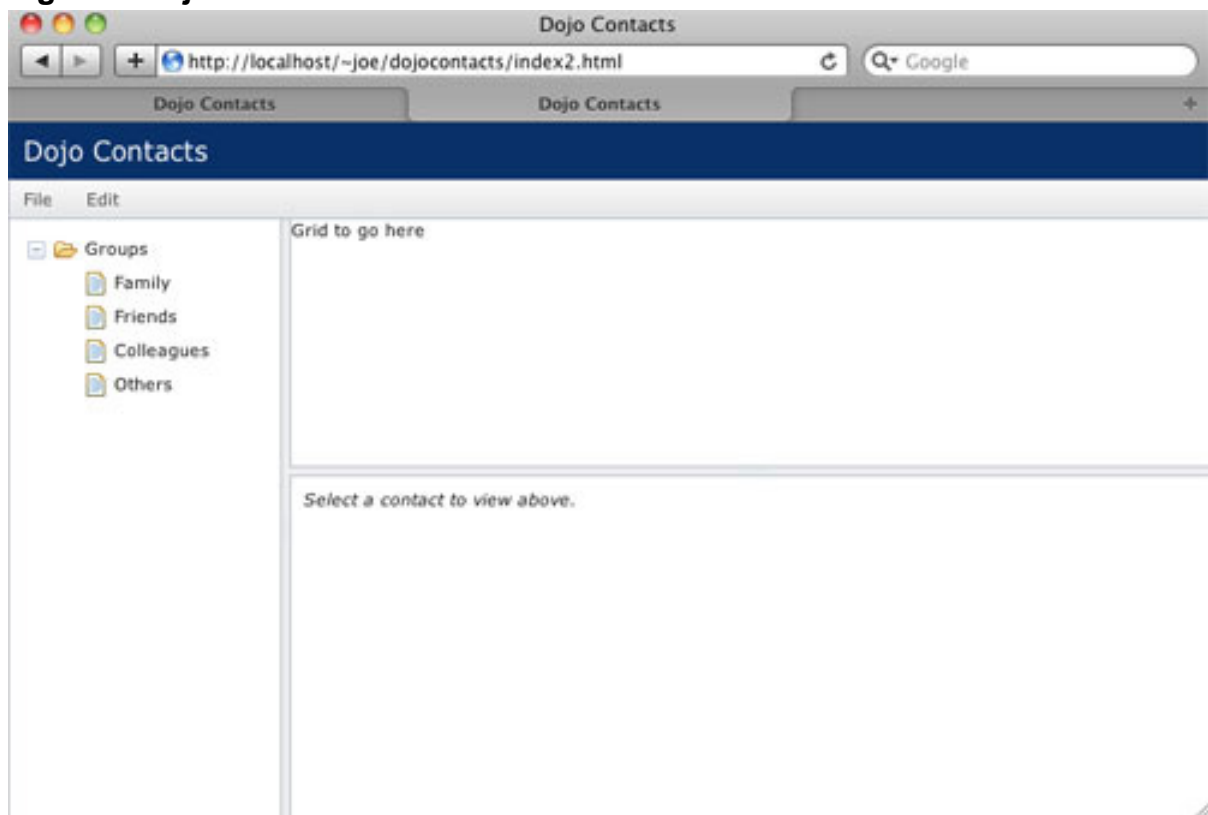
<div dojoType="dojo.data.ItemFileWriteStore" jsId="groupsStore"
url="data/groups.php"></div>
<div dojoType="dijit.tree.TreeStoreModel" jsId="groupsModel"
childrenAttrs="groups" store="groupsStore" query="{id:0}"></div>
<div dojoType="dijit.Tree" id="groupsTree" jsId="groupsTree" model="groupsModel">
  <div dojoType="dijit.Menu" targetNodeIds="groupsTree">
    <div dojoType="dijit.MenuItem" jsId="ctxMnuRenameGroup"
disabled="true">Rename Group</div>
    <div dojoType="dijit.MenuItem" jsId="ctxMnuDeleteGroup"
disabled="true">Delete Group</div>
  </div>
</div>

```

In Listing 9, you'll notice the attribute `jsId` is used in each component. This attribute tells the Dojo parser to create JavaScript variables for each of these components and names it as the value given here. This makes it much easier to work with these components later, as it removes the need to use `dijit.byId` to query the DOM in an effort to find these widgets. It should also give a performance reward, as DOM querying can be expensive in terms of application speed and responsiveness, particularly on slower web browsers.

If you reload your browser, the application should now look like the window in Figure 4 (assuming you have the PHP scripts set up correctly and the data for the tree actually loads).

Figure 4. Dijit Tree in action



Adding the Grid component

Adding the grid to the application should work similarly to the tree component. Unlike the tree, however, the grid can be easily reloaded asynchronously by refreshing the data store, so no model is required, and a read-only store can safely be used. First, add the following line to the script.js file so the grid component is available for use in the application:

```
dojo.require("dojox.grid.DataGrid");
```

Note that the Data Grid widget is included in DojoX and not the Dijit library. DojoX components often require additional stylesheets to be loaded (in this case, these styles have already been included in the basic template I created earlier).

Now you can add the grid to the main application file. Data grids in Dojo can be applied to a standard `<table>` element, allowing you to easily define the headings you want to use. The grid can define a series of attributes to set how the grid should work; for example, whether client-side column sorting should be enabled. Find the "Grid to go here" reference in the index.html file and replace it with the code in Listing 10.

Listing 10. Adding the grid to the application

```
<span dojoType="dojo.data.ItemFileReadStore" jsId="contactsStore"
url="data/contacts.php"></span>
<table dojoType="dojox.grid.DataGrid" id="contactsGrid" jsId="contactsGrid"
columnReordering="true" sortFields="['last_name','first_name']" store="contactsStore"
query="{first_name: '*'}"
clientSort="true" selectionMode="single" rowHeight="25" noDataMessage="<span
class='dojoxGridNoData'>No contacts found in this group</span>">
  <thead>
    <tr>
      <th field="last_name" width="200px">Last Name</th>
      <th field="first_name" width="200px">First Name</th>
      <th field="email_address" width="100%">E-mail Address</th>
    </tr>
  </thead>

  <script type="dojo/method" event="onRowContextMenu" args="e">
  </script>
</table>
<div dojoType="dijit.Menu" id="gridMenu" targetNodeIds="contactsGrid">
  <div dojoType="dijit.MenuItem" jsId="ctxMnuEditContact"
disabled="true">Edit Contact</div>
  <div dojoType="dijit.MenuItem" jsId="ctxMnuMoveContact"
disabled="true">Move Contact</div>
  <div dojoType="dijit.MenuItem" jsId="ctxMnuDeleteContact"
disabled="true">Delete Contact</div>
</div>
```

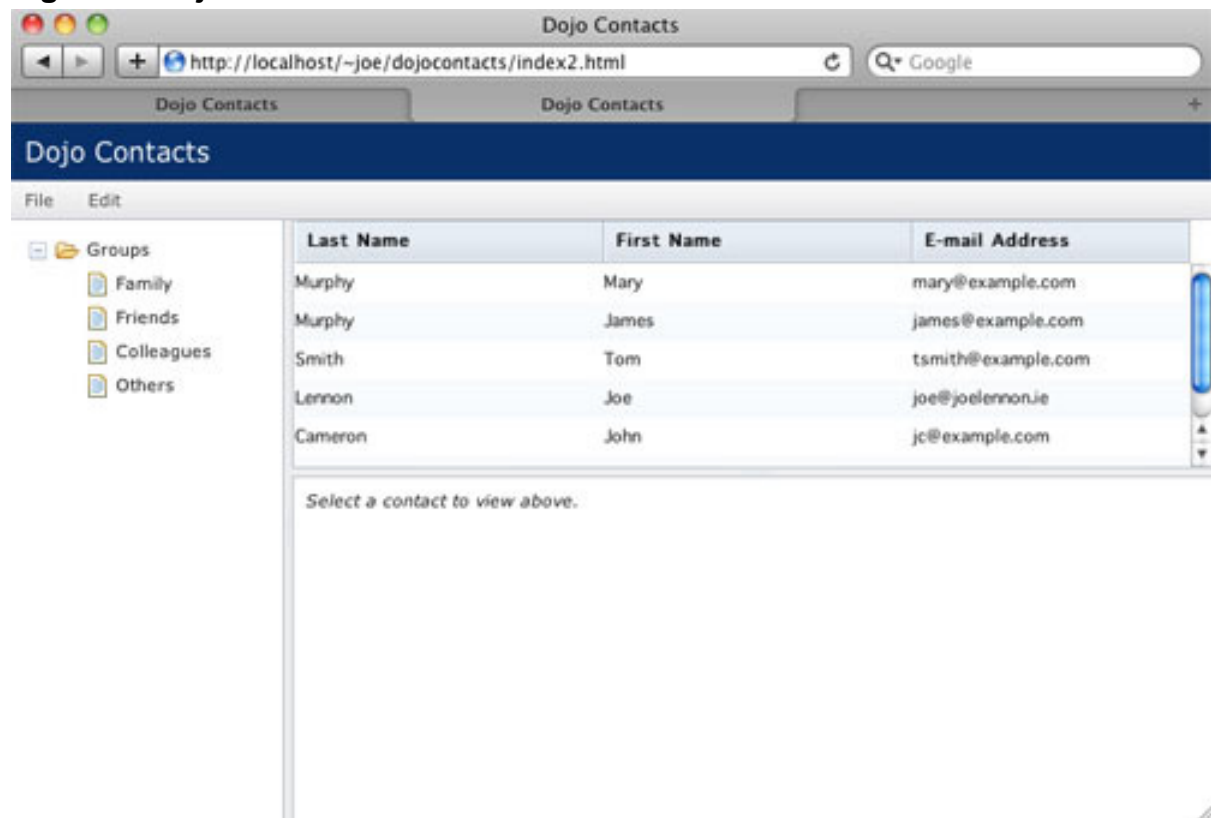
Again, like the tree previously, the grid also has a context menu available. You will

see how to enable these menus later in this tutorial. If you're wondering why there is an inline Dojo method `<script>` block, this is due to a bug in the DataGrid component where a context menu will not display without this empty block being included. It is worth pointing out that you could actually put JavaScript code directly in your layout code using these blocks if you like, but, personally, I prefer to separate my application logic from the UI elements where possible.

The grid loads the data using a read-only data store, which connects to the API `data/contacts.php`. It automatically sorts the data on the client side by last name and then first name. The grid only allows one row to be selected at a time, and defines an error message that is displayed if no data is found.

At this stage, your application should look like Figure 5 (again assuming you have added the PHP scripts from the source code bundle and configured the server side correctly):

Figure 5. DojoX Data Grid in action



Hidden application components: dijit.Dialog windows

At this point, the main application UI is complete. In this section, you will add some hidden dialog windows that will display in a modal way over the main application window when you select a menu option. These pop-up windows typically contain

web forms that let you define the various properties of a group or contact, but you will also create a simple **OK** message window that looks somewhat nicer than the standard JavaScript alert dialog box.

There are five of these dialogs in total. These dialog windows contain various Dijit form fields, including ValidationTextBoxes, FilteringSelects, and Buttons. To include all of these additional widgets, you must load them using `dojo.require`, so add the code from Listing 11 to your script.js file.

Listing 11. Loading dialog and form components

```
dojo.require("dijit.Dialog");
dojo.require("dijit.form.Form");
dojo.require("dijit.form.ValidationTextBox");
dojo.require("dijit.form.TextBox");
dojo.require("dijit.form.FilteringSelect");
dojo.require("dijit.form.Button");
```

You can now start adding the dialog boxes. These components should be added outside of the main application `BorderContainer` to ensure that they appear over all other elements on the page. Start adding them just above the first `<script>` element near the bottom of the `index.html` file.

First, add the "New Group" and "Rename Group" dialog boxes. The code for these windows is shown in Listing 12.

Listing 12. New Group and Rename Group dialog box code

```
<div id="newGroupDialog" jsId="newGroupDialog" dojoType="dijit.Dialog" title="Create
New Group" draggable="false">
  <div dojoType="dijit.form.Form" id="newGroupForm"
jsId="newGroupForm" action="data/new_group.php" method="post">
    <input type="hidden" name="new_group_ajax"
id="new_group_ajax" value="0">
    <label for="new_group_name">Group Name:</label>
    <input type="text" name="new_group_name"
id="new_group_name" required="true" dojoType="dijit.form.ValidationTextBox" />
    <button dojoType="dijit.form.Button" type="submit">Submit</button>
    <button dojoType="dijit.form.Button" jsId="newGroupCancel"
type="button">Cancel</button>
  </div>
</div>

<div id="editGroupDialog" jsId="editGroupDialog" dojoType="dijit.Dialog"
title="Rename Group" draggable="false">
  <div dojoType="dijit.form.Form" id="editGroupForm" jsId="editGroupForm"
action="data/edit_group.php" method="post">
    <input type="hidden" name="edit_group_ajax" id="edit_group_ajax"
value="0">
    <input type="hidden" name="edit_group_id" id="edit_group_id">
    <table cellpadding="4" cellspacing="4">
      <tr>
        <td><label for="edit_group_old">Old Group
Name:</label></td>
        <td><input type="text" name="edit_group_old" id="edit_group_old"
disabled="true" dojoType="dijit.form.TextBox" /></td>
```

```

        </tr>
        <tr>
            <td><label for="edit_group_name">New Group
Name:</label></td>
            <td><input type="text" name="edit_group_name"
id="edit_group_name" required="true" dojoType="dijit.form.ValidationTextBox"
style="margin-bottom: 6px" /></td>
        </tr>
        <tr>
            <td colspan="2" align="center">
                <button dojoType="dijit.form.Button"
type="submit">Submit</button>
                <button dojoType="dijit.form.Button" jsId="editGroupCancel "
type="button">Cancel</button>
            </td>
        </tr>
    </table>
</div>
</div>

```

The New Group dialog contains a form, which will submit to the PHP script `data/new_group.php`. This form contains a `ValidationTextBox` form control, which is a Dijit-styled text box with automatic validation. In this case, the control has been flagged as mandatory by setting the attribute `required` to `true`. The dialog also contains Submit and Cancel buttons. You will implement the code that handles this form (and all other forms that are created in this section) later in the tutorial.

The Rename Group dialog allows users to change the name of a group. It displays a read-only `TextBox` control with the current group name and provides a `ValidationTextBox` for the new group name to be supplied. Again, it also features a pair of buttons.

Next, add the Move Contact dialog window, as shown in Listing 13.

Listing 13. Move Contact dialog mark-up

```

<div id="moveContactDialog" jsId="moveContactDialog" dojoType="dijit.Dialog"
title="Move Contact" draggable="false">
    <div dojoType="dijit.form.Form" id="moveContactForm" jsId="moveContactForm"
action="data/move_contact.php" method="post">
        <input type="hidden" name="move_contact_ajax"
id="move_contact_ajax" value="0">
        <input type="hidden" name="move_contact_id" id="move_contact_id">
        <table cellpadding="4" cellspacing="4">
            <tr>
                <td><label for="move_contact_name">Contact Name:
</label></td>
                <td><input type="text" name="move_contact_name"
id="move_contact_name" disabled="true" dojoType="dijit.form.TextBox" /></td>
            </tr>
            <tr>
                <td><label for="move_contact_old">Current
Group:</label></td>
                <td><input type="text" name="move_contact_old"
id="move_contact_old" disabled="true" dojoType="dijit.form.TextBox" /></td>
            </tr>
            <tr>
                <td><label for="move_contact_new">New Group:</label>

```

```

                <td><input dojoType="dijit.form.FilteringSelect"
name="move_contact_new" store="groupsStore" searchAttr="name" query="{type:'node'}"
id="move_contact_new" required="true" style="margin-bottom: 6px" /></td>
            </tr>
            <tr>
                <td colspan="2" align="center">
                    <button dojoType="dijit.form.Button"
type="submit">Submit</button>
                    <button dojoType="dijit.form.Button" jsId="moveContactCancel"
type="button">Cancel</button>
                </td>
            </tr>
        </table>
    </div>
</div>

```

This dialog displays the selected contact's name and the current group they belong to. It then displays a `FilteringSelect` drop-down list, which allows the user to select a new group to move the contact to. This drop-down is actually backed by the same data store as the `Tree` control that displays the list of groups on the left side of the interface. However, the `query` attribute of the `FilteringSelect` component lets you tell Dojo not to include the root `Groups` element, so the user cannot select it as an option to move the contact to.

The next dialog is the Edit Contact dialog, which is actually used for both adding new contacts and modifying existing ones. The code for this dialog is in Listing 14.

Listing 14. Edit/Add Contact dialog mark-up

```

<div id="editContactDialog" jsId="editContactDialog" dojoType="dijit.Dialog"
title="Edit Contact" draggable="false">
    <div dojoType="dijit.form.Form" id="editContactForm" jsId="editContactForm"
action="data/edit_contact.php" method="post">
        <input type="hidden" name="edit_contact_ajax" id="edit_contact_ajax"
value="0">
        <input type="hidden" name="edit_contact_real_id"
id="edit_contact_real_id">
        <table cellpadding="4" cellspacing="4">
            <tr><td><label for="edit_contact_id">Contact ID:
</label></td>
            <td><input type="text" name="edit_contact_id" id="edit_contact_id"
disabled="true" dojoType="dijit.form.TextBox" /></td></tr>
            <tr><td><label for="move_contact_new">Group:
</label></td>
            <td><input dojoType="dijit.form.FilteringSelect"
name="edit_contact_group" store="groupsStore" searchAttr="name" query="{type:'node'}"
id="edit_contact_group" required="true" /></td></tr>
            <tr><td><label for="edit_contact_first_name">First Name:
</label></td>
            <td><input type="text" name="edit_contact_first_name"
id="edit_contact_first_name" required="true" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
            <tr><td><label for="edit_contact_last_name">Last Name:
</label></td>
            <td><input type="text" name="edit_contact_last_name"
id="edit_contact_last_name" required="true" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
            <tr><td><label for="edit_contact_email_address">
Email Address:</label></td>
            <td><input type="text" name="edit_contact_email_address"

```

```

id="edit_contact_email_address" required="true" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
    <tr><td><label for="edit_contact_home_phone">Home
Phone:</label></td>
    <td><input type="text" name="edit_contact_home_phone"
id="edit_contact_home_phone" required="false" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
    <tr><td><label for="edit_contact_work_phone">Work
Phone:</label></td>
    <td><input type="text" name="edit_contact_work_phone"
id="edit_contact_work_phone" required="false" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
    <tr><td><label for="edit_contact_twitter">Twitter:
</label></td>
    <td><input type="text" name="edit_contact_twitter"
id="edit_contact_twitter" required="false" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
    <tr><td><label for="edit_contact_facebook">Facebook:
</label></td>
    <td><input type="text" name="edit_contact_facebook"
id="edit_contact_facebook" required="false" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
    <tr><td><label for="edit_contact_linkedin">LinkedIn:
</label></td>
    <td><input type="text" name="edit_contact_linkedin"
id="edit_contact_linkedin" required="false" dojoType="dijit.form.ValidationTextBox"
/></td></tr>
    <tr><td colspan="2" align="center">
        <button dojoType="dijit.form.Button"
type="submit">Submit</button>
        <button dojoType="dijit.form.Button" jsId="editContactCancel"
type="button">Cancel</button>
    </td></tr>
</table>
</div>
</div>

```

There's quite a bit of mark-up for this form, but that's only because of the number of fields it contains; it's not any more complex than the Move Contact dialog, really. It has a display-only field for the Contact ID value, which will be set to [NEW] in JavaScript in the case of creating new contacts (the ID will be auto-generated in MySQL). It also contains form fields for the Group (the same drop-down as the Move Contact dialog), first name, last name, email address, home phone, work phone, Twitter account, Facebook account, and LinkedIn account.

The final dialog that needs to be created is a standard "OK" dialog box that will be used as a success or error message display window whenever a save operation is performed using XHR/Ajax. The code for this dialog is shown in Listing 15.

Listing 15. OK message dialog box mark-up

```

<div id="okDialog" jsId="okDialog" dojoType="dijit.Dialog" title="Title"
draggable="false">
    <p id="okDialogMessage" style="margin-top: 5px">Message</p>
    <div align="center">
        <button dojoType="dijit.form.Button" jsId="okDialogOK"
type="button">OK</button>
    </div>
</div>

```

There's nothing at all complex about this dialog—it simply contains a message and a button to close it. With this dialog now created, that concludes the interface code for the application. You can reload the interface in your browser if you wish, but you won't see any difference to the screenshot in [Figure 5](#), as these dialogs need to be connected using JavaScript. The next section explains how to do this.

Section 5. Application interface logic

With the user interface mark-up created, you can now start to connect the various components to one another using JavaScript.

Adding logic on load

Any JavaScript code that you add should be executed when the DOM has finished loading, so before doing anything, add the code in Listing 16 to the end of the script.js file in the js subdirectory of your application.

Listing 16. Add logic on load

```
dojo.addOnLoad(function() {  
    //Application code goes here  
});
```

All the source code you add from here on should be added inside this function block. First, let's connect the tree and the data grid so that when you click on a node in the tree, the grid will display the contacts in the selected group.

Connecting the tree to the grid

Add the following functions in Listing 17 inside the `dojo.addOnLoad` function block.

Listing 17. Updating the data grid from the tree

```
function refreshGrid() {  
    contactsGrid.selection.clear();  
    mnuEditContact.set("disabled", true);  
    mnuMoveContact.set("disabled", true);  
    mnuDeleteContact.set("disabled", true);  
    ctxMnuEditContact.set("disabled", true);  
    ctxMnuMoveContact.set("disabled", true);  
    ctxMnuDeleteContact.set("disabled", true);  
    dijit.byId("contactView").set("content", '<em>Select a contact to
```



```

view above.</em>');
}

function updateDataGrid(item) {
    var newURL = "data/contacts.php?group_id="+item.id;
    var newStore = new dojo.data.ItemFileReadStore({url: newURL});
    contactsGrid.setStore(newStore);
    refreshGrid();
}

function selectNode(e) {
    var item = dijit.getEnclosingWidget(e.target).item;
    if(item !== undefined) {
        groupsTree.set("selectedItem",item);
        if(item.id != 0) {
            mnuRenameGroup.set("disabled",false);
            mnuDeleteGroup.set("disabled",false);
            ctxMnuRenameGroup.set("disabled",false);
            ctxMnuDeleteGroup.set("disabled",false);
        } else {
            mnuRenameGroup.set("disabled",true);
            mnuDeleteGroup.set("disabled",true);
            ctxMnuRenameGroup.set("disabled",true);
            ctxMnuDeleteGroup.set("disabled",true);
        }
    }
}
}

```

The `refreshGrid` function simply clears any selections in the grid and the main contact view pane and disables any related menu options. The `updateGrid` function refreshes the grid's data store, telling it to look for contacts in the selected group. Finally, the `selectNode` function sets the selected node on the tree, and enables/disables menu options as appropriate (they should be enabled if a "real" group is selected and disabled if the root `Groups` node is selected).

Now it's time to connect these functions to events so that the tree and grid can talk to one another (see Listing 18).

Listing 18. Connecting the tree to the grid

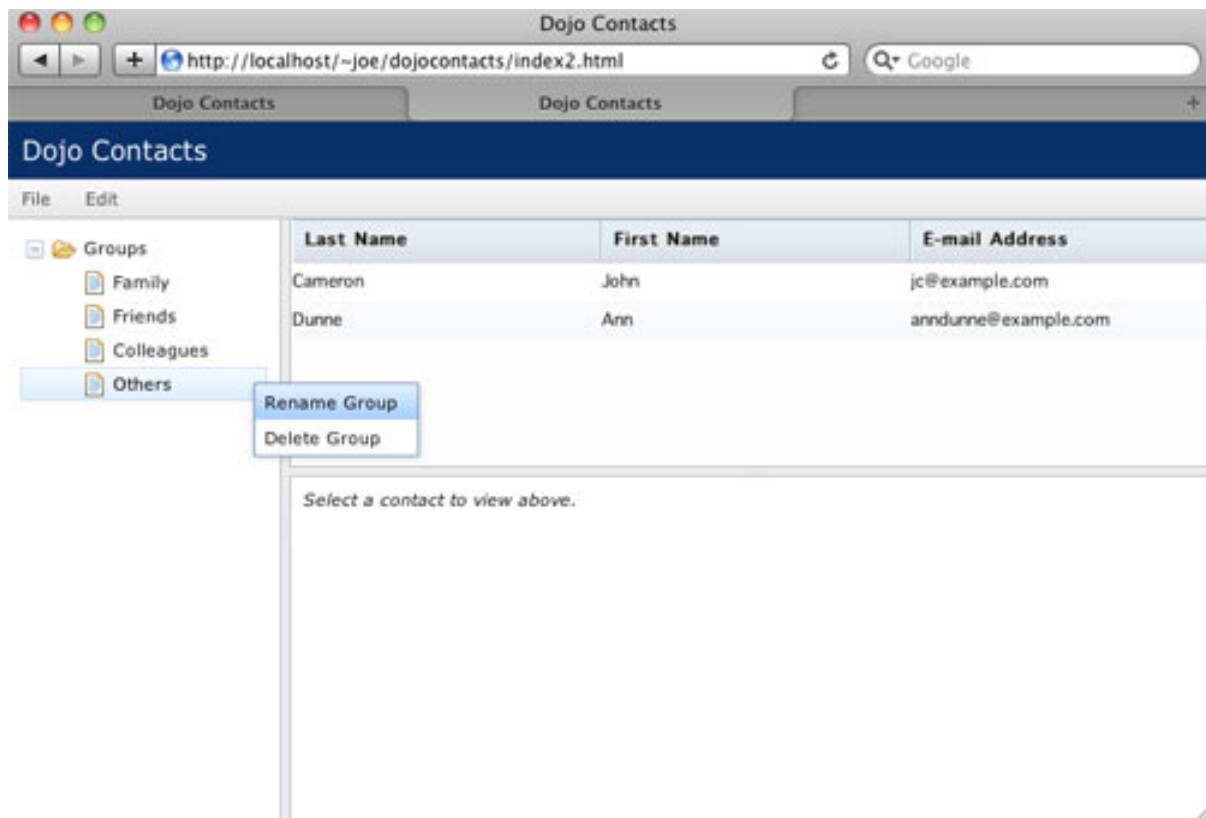
```

dojo.connect(groupsTree, "onClick", null, updateDataGrid);
dojo.connect(groupsTree, "onMouseDown", null, selectNode);

```

These events ensure that tree nodes get selected, even if you right-click a node. It also updates the data grid when you click on a tree node. Reload the application in your browser, and you should now be able to select a tree and see the relevant group contacts in the grid. You should also see that when you select a group (and not the `Groups` root node), the context menu options and application menu options become available. Of course, the menus don't actually do anything yet; you will see how to implement those a little bit later in the tutorial. Figure 6 shows the application with a group selected and the context menu in action.

Figure 6. Connected tree and data grid



Connecting the grid to the display pane

Next, you need to connect the grid and the main contact display pane (the bottom section on the right side of the interface). First, add the two functions given in Listing 19 to the script.js file, below the code you added from [Listing 18](#).

Listing 19. Updating the contact view from the grid

```
function selectRow(e) {
    if(e.rowIndex != null) {
        this.selection.clear();
        this.selection.setSelected(e.rowIndex, true);

        mnuEditContact.set("disabled", false);
        mnuMoveContact.set("disabled", false);
        mnuDeleteContact.set("disabled", false);
        ctxMnuEditContact.set("disabled", false);
        ctxMnuMoveContact.set("disabled", false);
        ctxMnuDeleteContact.set("disabled", false);
    }
}

function displayContact(id) {
    var item = this.getItem(id);
    var contactId = item.id;
    contactView.set("href", "data/contact.php?contact_id="+contactId);

    mnuEditContact.set("disabled", false);
}
```

```
mnuMoveContact.set("disabled", false);
mnuDeleteContact.set("disabled", false);
ctxMnuEditContact.set("disabled", false);
ctxMnuMoveContact.set("disabled", false);
ctxMnuDeleteContact.set("disabled", false);
}
```

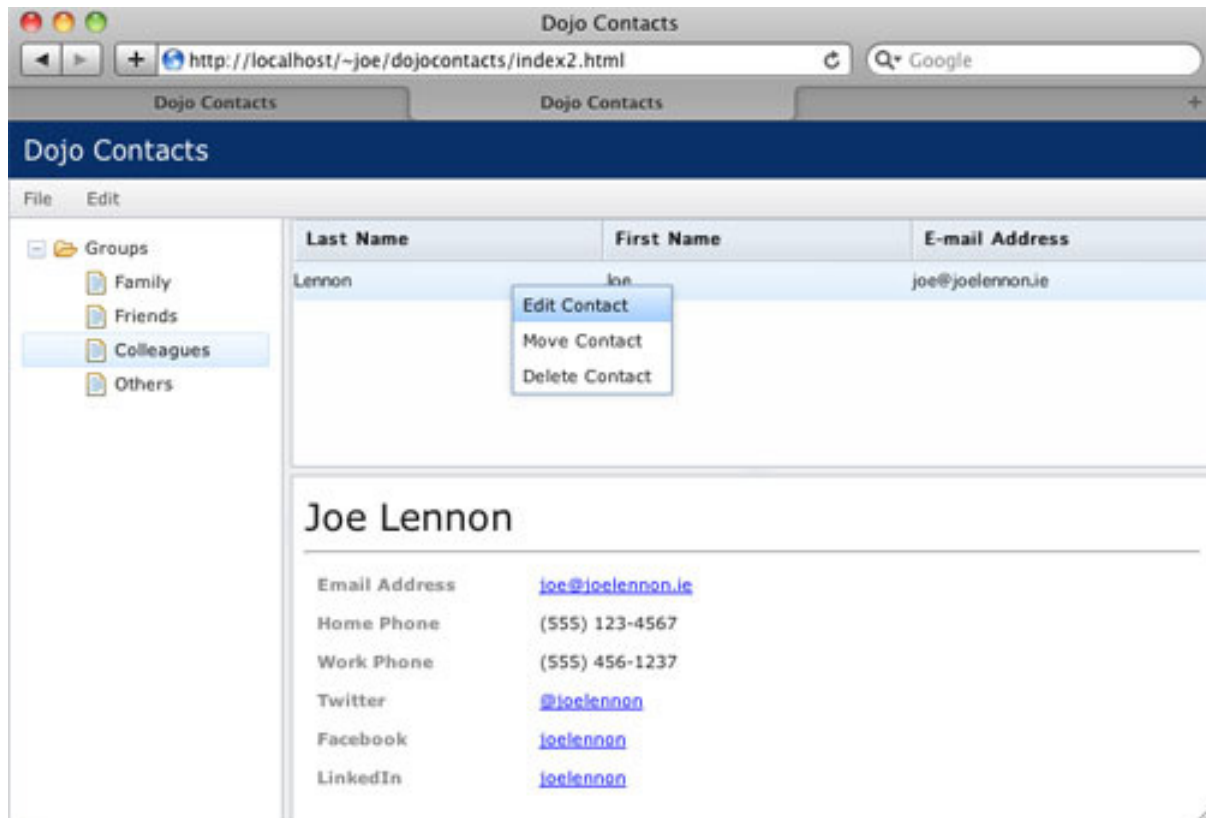
The `selectRow` function is used to set the row when a user right-clicks on the grid, and the `displayContact` function sets the `href` value of the `ContentPane` where the contact details are to be displayed. This content is loaded asynchronously from the PHP script `contact.php`, which is passed the ID of the selected contact. This function also enables menus as appropriate. These functions don't do anything unless they are connected to events, so let's wire them up now (see Listing 20).

Listing 20. Connecting the grid to the display pane

```
dojo.connect(contactsGrid, "onRowContextMenu", null, selectRow);
dojo.connect(contactsGrid, "onSelected", null, displayContact);
```

You can now reload the application in your browser and you should be able to select a contact in the grid and see the contact's details in the view pane. The relevant options should also be enabled in both the application and context menus. This is shown in Figure 7.

Figure 7. Connected grid and display pane



Section 6. Connecting menu options to dialog windows

Up until this point, none of the menu options in the application actually do anything. Similarly, there are a series of dialog windows that never appear. In this section, you will connect the menus to open these dialog windows accordingly.

Creating functions for configuring dialog windows

First up, create the functions given in Listing 21, which will configure the dialog windows correctly for the selected operation.

Listing 21. Functions for configuring dialog windows for selected operation

```
function renameGroup() {
    var group = groupsTree.get("selectedItem");
    var groupId = group.id;
    var groupName = group.name;

    dojo.byId("edit_group_id").value = groupId;
```

```

        dijit.byId("edit_group_old").set("value", groupName);
        editGroupDialog.show();
    }
    function refreshGroupDropDown() {
        var theStore = dijit.byId("edit_contact_group").store;
        theStore.close();
        theStore.url = "data/groups.php";
        theStore.fetch();
    }
    function newContact() {
        var contact = contactsGrid.selection.getSelected()[0];
        refreshGroupDropDown();
        dojo.byId("edit_contact_real_id").value = "";
        dojo.byId("edit_contact_id").value = "[NEW]";
        dijit.byId("edit_contact_group").reset();
        dijit.byId("edit_contact_first_name").reset();
        dijit.byId("edit_contact_last_name").reset();
        dijit.byId("edit_contact_email_address").reset();
        dijit.byId("edit_contact_home_phone").reset();
        dijit.byId("edit_contact_work_phone").reset();
        dijit.byId("edit_contact_twitter").reset();
        dijit.byId("edit_contact_facebook").reset();
        dijit.byId("edit_contact_linkedin").reset();

        dijit.byId("editContactDialog").set("title", "New Contact");
        dijit.byId("editContactDialog").show();
    }
    function editContact() {
        refreshGroupDropDown();
        var contact = contactsGrid.selection.getSelected()[0];
        dojo.byId("edit_contact_real_id").value = contact.id;
        dojo.byId("edit_contact_id").value = contact.id;
        dijit.byId("edit_contact_group").set("value", contact.group_id);
        dojo.byId("edit_contact_first_name").value = contact.first_name;
        dojo.byId("edit_contact_last_name").value = contact.last_name;
        dojo.byId("edit_contact_email_address").value = contact.email_address;
        dojo.byId("edit_contact_home_phone").value = contact.home_phone;
        dojo.byId("edit_contact_work_phone").value = contact.work_phone;
        dojo.byId("edit_contact_twitter").value = contact.twitter;
        dojo.byId("edit_contact_facebook").value = contact.facebook;
        dojo.byId("edit_contact_linkedin").value = contact.linkedin;

        dijit.byId("editContactDialog").set("title", "Edit Contact");
        dijit.byId("editContactDialog").show();
    }
    function moveContact() {
        var contact = contactsGrid.selection.getSelected()[0];
        var contactName = contact.first_name+" "+contact.last_name;
        var groupName = contact.name;

        dojo.byId("move_contact_id").value = contact.id;
        dojo.byId("move_contact_name").value = contactName;
        dojo.byId("move_contact_old").value = groupName;

        dijit.byId("moveContactDialog").show();
    }
}

```

Next, you need to connect these functions to the relevant menu options using `dojo.connect` (see Listing 22).

Listing 22. Connecting menu options to functions

```

dojo.connect(mnuNewGroup, "onClick", null, function(e) {
    newGroupDialog.show();
});

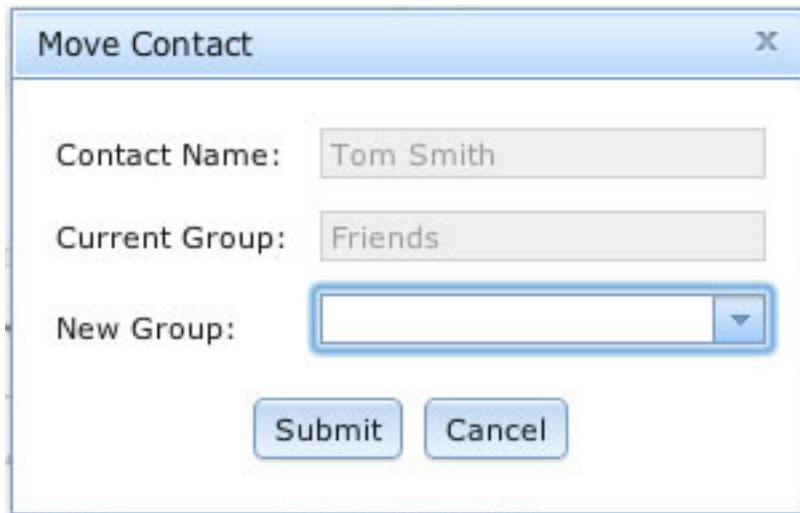
```

```
});  
dojo.connect(mnuRenameGroup, "onClick", null, renameGroup);  
dojo.connect(ctxMnuRenameGroup, "onClick", null, renameGroup);  
dojo.connect(mnuNewContact, "onClick", null, newContact);  
dojo.connect(mnuEditContact, "onClick", null, editContact);  
dojo.connect(ctxMnuEditContact, "onClick", null, editContact);  
dojo.connect(mnuMoveContact, "onClick", null, moveContact);  
dojo.connect(ctxMnuMoveContact, "onClick", null, moveContact);
```

In Listing 22, the New Group menu option is connected to an anonymous function, as it only has a single line to open the New Group dialog, so it doesn't really warrant a function of its own. Of course, if you prefer, you could put all of the functions used here in anonymous blocks like this, but I've separated them out, as it's easier to explain this way.

You can now reload the application and load any menu (except for the Delete options) and see the relevant dialog window open and load as appropriate. You'll probably notice right away that none of the buttons in these dialogs (except the hide "X" icon in the top right) actually do anything yet. You'll implement those features shortly. Figure 8 shows an example of the Move Contact window in action.

Figure 8. Move Contact dialog window working



Deleting groups and contacts

In the previous section, you learned how to connect all menus to their relevant dialogs, except for the delete actions. That is because the delete actions do not have a dialog window. Instead, the application should prompt the user to confirm that they want to delete a group or contact, and if they click the appropriate button, it should proceed with the deletion.

Dojo's shortcomings are few and far between, but, unfortunately, one of them is an out-of-the-box confirmation dialog box. Luckily, it's quite simple to build a custom

dialog that does just that. To create a confirmation dialog box, add the function shown in Listing 23 to the script.js file.

Listing 23. Function to create a confirmation dialog box

```
function confirmDialog(title, body, callbackFn) {
    var theDialog = new dijit.Dialog({
        id: 'confirmDialog',
        title: title,
        draggable: false,
        onHide: function() {
            theDialog.destroyRecursive();
        }
    });

    var callback = function(mouseEvent) {
        theDialog.hide();
        theDialog.destroyRecursive(false);

        var srcEl = mouseEvent.srcElement ? mouseEvent.srcElement : mouseEvent.target;

        if(srcEl.innerHTML == "OK") callbackFn(true);
        else callbackFn(false);
    };

    var message = dojo.create("p", {
        style: {
            marginTop: "5px"
        },
        innerHTML: body
    });
    var btnsDiv = dojo.create("div", {
        style: {
            textAlign: "center"
        }
    });
    var okBtn = new dijit.form.Button({label: "OK", id: "confirmDialogOKButton",
onClick: callback });
    var cancelBtn = new dijit.form.Button({label: "Cancel",
id: "confirmDialogCancelButton", onClick: callback });

    theDialog.containerNode.appendChild(message);
    theDialog.containerNode.appendChild(btnsDiv);
    btnsDiv.appendChild(okBtn.domNode);
    btnsDiv.appendChild(cancelBtn.domNode);

    theDialog.show();
}
```

This function accepts three arguments: the title to display in the dialog, the message to show, and the function that should be called back after the user clicks OK or Cancel. This function is called with a `true` value if OK is pressed and a `false` value if Cancel is pressed.

You can now use this function to create confirmation dialogs for the Delete Group and Delete Contact menu options. The results of deleting (and other CRUD operations you'll see in the next section) cause the `okDialog` box to be shown. So, let's get a handle to the message in that box now, while also getting the OK button to hide the dialog (see Listing 24).

Listing 24. Use the OK button to hide the dialog

```
var okDialogMsg = dojo.byId("okDialogMessage");
dojo.connect(okDialogOK, "onClick", null, function(e) {
    dijit.byId("okDialog").hide();
});
```

Listing 25 contains the code for the function that handles deleting groups.

Listing 25. deleteGroup function

```
function deleteGroup() {
    confirmDialog("Confirm delete", "Are you sure you wish to delete this group?
This will also delete any contacts in this group.<br />This action cannot
be undone.", function(btn) {
        if(btn) {
            var group = groupsTree.get("selectedItem");
            var groupId = group.id;
            var groupName = group.name;

            dojo.xhrPost({
                url: "data/delete_group.php",
                handleAs: "json",
                content: {
                    "group_id": groupId
                },
                load: function(data) {
                    if(data.success) {
                        groupsStore.fetch({
                            query: {"id": groupId.toString()},
                            onComplete: function (items, request) {
                                if(items) {
                                    var len=items.length;
                                    for(var i=0;i<len;i++) {
                                        var item = items[i];
                                        groupsStore.deleteItem(item);
                                    }
                                }
                            },
                            queryOptions: { deep: true}
                        });
                        groupsStore.save();

                        groupsTree.set("selectedItem", groupsModel.root);
                        updateDataGrid(groupsModel.root);
                        okDialog.set("title", "Group deleted successfully");
                        okDialogMsg.innerHTML = "The group <strong>" + groupName + "
</strong> was deleted successfully.";
                        okDialog.show();
                    }
                    else {
                        okDialog.set("title", "Error deleting group");
                        okDialogMsg.innerHTML = data.error;
                        okDialog.show();
                    }
                },
                error: function(data) {
                    okDialog.set("title", "Error deleting group");
                    okDialogMsg.innerHTML = data;
                    okDialog.show();
                }
            });
        }
    });
}
```



```

    });
}

```

There's a bit of new ground to cover here, but the `deleteContact` function is very similar, so let's add that now too, and I'll discuss both in tandem.

Listing 26. deleteContact function

```

function deleteContact() {
    var confirmed = false;
    confirmDialog("Confirm delete", "Are you sure you wish to delete this
contact?<br />This action cannot be undone.", function(btn) {
        if(btn) {
            var contact = contactsGrid.selection.getSelected()[0];
            var contactId = contact.id;
            var contactName = contact.first_name+" "+contact.last_name;

            dojo.xhrPost({
                url: "data/delete_contact.php",
                handleAs: "json",
                content: {
                    "contact_id": contactId
                },
                load: function(data) {
                    if(data.success) {
                        var treeSel = groupsTree.get("selectedItem");
                        var groupId;
                        if(treeSel) {
                            groupId = treeSel.id;
                        } else {
                            groupId = 0;
                        }
                        var url = contactsStore.url+"?group_id="+groupId;
                        var newStore = new dojo.data.ItemFileReadStore({url:url});
                        contactsGrid.setStore(newStore);
                        refreshGrid();

                        okDialog.set("title","Contact deleted successfully");
                        okDialogMsg.innerHTML = "The contact <strong>"+
+contactName+"</strong> was deleted successfully.";
                        okDialog.show();
                    }
                    else {
                        okDialog.set("title","Error deleting contact");
                        okDialogMsg.innerHTML = data.error;
                        okDialog.show();
                    }
                },
                error: function(data) {
                    okDialog.set("title","Error deleting contact");
                    okDialogMsg.innerHTML = data;
                    okDialog.show();
                }
            });
        }
    });
}

```

Both of these functions use the new confirmation dialog function to ask the user to confirm the deletion. Deleting a group will cascade delete any contacts in that group, so the user is warned about that in the case of attempting to delete a group. If the

user confirms, the functions go to either the group's tree or contacts grid as required and gets the detail of the item that needs to be deleted. An Ajax `dojo.xhrPost` function call is then used to asynchronously call the relevant server-side PHP API. The JSON response received from this is then parsed and used to display a relevant success or error message.

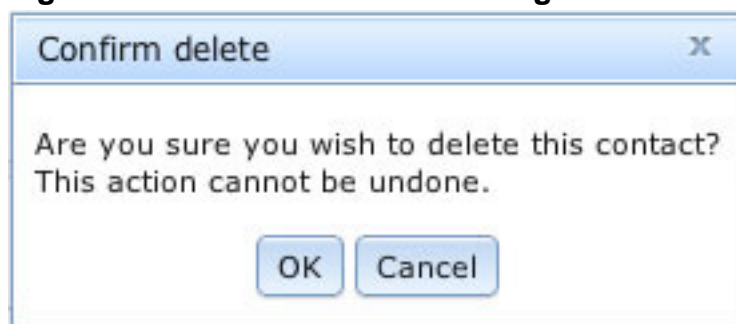
Finally, you need to connect these functions to the relevant events (that is, the menu options). The code in Listing 27 does just that.

Listing 27. Connecting menu options to delete functions

```
dojo.connect(mnuDeleteContact, "onClick", null, deleteContact);
dojo.connect(ctxMnuDeleteContact, "onClick", null, deleteContact);
dojo.connect(mnuDeleteGroup, "onClick", null, deleteGroup);
dojo.connect(ctxMnuDeleteGroup, "onClick", null, deleteGroup);
```

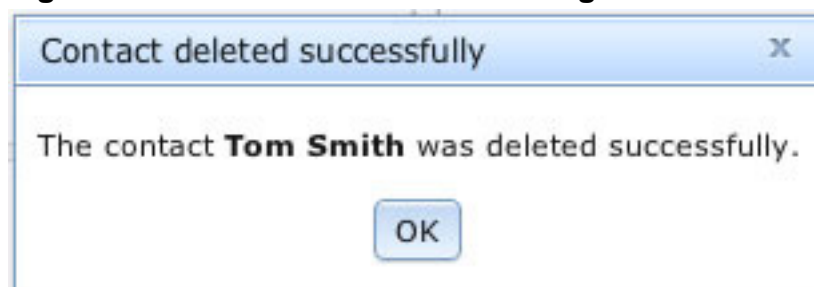
You can now save and reload your application. If you try to delete a contact, you should see the message shown in Figure 9.

Figure 9. Delete confirmation dialog box



Pressing OK will actually go ahead and delete the contact, and the following message will subsequently be displayed (see Figure 10).

Figure 10. Deletion successful message



You'll also notice that the relevant contact has been removed from the underlying grid. If you delete a group, you should see similar functionality (except that deleting a group also deletes all the contacts in that group).

Section 7. Saving data from dialog boxes using Ajax

The only function that has yet to be added to the application is the implementation of the dialog boxes for creating new groups and contacts, renaming groups, and editing and moving existing contacts. Let's wrap up the application by implementing these functions now.

In the previous section, you learned how to implement the delete function. This used the `dojo.xhrPost` function to call a PHP script using Ajax. You will be adding similar function calls to perform the other operations, except you will tap into the form submit actions to override the default functions of the forms.

Adding and renaming groups

Let's start with the adding and renaming of groups. Add the two functions in Listing 28 to your `script.js` file.

Listing 28. Functions to add and rename groups

```
function doNewGroup(e) {
    e.preventDefault();
    e.stopPropagation();
    dojo.byId("new_group_ajax").value = "1";
    if(this.isValid()) {
        dojo.xhrPost({
            form: this.domNode,
            handleAs: "json",
            load: function(data) {
                if(data.success) {
                    okDialog.set("title", "Group created successfully");
                    okDialogMsg.innerHTML = "The group <strong>"+data.name+
"</strong> was created successfully.";

                    groupsStore newItem({"id":data.id.toString(), "name":data.name},
{"parent": groupsModel.root, "attribute":"groups"});
                    groupsStore.save();

                    newGroupDialog.hide();
                    okDialog.show();
                }
            },
            error: function(error) {
                okDialog.set("title", "Error creating group");
                okDialogMsg.innerHTML = error;
                okDialog.show();
            }
        });
    }
}
```

```

    });
  }
}

//Process the editing of an existing group in the database
function doEditGroup(e) {
  e.preventDefault();
  e.stopPropagation();
  dojo.byId("edit_group_ajax").value = "1";
  if(this.isValid()) {
    dojo.xhrPost({
      form: this.domNode,
      handleAs: "json",
      load: function(data) {
        if(data.success) {
          okDialog.set("title","Group renamed successfully");
          okDialogMsg.innerHTML = "The group <strong>"+data.name+
"</strong> was renamed successfully.";

          var group = groupsTree.get("selectedItem");
          groupsStore.setValue(group, "name", data.name);
          groupsStore.save();

          editGroupDialog.hide();
          okDialog.show();
        }
        else {
          okDialog.set("title","Error renaming group");
          okDialogMsg.innerHTML = data.error;
          okDialog.show();
        }
      },
      error: function(error) {
        okDialog.set("title","Error renaming group");
        okDialogMsg.innerHTML = error;
        okDialog.show();
      }
    });
  }
}
}
}

```

These functions will be called when the user tries to submit the New Group or Rename Group forms. They override the default submit action and update a flag to say that the form is being submitted using Ajax and should receive a JSON response. An XHR POST is then performed to the URL in the form's `action` attribute. When the response is received, the tree is updated accordingly, and a success dialog message is displayed. Before you can test the functions, however, you need to connect them to the relevant forms. This is shown in Listing 29.

Listing 29. Connecting form submit to relevant functions

```

dojo.connect(newGroupDialog, "onShow", null, function(e) {
  dijit.byId("new_group_name").reset();
});
dojo.connect(newGroupForm, "onSubmit", null, doNewGroup);
dojo.connect(newGroupCancel, "onClick", null, function(e) {
  newGroupDialog.hide();
});

dojo.connect(editGroupDialog, "onShow", null, function(e) {
  dijit.byId("edit_group_name").reset();
});

```

```

dojo.connect(editGroupForm, "onSubmit", null, doEditGroup);
dojo.connect(editGroupCancel, "onClick", null, function(e) {
    editGroupDialog.hide();
});

```

When the dialog boxes are displayed, the form values will be reset. Also, when you click Cancel in a dialog box, the dialog is hidden. You should now be able to add and rename groups in the application.

Adding, editing, and moving contacts

Adding and editing contacts are actually done using the same dialog, and you have already provided the functions to distinguish between new and existing contacts. As a result, only a single function is required to take care of this operation. You will also create a function now for handling moving contacts from one group to another. These functions are defined in Listing 30.

Listing 30. Functions to add, edit, and move contacts

```

function doMoveContact(e) {
    e.preventDefault();
    e.stopPropagation();
    dojo.byId("move_contact_ajax").value = "1";
    if(this.isValid()) {
        dojo.xhrPost({
            form: this.domNode,
            handleAs: "json",
            load: function(data) {
                if(data.success) {
                    okDialog.set("title", "Contact moved successfully");
                    okDialogMsg.innerHTML = "The contact was moved successfully.";

                    var treeSel = groupsTree.get("selectedItem");
                    var groupId;
                    if(treeSel) {
                        groupId = treeSel.id;
                    } else {
                        groupId = 0;
                    }
                    var url = contactsStore.url+"?group_id="+groupId;
                    var newStore = new dojo.data.ItemFileReadStore({url:url});
                    contactsGrid.setStore(newStore);
                    refreshGrid();

                    moveContactDialog.hide();
                    okDialog.show();
                }
            },
            error: function(error) {
                okDialog.set("title", "Error moving contact");
                okDialogMsg.innerHTML = error;
                okDialog.show();
            }
        });
    }
}

```

```

    });
  }
}

//Process the editing of an existing contact in the database
function doEditContact(e) {
  e.preventDefault();
  e.stopPropagation();
  dojo.byId("edit_contact_ajax").value = "1";
  if(this.isValid()) {
    dojo.xhrPost({
      form: this.domNode,
      handleAs: "json",
      load: function(data) {
        if(data.success) {
          if(data.new_contact) {
            okDialog.set("title", "Contact added successfully");
            okDialogMsg.innerHTML = "The contact was added successfully.";
          } else {
            okDialog.set("title", "Contact edited successfully");
            okDialogMsg.innerHTML = "The contact was edited successfully.";
          }
        }

        var treeSel = groupsTree.get("selectedItem");
        var groupId;
        if(treeSel) {
          groupId = treeSel.id;
        } else {
          groupId = 0;
        }
        var url = contactsStore.url+"?group_id="+groupId;
        var newStore = new dojo.data.ItemFileReadStore({url:url});
        contactsGrid.setStore(newStore);
        refreshGrid();

        editContactDialog.hide();
        okDialog.show();
      },
      error: function(error) {
        okDialog.set("title", "Error editing contact");
        okDialogMsg.innerHTML = error;
        okDialog.show();
      }
    });
  }
}

```

These functions work in a similar fashion to their corresponding "group" functions, so I won't go into too much detail. Rather than update the tree, however, these functions reload the grid control asynchronously. It's also worth noting that the `doEditContact` function has some additional logic to determine if the user has just added a new contact or edited an existing one, so that the correct message can be displayed to the user.

The final bit of code that's required is the code to connect these functions to the relevant forms. The code for this is shown in Listing 31.

Listing 31. Connecting the add, edit, and move contact actions

```
dojo.connect(moveContactDialog, "onShow", null, function(e) {
    var theStore = dijit.byId("move_contact_new").store;
    theStore.close();
    theStore.url = "data/groups.php";
    theStore.fetch();
    dijit.byId("move_contact_new").reset();
});
dojo.connect(moveContactForm, "onSubmit", null, doMoveContact);
dojo.connect(moveContactCancel, "onClick", null, function(e) {
    moveContactDialog.hide();
});

dojo.connect(editContactForm, "onSubmit", null, doEditContact);
dojo.connect(editContactCancel, "onClick", null, function(e) {
    editContactDialog.hide();
});
```

When the Move Contact dialog is shown, it reloads the store behind the drop-down list of groups so that it includes any newly added, renamed groups, and excludes any deleted ones. With this code added, you can save your application and load it in your browser to see it working in all its glory.

Suggested improvements

Although the sample application created in this tutorial is fully functional, there are a number of features that could be added to enhance the user experience and make the application even more impressive. Unfortunately, I did not have the scope to add these in this tutorial, but they should be relatively straightforward to add should you want to do so yourself. Some of these enhancements might include:

- Allow subgroups to be added to groups
- Allow contacts to be added to more than one group
- Drag and drop moving of contacts
- Re-ordering groups with drag and drop
- Flexible contact fields (let users add their own fields for individual contacts)
- Inline contact editing
- Inline grid editing
- Multiple row selection and deletion

- Add a "Trash" feature for deleted items, allowing for recovery and drag and drop deletion
 - Implement Dojo and PHP object orientation to tidy up code
 - Allow the user to move contacts to another group when deleting groups, rather than just deleting them
 - Use MySQLi in PHP scripts for additional security
 - Add more complex validation rules
 - Add log-in system to allow for more than one user
-

Section 8. Summary

This tutorial was centered on the creation of a full-featured sample application that lets you manage contacts. It provided a step-by-step guide to creating a user interface using Dojo widgets, and communicating with a MySQL database using XHR/Ajax calls to a series of PHP API scripts. The tutorial should have given you a head start so that you can take the sample application and amend it to create your own complex Dojo applications.

Downloads

Description	Name	Size	Download method
Tutorial source code	dojo.ajax.tutorial.source.zip	13KB	HTTP

[Information about download methods](#)

Resources

Learn

- Read the three-part series, "[Dojo from the ground up](#)" to get started with Dojo development.
- Visit the home page for the [Dojo Toolkit](#).
- Check out some [Dojo Toolkit Demos](#).
- [Introducing The Dojo Toolkit](#): Read an excellent introduction to the Dojo Toolkit from the Opera developer site.
- Read the [Dijit](#) page from the Dojo documentation site.
- Learn about [Dijit Themes and Theming](#) from the Dojo documentation site.
- [Introduction to the Dojo toolkit, Part 1: Setup, core, and widgets](#): Read another fine introduction to the Dojo toolkit from Javaworld.
- [Dojo 1.5: Ready to power your web app](#): Learn about some of the new features in Dojo 1.5 from this article on Sitepen.
- [Introduction to the Dojo Toolkit: Tutorial](#): Read an introductory tutorial from Ajax Matters.
- Read about [Dijit's TabContainer Layout: Easy Tabbed Content](#) on David Walsh's blog.
- Read [A Localization Primer for Dojo Dijit and Dojox Controls](#), by Rob Gravelle, for an overview of requirements and general considerations to localize a web form using Dojo's Dijit and Dojox widgets.
- [Basic Dijit knowledge in Dojo](#), by Peter Svensson, is another excellent introduction to Dijit.
- "[Internationalizing web applications using Dojo](#)" (developerWorks, August 2008): Discover a way to perform native language support in the context of web sites and web applications using the i18n feature of the Dojo toolkit.
- "[Consuming web services with the Dojo Toolkit](#)" (developerWorks, September 2010): Learn how to consume services using the Dojo Toolkit to enable Ajax on a web page.
- [IBM - Dojo Extension sample](#): The Dojo Extension Feature Set can be used to enable a IBM WebSphere Portlet Factory model to leverage functionality provided by the Dojo JavaScript Toolkit.
- Read Nathan Toone's entry on [Understanding dojo.declare, dojo.require, and dojo.provide](#) from [DojoCampus.org](#).

- Read [Dojo Confessions \(Or: How I gave up my jQuery Security Blanket and Lived to Tell the Tale\)](#) by Rebecca Murphey to learn how the author made the switch from jQuery to Dojo.
- [Enterprise Dojo](#) by Dan Lee shows you how to write highly cohesive code with JavaScript using Dojo.
- [Dojo: Using the Dojo JavaScript Library to Build Ajax Applications](#) by James E. Harmon from Addison-Wesley Professional.
- [Getting Started with Dojo](#) by Kyle Hayes and Peter Higgins from friends of ED.
- ["Writing a custom Dojo application"](#) (developerWorks, December 2008): Find out much more about Dojo in this developerWorks article.
- ["Develop HTML widgets with Dojo"](#) (developerWorks, October 2006): Explore Dojo's extensibility in this developerWorks article.
- ["Using the Dojo Toolkit with WebSphere Portal"](#) (developerWorks, November 2007): Learn how to install, configure, use, and leverage the Dojo Toolkit in WebSphere Portal applications.
- The developerWorks [Web Development zone](#) specializes in articles covering various web-based solutions.

Get products and technologies

- Download the [Dojo Toolkit](#). Version 1.5 was used in this article.
- Get the [Apache web server](#).
- Get [MySQL](#), version 4 or later.
- Get [PHP](#), version 5 or later
- Access the [Dojo Toolkit API documentation](#).
- Download [IBM product evaluation versions](#), and get your hands on application development tools and middleware products from DB2, Lotus, Rational, Tivoli, and WebSphere.

Discuss

- Get involved in the [My developerWorks community](#). Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Joe Lennon

Joe Lennon is a 25-year-old mobile and web application developer from Cork, Ireland. Joe works for Core International, where he leads the development of Core's mobile HR self service solutions. Joe is also a keen technical writer, having written many articles and tutorials for IBM developerWorks on topics such as DB2 pureXML, Flex, JavaScript, Adobe AIR, .NET, PHP, Python and much more. Joe's first book, *Beginning CouchDB* was published in late 2009 by Apress. In his spare time, Joe enjoys travelling, reading and video games.