Building CouchApps

Create web applications stored in an Apache CouchDB database

Martin Brown

May 03, 2011

Apache CouchDB is an open source document-oriented database management system that allows you to create full database-driven applications using nothing but HTML, CSS and JavaScript. In this tutorial, you will learn how to create your own CouchApp that will perform database operations using Ajax powered by the jQuery framework. The application you will build is a contact manager that allows you to view, create, edit, and delete your contacts.

Before you start

This tutorial is for web application developers interested in creating database-driven applications using nothing but HTML, CSS, and JavaScript. You should know how to write JavaScript and how to manipulate the Document Object Model (DOM) of an HTML page using JavaScript. You should also have some experience using a library tool, such as jQuery or Dojo.

About this tutorial

Apache CouchDB is an open source document-oriented database management system that stores data as JSON objects. Traditional database systems allow you to perform data retrieval and update functions using a series of SQL statements that are executed through some form of proprietary client software or API. Apache CouchDB is different — you send your queries or updates using a RESTful HTTP API, making it simple to communicate with Apache CouchDB in virtually any modern programming language.

Because of the architecture Apache CouchDB is built on, it is actually possible to build entire web applications that reside inside an Apache CouchDB database. We call these applications CouchApps. CouchApps allow you to create full database-driven applications using nothing but HTML, CSS and JavaScript. The beauty of these apps is that they allow you to take full advantage of Apache CouchDB’s powerful replication features to replicate your CouchApp across Apache CouchDB instances. This allows you to keep your CouchApp on several devices, and synchronize them, with automated incremental replication keeping your data up-to-date on each device.

In this tutorial, you will learn how to create your own CouchApp using HTML, CSS, and JavaScript. Your application will perform database operations using Ajax powered by the jQuery framework.
The application you will build is a contact manager that allows you to view, create, edit, and delete your contacts. Finally, you will learn how to replicate this application between two Apache CouchDB instances.

**Prerequisites**

You will need the following tools to follow along with this tutorial:

- An Apache CouchDB database instance, v1.0.1 or higher
- The CouchApp tool, version 0.7.0 or higher

See [Related topics](#) for download information and [Downloadable resources](#) for the source code of our sample application.

**Introducing Apache CouchDB and CouchApps**

In this section we'll look at the advantages of using Apache CouchDB over a traditional database solution.

**Apache CouchDB versus traditional database solutions**

Apache CouchDB is a database system that works differently compared to traditional database solutions such as IBM DB2, Oracle, or MySQL. In place of the structured format of databases, tables, and columns offered by these solutions, Apache CouchDB instead works by storing documents. Documents are free-form structures, which means that you can have any combination of fields, and field structures, and this can be different for every document in the database.

For example, with a traditional relational database you might store the basic information about a contact by defining a contact table using the statement shown in **Listing 1**.

**Listing 1. Storing basic contact information in a traditional relational database**

```sql
CREATE TABLE contact (  
    id int(11) NOT NULL AUTO_INCREMENT,  
    title char(20) DEFAULT NULL,  
    firstname char(20) DEFAULT NULL,  
    lastname char(20) DEFAULT NULL,  
    PRIMARY KEY (id)
)
```

This places a rigid structure on your data, which can be both useful and restricting at the same time. For example, what happens if you need to add a middle name to the table? You would need to add a new field to accommodate the new data. Another element to consider is that additional information — for example, the telephone numbers used to contact an individual — would probably exist in another table. To get the phone number for an individual you would need to perform a query with a join (or subquery) to match the base contact table to the phone number table. You would have to do the same with other data points, such as addresses and email accounts, or with more flexible data, such as important dates, spouses, and other links.
Storing information in Apache CouchDB

With Apache CouchDB, information is instead stored in documents, and the documents are freeform, written using JavaScript Object Notation (JSON), allowing you to construct a document that contains lists, hashes, as well as traditional fields, all into a single document. For example, Listing 2 shows a contact in JSON format.

Listing 2. Contact information in JSON format

```json
{
  "firstname": "Martin",
  "address": {
    "home": [
      "Some road",
      "Some town",
      "Postcode",
      "Country"
    ],
  },
  "title": "Mr",
  "lastname": "Brown",
  "phones": {
    "home": "09874978",
    "mobile": "0892374908"
  }
}
```

In Listing 2, all of the information about the contact is in a single document. However, the document structure is not fixed in any way. Listing 3 shows a different contact from the same database.

Listing 3. Different contact

```json
{
  "email": {
    "work": "sample@example.com",
    "home": "other@example.com"
  },
  "firstname": "Paulie"
}
```

Don't think that the use of such a freeform structure for your database content means that you lose the ability to enforce a structure. You can use a validation routine that can check not only the structure of the document, but also the contents of those fields.

Documents within Apache CouchDB are stored using a document ID. You can use any string as a document ID, so your contact could be stored with the document ID 'MartinBrown', or you can allow Apache CouchDB to create a UUID (Universally Unique ID).

One final element of the Apache CouchDB system is that unlike traditional databases, you do not need a special library or interface system to access or update the data. Instead, the entire interface is built around a REST-like interface accessible through anything that can access a web page over HTTP. Thus, we can access the document MartinBrown, stored within the database 'contacts', on the machine 'Apache CouchDB' by opening a web browser and accessing
http://127.0.0.1:5984/contacts/MartinBrown. The contact ‘Paulie’ would be stored within the URL http://127.0.0.1:5984/contacts/Paulie.

**What is a CouchApp?**

This simple web interface to the database also provides the basis of the CouchApp. A CouchApp is an HTML5 and JavaScript-based application to the documents stored within an Apache CouchDB database. The documents and code that make up the interface and application are also stored within Apache CouchDB as *design documents*. The result is an application (including display elements) that can be entirely self-contained within the database that provides the data, making the entire process of building and interacting with your application focused on the information that you want to present.

The use of JavaScript as a core part of CouchApps also extends to the server, where JavaScript is used to construct views of your database. In a traditional database, like Oracle, the SQL and the database structure provide the ability to pull out information. This makes it easy to pull out a list of all the records where the `firstname` field is 'Martin'. However, with Apache CouchDB, the data is stored in documents, not tables. To achieve the same result you would need to open every document in the database, work out if it contained the specified field, and then add the document to a list if the field matched what you wanted. This is a time (and CPU) expensive process, especially if your database contains thousands or even millions of documents.

To improve the performance for these types of operation, Apache CouchDB uses views. Views perform the operation of iterating over every document and building a list of the documents with specific fields. The views are built on the server, and the resulting view is stored on disk as an index to the underlying documents. This improves the performance when retrieving lists of documents in this fashion, and makes it easy to pull out records according to whether they match a specific field value.

To make the entire process of building a CouchApp easier, there is a command-line tool called CouchApp that can create stub and template code for your Apache CouchDB application, while creating files on the local filesystem that you can then edit and 'push' to your Apache CouchDB server using the CouchApp command line tool. This simplifies the entire process and means that you can concentrate on building the application without worrying about uploading the application to Apache CouchDB.

There are some additional differences that we won't describe in detail here, but that we will cover in part as we go through the rest of the tutorial. For example, documents can also include attachments (one or more files associated with the document), and all document revisions are stored, with each update to a document forming a new 'revision'. The CouchApp tools hide this complexity to make the entire system easier to use.

**Installing CouchApp**

In this section we will install and configure CouchApp
Installing Apache CouchDB

Before installing CouchApp, you need to install Apache CouchDB. You can download Apache CouchDB in a variety of formats, including as source, which you can build yourself, or as a standalone application for running on Windows™, Mac OS X and Linux®. For example, the Mac OS X application, Apache CouchDBX, runs as a standard application.

On Linux and UNIX®, you will get a binary, couchdb, which you can run from the command-line (see Listing 4).

Listing 4. Running the couchdb binary from the command-line

```bash
$ couchdb
Apache CouchDB 1.0.1 (LogLevel=info) is starting.
Apache CouchDB has started. Time to relax.
[info] [<0.33.0>] Apache CouchDB has started on http://127.0.0.1:5984/
```

You are now ready to go! You can check that the database is running by accessing the given URL. If you want to open up your server so that it can be accessed over the network, change the bind parameter within the local.ini configuration file to match the IP address (not hostname) of your Apache CouchDB server.

Installing CouchApp

For the CouchApp command line tool, you need to download the CouchApp tar or Zip package from GitHub (see Related topics). You will need an installation of Python on your machine (if you do not already have it installed), but the CouchApp installer will handle all of the dependencies of additional libraries for you.

After you have downloaded the package, extract it, and then change into the CouchApp directory:

```
$ cd couchapp
```

Now run the Python setup tool to download and install any dependencies and install the couchapp tool:

```
$ python setup.py install
```

You can test the installation by running CouchApp, which should return the help information for the tool:

```
$ couchapp
```

Note: Normally you would not create a database that allows anybody to access and update it. Apache CouchDB does support authentication and different levels of security and authority for performing different operations, but we do not cover them in this tutorial.

Configuring your database and creating your CouchApp

As a good way of understanding the simplicity of Apache CouchDB, you can create a new database within your Apache CouchDB instance from the command using the curl command line tool. To create a database, you issue a PUT HTTP command to the URL of the database you want to create. For example, to create a contact database, you could use the command line:

```
$ curl -X PUT http://127.0.0.1:5984/contacts
```
If you check the content of the file contacts, it should specify that the operation completed successfully.

Alternatively, go into Futon, using http://127.0.0.1:5984/_utils. You can create a new database using the Futon administration interface.

You now have an empty database. To create a stub application, you can use CouchApp to generate all of the basic files that you need on your file system ready to be uploaded to your Apache CouchDB database. You can do this by running: $ couchapp generate app contacts.

This creates a directory, contacts, which contains an array of files and contents that we can use to build our contacts application. You can see a top-level file list in Listing 5.

### Listing 5. Top-level file list

```bash
$ ls -al contacts/
total 605-
drwxrwxrwx 9 mcco mcco 4096 Dec 1 14:49 ./
drwxrwxrwx 3 mcco mcco 4096 Dec 1 14:49 ../
-rw-rw-rw- 1 mcco mcco  174 Dec 1 14:49 .couchappignore
-rw-rw-rw- 1 mcco mcco 1660 Dec 1 11:51 README.md
-drwxrwxrwx 3 mcco mcco 4096 Dec 1 14:49 _attachments/
 -rw-rw-rw- 1 mcco mcco  10 Dec 1 11:51 language
-drwxrwxrwx 2 mcco mcco 4096 Dec 1 14:49 evently/
-drwxrwxrwx 2 mcco mcco 4096 Dec 1 14:49 shows/
-drwxrwxrwx 2 mcco mcco 4096 Dec 1 14:49 updates/
-drwxrwxrwx 3 mcco mcco 4096 Dec 1 14:49 vendor/
-drwxrwxrwx 3 mcco mcco 4096 Dec 1 14:49 views/
```

Some of the major elements of this are:

- **views**— contain the Views on the database. These are JavaScript functions that build a list of keys and data that you want returned, the equivalent of a typical database query.
- **lists**— contain lists that are used to build formatted versions of the view output. Lists are JavaScript functions that take the information from a view and format the information (usually as HTML) for display.
- **shows**— display a single document, rather than a list of documents as provided by a view. Like lists, a show is defined as a JavaScript function.
- **attachments**— contain attachments for the application, including index.html and JavaScript files.

Views are critical to the way you access information from the database when you don’t know the document ID of the document you want to load. Lists and shows provide built-in methods for displaying information. However, you use other methods, such as the jQuery library, to get information out of the database using the view to return the list of documents that is then processed by the jQuery Apache CouchDB library.
Editing index.html

The default document, index.html, is contained within the database attachments directory, contacts/_attachments/index.html. You should edit this to contain some default links to the database.

To make the best use of the environment offered by jQuery and Apache CouchDB we will define the entire interface for the Contacts application dynamically. This will use JavaScript to dynamically provide the different elements, include displaying the results of the view and the forms used to edit the contact information.

For that to work, you need to edit the index.html file to that shown in Listing 6.

Listing 6. Editing the index.html file

```html
<!DOCTYPE html>
<html>
<head>
<title>Contacts</title>
<link rel="stylesheet" href="style/main.css" type="text/css">
<script src="vendor/couchapp/loader.js"></script>
<script src="recordedit.js"></script>
</head>
<body>
<div id="account"></div>
<h1>Contacts</h1>
<div id="items"><div id="add"><a href="#" class="add">Add Contact</a></div>
<div id="contacts"></div>
<div id="contactform"></div>
</div>
</body>
</html>
```

The key elements of structure are:

- The loading of the vendor/couchapp/loader.js script. This in turn loads the jQuery and jQuery Couch libraries, among others.
- The loading of the recordedit.js script. This is the script we will populate with the JavaScript functions used to build the application.
- A button that will be used to trigger the Add form for creating a new contact.
- A div element, with the id contacts, that will be used to display the contact list.
- A div element, with the id contactform, that will be used to display the contact form.

Once you have edited the file, you need to push the application to your Apache CouchDB database using the CouchApp command-line tool: $ couchapp push contacts http://127.0.0.1:5984/contacts.

The first argument is the instruction to push (publish) the application, the second is the local directory, contacts, where the application is stored, and the third is the URL of the Apache CouchDB database where you want to upload the database. Once the push has completed
successfully, you can view the uploaded application using the URL: http://127.0.0.1:5984/contacts/_design/contacts/index.html.

It is worth dissecting this URL:

- `127.0.0.1:5984` is the hostname, and port number, of the Apache CouchDB server. By default servers run on port 5984.
- `contacts` is the name of the database.
- `_design` is a special identifier to Apache CouchDB that indicates you want to access the design document. Design documents contain the view, list, and show definitions. You can have more than one design document for a given database.
- `contacts` is the name of the design document. The CouchApp tool creates design document with the same name as your application by default.
- `index.html` is the name of the attachment for the contacts design document.

CouchDB also includes a rewriting module that will simplify these URLs to something more friendly. See Related topics for an article on this topic.

With the basic document in place, you can start to build the rest of the application.

**Displaying a list of contacts**

In this section, we will create and display a list of contacts.

**Creating a view**

To construct a list of contacts, you first need to create a view. This is a JavaScript function that accepts a document as the only argument. The function is executed by Apache CouchDB on every document in the database, and it should output keys (which are used for displaying and filtering information) and corresponding values that you want to output for each document in this view. This process is called the map, as you are mapping the document contents to the information that you want to extract. There is another step, called reduce, which can be used to summarize or simplify the information, but we will not need that for a contacts application.

For example, to output the name from the contact document as the key, and the entire contact record as the value, you would need to write the view shown in Listing 7.

**Listing 7. Writing the view**

```javascript
function(doc) {
    if (doc.name) {
        emit(doc.name, doc);
    }
}
```

The anonymous function accepts a single argument, the document. The function then checks to ensure the record has a field called name, and if true, the `emit()` function then returns two values: The first is the key, and the second is the value, in this case, a copy of the document. Both keys and values can be any valid JSON structure. Keys are used by Apache CouchDB during searching.
and paging. The values merely contain the information that you want to expose in this view when it is accessed.

Within CouchApp, you can create a new view on the command line using the generate command:

```
$ couchapp generate view contacts byname
```

This creates the view `byname`, in the directory `contacts/views/byname`, and creates two files, `map.js` and `reduce.js`. Edit the `map.js` file and change it to the function in Listing 6.

You can now push the application to your Apache CouchDB database again. Views are accessible through the browser interface. You can access the view `byname` on the contacts design document by accessing the URL `http://127.0.0.1:5984/contacts/_design/contacts/_view/byname`. Again, we are using the design document contact, this time requesting the output of a view (identified by the `_view`) in the path, with the view name of `byname`.

At this stage, the view will be empty: `{"total_rows":0,"offset":0,"rows":[]}`.

### Displaying the view within the application

To display the view within our application, we can use the jQuery Couch library to access the view, iterate over each record returned by the view, and then print the record information.

A function for this is shown in **Listing 8**.

**Listing 8. Displaying the view within our application**

```javascript
db = $.couch.db("contacts");
function updatecontacts() {
  "contacts".empty();

db.view("contacts/byname", {success: function(data) {
  for (i in data.rows) {
    "contacts".append('<div id="' + data.rows[i].value._id + '" class="contactrow"><span>' + data.rows[i].value.name + '</span><span>" + data.rows[i].value.phone + '</span><span>" + ' + data.rows[i].value._id + '" + data.rows[i].value._id + ' + Edit Contact</a>' + ' + Remove Contact</a>' + ' + "</div></div>
  }
}
});
```

The first line in **Listing 7** sets a variable used to access the database. The `updatecontacts()` function first empties the `div` element that will be used to display the contacts list, then it accesses the results of the view that was just created. If the view access was successful, an anonymous function is called with the returned view data as a JSON structure. The function then iterates over the content, and builds a contact row that outputs the contact name and phone number.
The view results is represented as an array (rows), with each element of the array being a JSON structure containing the contents of the key returned by the view, and the value returned by the view. Hence, we can access the phone number of a contact record returned by the view by accessing the `value.field` portion of the array element.

The output produces an Edit Contact and a Remove Contact link, which also includes the ID of the underlying Apache CouchDB document. This will be used to provide the information when updating and deleting a contact.

To add this function to the contacts application, create a new file, `contacts/_attachments/recordedit.js`, and add the function to the file.

The second step is to ensure that the document is loaded, and the `updatecontacts()` function is called to display the current contact list. jQuery makes this easy by providing a `ready()` function on the document. Everything within the function you assign to this operation will be executed when the document has finished loading. You can see the definition in Listing 9.

**Listing 9. ready() function**

```javascript
$(document).ready(function() {
    updatecontacts();
});
```

Push the application again, and reload the index page. There shouldn't be any changes to what is displayed, but it is good practice to ensure that the application has not been broken when adding in new components.

Of course, the list will still be empty, so let's create a form that can be used to view some contacts.

### Creating, editing, and deleting contacts

In this section, we will show you how to create, edit, and delete contacts.

#### Creating new contacts

As a web application, creating a new contact involves providing a form to the user that can be filled in, and then the content of the form can be written to the Apache CouchDB database. The first step for that is a new function that will dynamically generate the HTML for a form, and then attach this to the `contactform` div element defined in `index.html` (see Listing 10).

The function for this is shown in Listing 9.

**Listing 10. Dynamically generating the HTML as a form**

```javascript
function contactform(doctoedit) {
    var formhtml = '<form name="update" id="update" action="">';
    if (docotoedit) {
        formhtml += '<input type="text" name="id" value="' + doctoedit + '" />
    formhtml +='</form>
    return formhtml;
}
```
The last line in Listing 9 is the key. The $( ) construct is shorthand for using jQuery functions, and here we are using it to append a form to an existing element. jQuery makes it easy to access the DOM elements of an HTML page. In this case, using the # prefix looks for an element within the page DOM with the specified id attribute. This follows the same format as used for CSS formatting, which makes it easy to look up different elements. The period prefix looks for items with the specified class. We will see an example of that later.

The function itself accepts a single argument, a document to be edited. We'll use this when we look at editing an existing contact. It's used in the function when building the form first to introduce the document ID for the contact (which will be needed when we update the contact record), and when setting the value of each field in the form.

The basics of the form, however, are straightforward; we generate text input elements with the name and ID of the field (name, phone, email, and so on).

For the form to be activated, you need to enable the Add Contact link in the index.html file so that it calls the function. You can do this by adding the operation to the ready( ) function. This ensures that the button is not active until the document has loaded. See Listing 11 for the jQuery code to update the operation when the link is clicked.

Listing 11. Showing the contact form

```javascript
$("a.add").live('click', function(event) {
  contactform();
});
```
Push the application again to Apache CouchDB. You should now find that the **Add Contact** button populates the form with empty values. You can see a sample of this in *Figure 1.*

**Figure 1. Empty Contact Form**

![Empty Contact Form](image)

The second part of creating a contact is actually saving the document to the database when the **Submit** button is pressed. The function to handle this is shown in *Listing 12.*

**Listing 12. Saving the document to the database when the Submit button is pressed**

```javascript
$("input.update").live('click', function(event) {
    var form = $(event.target).parents("form#update");

    db.saveDoc(builddocfromform(null,form), {
        success: function() {
            $("form#update").remove();
            updatecontacts();
        }
    });

    return false;
});
```

This JavaScript fragment should be added to the existing `ready()` function. The fragment adds the function that will be called when the Update button is clicked. The first line of the inline function creates a variable through which we can access the fields of the form.

The `saveDoc()` function will save a JSON structure as a document to the Apache CouchDB. The first argument should be the document data, and the second a JavaScript object, which defines what happens if saving the document was a success. Remember that JavaScript operations that go out to access information are asynchronous, that is, the request is sent to the host (Apache CouchDB), and you have to wait for the response to come back before operating on the information.

The first argument to this function is the return value of another function, `builddocfromform()`. This is used to simplify the construction of the document from the form data, whether you are creating a new document, or editing an existing one. The code for this is shown in *Listing 13.*
Listing 13. builddocfromform() function

```javascript
function builddocfromform(doc, form) {
  if (!doc) {
    doc = new Object;
  }
  doc.name = form.find("input#name").val();
  doc.phone = form.find("input#phone").val();
  doc.email = form.find("input#email").val();

  return(doc);
}
```

The function accepts an existing document object, initializing it to an empty JavaScript object if document is not defined. Then, it uses the supplied form jQuery object to access each field in the form and populate the document object before returning it. You could add more fields to the function here (providing you also added the field definitions to the form HTML).

The anonymous function attached to the success field will be called if the document was written to the database successfully. If this occurs, the HTML of the contact form is removed by emptying the contactform div element content, and then the updatecontacts() function will be called, which will update the active list of contacts in the display.

You can now push the application to your Apache CouchDB instance again, and try adding a contact to the system. You should end up with one or more contacts in Apache CouchDB, as shown here in Figure 2.

Figure 2. Some contacts

![Contacts](image)

Editing existing contacts

We have already laid much of the groundwork for editing an existing contact. The JavaScript function for outputting the form already accepts an existing document, and the form is populated with the Apache CouchDB document ID and existing values from that object.

The two changes required are first to enable the Edit Contact link output against each contact in the list. Creating this all individually would be a nightmare, but jQuery provides functionality to identify when any link has been clicked by identifying the target DOM object. That information can be used to access the ID of the document, which was embedded into the link, and then to load the record from Apache CouchDB and call the form function. You can see this in Listing 14.
Listing 14. Identifying when a link has been clicked by identifying the target DOM object

```
Listing 14. Identifying when a link has been clicked by identifying the target DOM object

$("#contacts").click(function(event) {
  var target = $(event.target);
  if (target.is('a')) {
    id = target.attr("id");
    if (target.hasClass("edit")) {
      db.openDoc(id, { success: function(doc) {
        contactform(doc);
      } });
    }
  }
});
```

This should be added to the `ready()` function. Following the lines of the code in order:

- First line identifies a click event within the `#contacts` DOM elements.
- Second line identifies the target that was clicked.
- Third line checks that what was clicked as an 'A' clickable element.
- Fourth line identifies the id attribute. In the contacts list, the id attribute of each link contains the document ID of the corresponding contact.
- Fifth line identifies the class of the link that was clicked. The **Edit Contact** links have a class of `edit`, the remove links a class of `remove`. If the link is an 'edit' link, access the Apache CouchDB to load the document (with `openDoc()`), and when the document has successfully been loaded, the `contactform()` function is called with the document data. This will present a contact form with the existing contact information to be edited.

The result is that when you click on the Edit Contact link against a contact, a form with the contact details is displayed.

You might wonder why the contact information that is displayed is not used to populate the form directly. The reason is that as a potentially multi-user application you want to ensure that the document has not been updated (or deleted) by another user before you edit it. Therefore, you want to ensure that the document exists, and you have the latest version as stored in the database.

The other half of the process is to change the function that is called when the **Submit** button on the form is pressed. Here you need to identify whether an existing record is being updated. Since the `contactform()` function only includes the document ID if we are updating an existing document, you can use this to determine the operation type. If we are updating an existing document, then that document should be loaded from the database, and then the form values updated, before the document is saved back to the database. The resulting code is shown in **Listing 15**.

Listing 15. Changing the function that is called when the Submit button on the form is pressed

```
Listing 15. Changing the function that is called when the Submit button on the form is pressed

$(':input.update').live('click', function(event) {
```

Building CouchApps
Again, we load the existing document and update the contents using the `builddocfromform()` function. This ensures that we are updating the latest version of the document. This is important because Apache CouchDB records the revision and changes in all documents. Therefore, you need to ensure that you are updating the latest version — the revision number is used as a check to ensure that you are updating the right version.

There is another reason why we load the document before updating the fields and saving it back. The form, as it stands, only supports name, phone, and email fields of the document. But what if the document contains other fields that this form does not yet support? By loading the entire existing document, and updating only the fields that were on the form, you won't lose any of the fields that the form does not know about.

Of course, there are times when you want to delete a record.

### Deleting existing contacts

Deleting an existing contact should be straightforward. You can add another hook to the Remove Contact link, like the Edit Contact link. However, you don't want the Remove Contact link to be accidentally clicked, so you can provide a confirmation process to ensure that the deletion is required.

You can use some of the principles already demonstrated to output a new set of links, and then use the click events to make these new links either confirm, or cancel, the remove request.

The code should be added after the `edit` function in Listing 14. The code is shown in **Listing 16**.

**Listing 16. Deleting an existing contact**

```javascript
if (target.hasClass("remove")) {
    html += '<span class="confirm">Really Delete? ' +
    ```
When the user clicks the **Remove Contact** link, two further links are added next to the contact. If the **Delete** link is clicked, the document is loaded (to confirm it still exists), and the `removeDoc()` function is called to delete it. If this is successful, you remove the entire contact row, which you identify by looking for the parent DOM element. If the user clicks Cancel, you just remove the confirmation links.

You can see a contact awaiting confirmation for deletion in **Figure 3**.

**Figure 3. Contact awaiting confirmation for deletion**

**Contacts**

![Add Contact](image)

![Edit Contact](image)

![Remove Contact](image)

![Really Delete?](image)

![Delete](image)

![Cancel](image)

The final application

With all of the different elements to the process, it can be difficult to see the entire application. **Listing 17** shows the recordedit.js file, which contains all of the JavaScript for the application.

**Listing 17. recordedit.js file**

```javascript
db = $.couch.db("contacts");
function updatecontacts() {
    $('"#contacts"').empty();
    db.view("contacts/byname", { confidence: function(data) {
        for (i in data.rows) {
            $('"#contacts"').append('<div id="' + data.rows[i].value._id + '" class="contactrow"><span>' + data.rows[i].value.name + '</span><span>' + data.rows[i].value.phone + '</span><span>' + '<a href="#" id="' + id + '" class="actuallydel">Delete</a>' + '<a href="#" id="' + id + '" class="canceldel">Cancel</a></span>').append(html);
        }[target.hasClass("actuallydel")]) {
            db.openDoc(id, {
                success: function(doc) {
                    db.removeDoc(doc, { success: function() {
                        target.parents("div.contactrow").remove();
                    });
                }
            });
        }
    });
}
if (target.hasClass("canceldel")) {
    target.parents("span.confirm").remove();
}
```
"</span><span>" +
  '<a href="#" id="" + data.rows[i].value._id +
 + '<" class="remove">Remove Contact</a>' +
  '</span></div>
});

function contactform(doctoedit) {
  var formhtml;
  formhtml =
  '<form name="update" id="update" action="">';
  if (doctoedit) {
    formhtml = formhtml +
    '<input name="docid" id="docid" type="hidden" value="' + doctoedit._id +
    '"/>';
  }
  formhtml = formhtml +
  '</form>';
  formhtml = formhtml +
  '<table>';
  formhtml = formhtml +
  '<tr><td>Name</td>' +
  '<td><input name="name" type="text" id="name" value="' +
  (doctoedit ? doctoedit.name : '') +
  '"/></td></tr>';
  formhtml = formhtml +
  '<tr><td>Phone</td>' +
  '<td><input name="phone" type="text" id="phone" value="' +
  (doctoedit ? doctoedit.phone : '') +
  '"/></td></tr>';
  formhtml = formhtml +
  '<tr><td>Email</td>' +
  '<td><input name="email" type="text" id="email" value="' +
  (doctoedit ? doctoedit.email : '') +
  '"/></td></tr>';
  formhtml = formhtml +
  '</table>' +
  '<input type="submit" name="submit" class="update" value="' +
  (doctoedit ? 'Update' : 'Add') +
  '"/>' +
  '</form>';
  $('#contactform').empty();
  $('#contactform').append(formhtml);
}

function builddocfromform(doc,form) {
  if (!doc) {
    doc = new Object;
  }
  doc.name = form.find("input#name").val();
  doc.phone = form.find("input#phone").val();
  doc.email = form.find("input#email").val();
  return(doc);
}

$(document).ready(function() {
  updatecontacts();
  $('#contacts').click(function(event) {
    var target = $(event.target);
    if (target.is('a')) {
id = target.attr("id");

if (target.hasClass("edit")) {
    db.openDoc(id, { success: function(doc) {
        contactform(doc);
    }});
}

if (target.hasClass("remove")) {
    html = '<span class="confirm">Really Delete? ' +
        '<a href="#" id="' + id + '" class="actuallydel">Delete</a>' +
        '<a href="#" id="' + id + '" class="canceldel">Cancel</a></span>'; 
    target.parent().append(html);
}

if (target.hasClass("actuallydel")) {
    db.openDoc(id, {
        success: function(doc) {
            db.removeDoc(doc, { success: function() {
                target.parents("div.contactrow").remove();
            }});
        }
    });
}

if (target.hasClass("canceldel")) {
    target.parents("span.confirm").remove();
}

$("a.add").live('click', function(event) {
    contactform();
});

$("input.update").live('click', function(event) {
    var form = $(event.target).parents("form#update");
    var id = form.find("input#docid").val();
    if (id) {
        db.openDoc(id, {
            success: function(doc) {
                db.saveDoc(builddocfromform(doc,form), {
                    success: function() {
                        $("form#update").remove();
                        updatecontacts();
                    }
                });
            },
        });
    } else {
        $db.saveDoc(builddocfromform(null,$form), {
            success: function() {
                $("form#update").remove();
                updatecontacts();
            },
        });
        return false;
    }
});
After you have the file updated, push the application using CouchApp up to your Apache CouchDB instance and try it out.

**One more thing — replicating your CouchApp**

One of the main features of Apache CouchDB is that you can replicate the documents in your database to another database, whether that database exists on the same Apache CouchDB instance or a remote one. The synchronization occurs in both directions, which means that you can replicate your contacts database from your desktop machine to your laptop, make changes on your laptop while away, and then synchronize those changes back to your desktop so that the two databases are kept in synchronization.

As an added bonus, CouchApps, which are just stored in the Apache CouchDB database as documents, are also synchronized. This means that when you replicate the contacts database you are also replicating the application code that makes up the CouchApp application. In environments that are more traditional, this would be difficult to achieve. With CouchApps, that functionality is provided as part of the Apache CouchDB functionality.

You can set up replication by sending the request to the Apache CouchDB server using a command line tool such as curl, but an alternative is to use the Futon tool, a CouchApp built into every Apache CouchDB instance, that provides a complete management and editing interface for Apache CouchDB and the documents stored in the databases within Apache CouchDB.

You can access Futon by visiting http://127.0.0.1:5984/_utils. This shows the Futon interface, as seen here in Figure 4.
Click on the Replicator link on the right hand side, and you will be presented with the form as seen here in Figure 5.

**Figure 5. Replicator form**

Replication can occur either in push (from the current Apache CouchDB instance to a remote database), or pull (from a remote Apache CouchDB instance to the local Apache CouchDB instance). Replication can also either be performed once, or you can set the replication to be continuous, which means that changes on one database will automatically be replicated to the other database if it is available.
For example, if you start a Apache CouchDB instance on your laptop, you can replicate the contacts database from the CouchDB server to the local contacts database. You can see the filled in form, and the result of starting the replication process, in Figure 6.

**Figure 6: Successful replication**

![Successful replication](image)

After the application is on the laptop instance of Apache CouchDB, you can edit and update the database using exactly the same interface because you have replicated the entire application. What's more, you can even replicate the changes back to your desktop when you get home.

**Suggested improvements**

The screenshot samples in this tutorial may look different than your application, because the CSS has been updated to improve the layout slightly. Fortunately, because we have used classes and IDs on all of the different components (including the form, contact list, and links), changing the formatting should be straightforward. The CSS is defined within the contacts/_attachments/style/main.css file on the local file system, and will be included when you push the application with couchapp. Changing the CSS is probably the easiest change you can make to improve your application.

After that, you may want to improve the data that is captured (which can be fixed by updating the form and the structure for storing that data). For example, as outlined in the introduction, you could add support for adding multiple phone numbers by storing the phone numbers within a separate structure in the document.

After you have extended the information displayed, you may want to start improving the contacts list so that you can display it in pages. Paging functionality in Apache CouchDB uses the key returned as part of the view. You may also want to construct different views, and provide search functionality, all of which are possible by building and constructing additional views that build different representations of the contact list.
Summary

CouchApps and Apache CouchDB provide a rich environment for building web applications. The entire process for constructing the forms, saving the data, and reporting on the database content, is entirely stored within the Apache CouchDB database. Using JavaScript, and the jQuery libraries simplifies a lot of the complexity of constructing the application. Meanwhile, Apache CouchDB eliminates the need to worry about defining tables or writing complex queries to get at the information that you have stored. Finally, with Apache CouchDB you can easily replicate your entire application, including all of its data, to another Apache CouchDB instance, including your laptop or mobile phone.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutorial source code</td>
<td>contacts_tar_bz2.zip</td>
<td>59KB</td>
</tr>
</tbody>
</table>
Related topics

• **JavaScript basics and DOM Manipulation**: View Siarhei Barysiuk's slides for a briefing on the basics and more.
• **jQuery documentation**: Find thorough documentation and assistance.
• **JavaScript and the Document Object Model** (Nicholas Chase, developerWorks, July 2002): Read more about the JavaScript approach to DOM and building a web page to which the user can add notes and edit note content.
• **JavaScript Tutorial**: Learn how to use the scripting language of the web.
• **jQuery**: Download the jQuery JavaScript library, available under either the MIT or GNU Public License. You can either download it and serve it locally, or include and serve it directly from the Google CDN.
• **jQuery Browser Compatibility page**: Review a list of recommended browsers that jQuery actively supports.
• **Apache CouchDB**: Download the latest version from CouchDB.
• **The CouchApp tool**: Go to GitHub website to download this tool, and find utilities to make standalone CouchDB application development simple.
• **developerWorks on Twitter**: Follow us for the latest news.