CouchDB
The Definitive Guide
(../index.html)

SEARCH

This edition of the book is a work in progress.
Please create **a pull request (https://github.com/oreilly/couchdb-guide/pulls)** or **report an issue (http://github.com/oreilly/couchdb-guide/issues)** for any corrections or suggestions you may have.

Validation Functions

(#validation)
In this chapter, we look closely at the individual components of Sofa's validation function. Sofa has the basic set of validation features you'll want in your apps, so understanding its validation function will give you a good foundation for others you may write in the future.

CouchDB uses the `validate_doc_update` function to prevent invalid or unauthorized document updates from proceeding. We use it in the example application to ensure that blog posts can be authored only by logged-in users. CouchDB's validation functions —like map and reduce functions—can't have any side effects; they run in isolation of a request. They have the opportunity to block not only end-user document saves, but also replicated documents from other CouchDBs.

Document Validation Functions

(#functions)
To ensure that users may save only documents that provide these fields, we can validate their input by adding another member to the `_design/` document: the `validate_doc_update` function. This is the first time you've seen CouchDB's external process in action. CouchDB sends functions and documents to a JavaScript interpreter. This mechanism is what allows us to write our document validation functions in JavaScript. The `validate_doc_update` function gets executed for each document you want to create or update. If the validation function raises an exception, the update is denied; when it doesn't, the updates are accepted.

Home (../index.html)

Draft edition (index.html)

Previous Page (views.html)

Next Page (show.html)

Home

Document validation is optional. If you don't create a validation function, no checking is done and documents with any content or structure can be written into your CouchDB database. If you have multiple design documents, each with a `validate_doc_update` function, all of those functions are called upon each incoming write request. Only if all of them pass does the write succeed. The order of the validation execution is not defined. Each validation function must act on its own. See **Figure 1, "The JavaScript document validation function" (#figure/1)**.

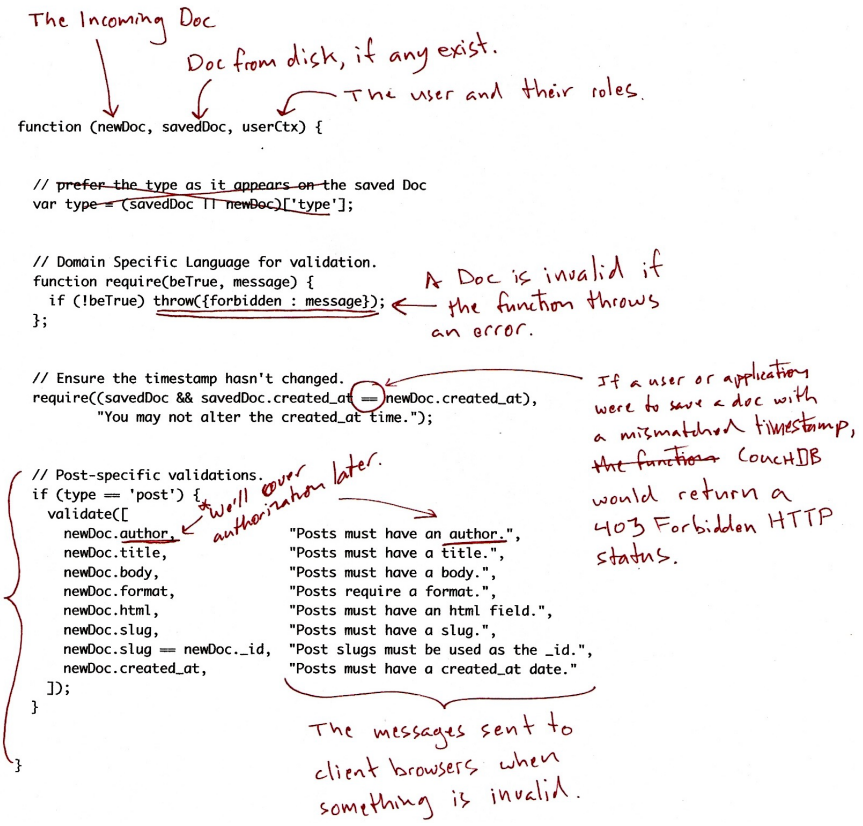

Figure 1. The JavaScript document validation function

(#figure/1)

Validation functions can cancel document updates by throwing errors. To throw an error in such a way that the user will be asked to authenticate, before retrying the request, use JavaScript code like:

```
throw({unauthorized : message});
```

When you're trying to prevent an authorized user from saving invalid data, use this:

```
throw({forbidden : message});
```

This function throws `forbidden` errors when a post does not contain

the necessary fields. In places it uses a `validate()` helper to clean up the JavaScript. We also use simple JavaScript conditionals to ensure that the `doc._id` is set to be the same as `doc.slug` for the sake of pretty URLs.

If no exceptions are thrown, CouchDB expects the incoming document to be valid and will write it to the database. By using JavaScript to validate JSON documents, we can deal with any structure a document might have. Given that you can just make up document structure as you go, being able to validate what you come up with is pretty flexible and powerful. Validation can also be a valuable form of documentation.

Validation's Context

(#context)

Before we delve into the details of our validation function, let's talk about the context in which they run and the effects they can have.

Validation functions are stored in *design documents* under the `validate_doc_update` field. There is only one per design document, but there can be many design documents in a database. In order for a document to be saved, it must pass validations on all design documents in the database (the order in which multiple validations are executed is left undefined). In this chapter, we'll assume you are working in a database with only one validation function.

Writing One

(#writing)

The function declaration is simple. It takes three arguments: the proposed document update, the current version of the document on disk, and an object corresponding to the user initiating the request.

```
function(newDoc, oldDoc, userCtx) {}
```

Above is the simplest possible validation function, which, when deployed, would allow all updates regardless of content or user roles. The converse, which never lets anyone do anything, looks like this:

```
function(newDoc, oldDoc, userCtx) {
  throw({forbidden : 'no way'});
}
```

Note that if you install this function in your database, you won't be able to perform any other document operations until you remove it from the design document or delete the design document. Admins can create and delete design documents despite the existence of this extreme validation function.

We can see from these examples that the return value of the function is ignored. Validation functions prevent document updates by raising errors. When the validation function passes without raising errors, the update is allowed to proceed.

Type

(#type)

The most basic use of validation functions is to ensure that documents are properly formed to fit your application's expectations. Without validation, you need to check for the existence of all fields on a document that your MapReduce or user-interface code needs to function. With validation, you know that any saved documents meet whatever criteria you require.

A common pattern in most languages, frameworks, and databases is using types to distinguish between subsets of your data. For instance, in Sofa we have a few document types, most prominently `post` and `comment`.

CouchDB itself has no notion of types, but they are a convenient shorthand for use in your application code, including MapReduce views, display logic, and user interface code. The convention is to use a field called `type` to store document types, but many frameworks use other fields, as CouchDB itself doesn't care which field you use. (For instance, the CouchRest Ruby client uses `couchrest-type`).

Here's an example validation function that runs only on posts:

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.type == "post") {
    // validation logic goes here
  }
}
```

Since CouchDB stores only one validation function per design document, you'll end up validating multiple types in one function, so the overall structure becomes something like:

```
function(newDoc, oldDoc, userCtx) {
```

```
  if (newDoc.type == "post") {
    // validation logic for posts
  }
  if (newDoc.type == "comment") {
    // validation logic for comments
  }
  if (newDoc.type == "unicorn") {
    // validation logic for unicorns
  }
}
```

It bears repeating that `type` is a completely optional field. We present it here as a helpful technique for managing validations in CouchDB, but there are other ways to write validation functions. Here's an example that uses *duck typing* instead of an explicit `type` attribute:

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.title && newDoc.body) {
    // validate that the document has an author
  }
}
```

This validation function ignores the `type` attribute altogether and instead makes the somewhat simpler requirement that any document with both a title and a body must have an author. For some applications, typeless validations are simpler. For others, it can be a pain to keep track of which sets of fields are dependent on one another.

In practice, many applications end up using a mix of typed and untyped validations. For instance, Sofa uses document types to track which fields are required on a given document, but it also uses duck typing to validate the structure of particular named fields. We don't care what sort of document we're validating. If the document has a `created_at` field, we ensure that the field is a properly formed timestamp. Similarly, when we validate the author of a document, we don't care what type of document it is; we just ensure that the author matches the user who saved the document.

Required Fields

(#required)

The most fundamental validation is ensuring that particular fields are available on a document. The proper use of required fields can make writing MapReduce views much simpler, as you don't have to test for all the properties before using them—you know all

documents will be well-formed.

Required fields also make display logic much simpler. Nothing says *amateur* like the word `undefined` showing up throughout your application. If you know for certain that all documents will have a field, you can avoid lengthy conditional statements to render the display differently depending on document structure.

Sofa requires a different set of fields on posts and comments. Here's a subset of the Sofa validation function:

```
function(newDoc, oldDoc, userCtx) {
  function require(field, message) {
    message = message || "Document must have a " +
field;
    if (!newDoc[field]) throw({forbidden : message});
  };

  if (newDoc.type == "post") {
    require("title");
    require("created_at");
    require("body");
    require("author");
  }
  if (newDoc.type == "comment") {
    require("name");
    require("created_at");
    require("comment", "You may not leave an empty
comment");
  }
}
```

This is our first look at actual validation logic. You can see that the actual error throwing code has been wrapped in a helper function. Helpers like the `require` function just shown go a long way toward making your code clean and readable. The `require` function is simple. It takes a field name and an optional message, and it ensures that the field is not empty or blank.

Once we've declared our helper function, we can simply use it in a type-specific way. Posts require a `title`, a `timestamp`, a `body`, and an `author`. Comments require a `name`, a `timestamp`, and the `comment` itself. If we wanted to require that every single document contained a `created_at` field, we could move that declaration outside of any type conditional logic.

Timestamps

(#timestamps)

Timestamps are an interesting problem in validation functions. Because validation functions are run at replication time as well as during normal client access, we can't require that timestamps be set close to the server's system time. We can require two things: that timestamps do not change after they are initially set, and that they are well formed. What it means to be well formed depends on your application. We'll look at Sofa's particular requirements here, as well as digress a bit about other options for timestamp formats.

First, let's look at a validation helper that does not allow fields, once set, to be changed on subsequent updates:

```
function(newDoc, oldDoc, userCtx) {
  function unchanged(field) {
    if (oldDoc && toJSON(oldDoc[field]) !=
toJSON(newDoc[field]))
      throw({forbidden : "Field can't be changed: " +
field});
  }
  unchanged("created_at");
}
```

The `unchanged` helper is a little more complex than the `require` helper, but not much. The first line of the function prevents it from running on initial updates. The `unchanged` helper doesn't care at all what goes into a field the first time it is saved. However, if there exists an already-saved version of the document, the `unchanged` helper requires that whatever fields it is used on are the same between the new and the old version of the document.

JavaScript's equality test is not well suited to working with deeply nested objects. We use CouchDB's JavaScript runtime's built-in `toJSON` function in our equality test, which is better than testing for raw equality. Here's why:

```
js> [] == []
false
```

JavaScript considers these arrays to be different because it doesn't look at the contents of the array when making the decision. Since they are distinct objects, JavaScript must consider them not equal. We use the `toJSON` function to convert objects to a string representation, which makes comparisons more likely to succeed in the case where two objects have the same contents. This is not guaranteed to work for deeply nested objects, as `toJSON` may serialize objects in an undefined order.

The `js` command gets installed when you install CouchDB's SpiderMonkey dependency. It is a command-line application that lets you parse, evaluate, and run JavaScript code. `js` lets you quickly test JavaScript code snippets like the one previously shown. You can also run a syntax check of your JavaScript code using `js file.js`. In case CouchDB's error messages are not helpful, you can resort to testing your code standalone and get a useful error report.

Authorship

(#authorship)

Authorship is an interesting question in distributed systems. In some environments, you can trust the server to ascribe authorship to a document. Currently, CouchDB has a simple built-in validation system that manages *node admins*. There are plans to add a database admin role, as well as other roles. The authentication system is pluggable, so you can integrate with existing services to authenticate users to CouchDB using an HTTP layer, using LDAP integration, or through other means.

Sofa uses the built-in node admin account system and so is best suited for single or small groups of authors. Extending Sofa to store author credentials in CouchDB itself is an exercise left to the reader.

Sofa's validation logic says that documents saved with an author field must be saved by the author listed on that field:

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.author) {
    enforce(newDoc.author == userCtx.name,
      "You may only update documents with author " +
userCtx.name);
  }
}
```

Wrapping Up

(#wrap)

Validation functions are a powerful tool to ensure that only documents you expect end up in your databases. You can test writes to your database by content, by structure, and by user who is making the document request. Together, these three angles let you build sophisticated validation routines that will stop anyone from tampering with your database.

Of course, validation functions are no substitute for a full security system, although they go a long way and work well with CouchDB's other security mechanisms. Read more about CouchDB's security in **Chapter 22, Security (security.html)**.