

Rolling Your Own Authentication for REST API's in XPages

a modest proposal by Tim Tripcony

A key ingredient of any successful application is a secure and reliable authentication mechanism. As more and more applications emerge that store their data "in the cloud", relying upon third-party applications to provide an interactive interface for that data, the ability to seamlessly integrate authentication into a universal API is becoming increasingly vital to any application's adoption rate. This article will familiarize you with one possible approach to adding this style of authentication to your own IBM Lotus Domino applications using XPages.

Table of Contents

Putting your data to REST.....	2
Making a Token gesture.....	5
Anonymity is a warm blanket.....	7
Welcome to the club.....	9
The Keymaster and the Gatekeeper.....	12
The sum of the parts.....	15
Epilogue.....	19

Putting your data to REST

The most common architecture used today to provide an API to cloud-based applications is known as REST. This term is shorthand for "[REpresentational State Transfer](#)". This is simply a fancy way of stating that this architecture allows an application to define how it can be instructed by a consumer of its data to manipulate the state of specific portions of that data. REST is a natural choice for such data interfaces, because it is already fundamental to how the entire Internet currently functions.

When you access nearly any website, your browser is sending what is known as a "GET" request to the server that hosts the site, which tells the server what information you are requesting. The server attempts to associate that request with resources that it can deliver. If the attempt is successful it sends back a representation of those resources; otherwise it sends back information about the error that was encountered. Similarly, when you submit data to a website – for example, searching for specific content – your browser sends a "POST" request, which transmits the information you are submitting to the server, which in turn attempts to send back information pertaining to the result of the submission. We are all familiar by now with this process, of course, even if we are unaware of the specifics of the protocol.

In addition to GET and POST requests, browsers and servers typically understand several other methods: these include (among others) PUT and DELETE. While most web applications now use GET and POST requests to allow the browser to instruct the server to replace or remove existing resources, the PUT and DELETE requests are the remaining two request methods that make up the REST standard. To sum up, the following is a brief recap of how the REST architecture defines each request method:

GET: retrieve the current state of the designated resource

POST: create a new resource

PUT: replace the resource with a new resource, or an updated version of the existing resource

DELETE: remove the resource

Blogging applications were among the first types of Internet applications to implement the REST architecture. If we consider an individual blog article to be a specific resource within a larger application, it becomes easy to visualize how this architecture functions in a "real world" scenario:

GET: retrieve information about an article, such as its title, author, "permalink", content, categorization, and the date it was posted

POST: add a new article to the blog

PUT: update an article to correct misspellings or grammatical mistakes, or to provide an update on the topic or events originally described in the article

DELETE: remove an article from the blog

The concept of a "permalink" was initially key to the entire REST architecture, as it allowed a URI (uniform resource identifier) to uniquely distinguish one resource within an application from all others. As convention dictates that the address of a blog article should never change, it was natural that each article's permalink – its permanent address – be used as its resource identifier within the application.

Hence, a typical early REST API was quite straightforward: if the application server received a GET request at the address matching an existing permalink, it would send back information about the corresponding article; a PUT or DELETE request to the same address would, respectively, update or remove the article. The API would also define a base address, where a POST request would create a new article and a GET would retrieve a list of all resources, with a URI for each – to which additional requests could be sent to retrieve or manipulate a specific resource.

As more and more applications have embraced the REST approach, many have adopted a slight variation upon the original design to account for certain complexities that may not fit the model previously described. Rather than accept requests at a separate address for each resource, each address identifies an action to be performed, and the application gleans from any request additional information (such as request headers or query string parameters) to identify what resources are created or impacted by the request and, in some cases, to further refine what operation is to be performed against those resources.

Still other applications define a single address for the entire API, relying entirely on request headers and parameters to determine what data is involved and what is being done with that data. Many do not even make use of PUT or DELETE requests, allowing all supported operations to be performed via GET or POST, as nearly all websites now do. While some argue that an API must treat these four request methods in precisely the way previously described to be a "true" REST API, there is no "right" way to do this; no official standard exists. For the present, however, the underlying principles of the REST architecture form the dominant model currently being used to define the API for widely adopted Internet applications.

So why is REST important? In short, because it allows your application to take on a life of its own. Twitter is perhaps the best example of this phenomenon in active use today. It does almost nothing. It's essentially just "reverse SMS": instead of sending a short message to one person, you're sending it to whomever happens to be listening, and it allows you to choose who to listen to. That's it. That's all it does. Or, at least, that's all it did when it first arrived. Replies, "retweets", and all other current uses of the service emerged only as users started turning the service into what they wanted it to be. While it is possible that this would still have happened organically if Twitter only had one official web interface, the viral adoption that allowed such patterns to develop into behavioral memes started only after a myriad of third-party applications began to consume the application's API. Conversely, it is quite possible that if Twitter were *only* an API, and had no official front-end of its own, it might still have grown to the levels of usage it currently enjoys simply because enough third-party applications sprang up to appeal to every conceivable category of end user.

By offering such an API – which allows your application to be defined not by a visual interface you designed, but by the abstract concepts that its data represents – you leave the potential that each service you provide represents open to interpretation by an unlimited number of external entities. Having been given access to a specific set of functions that your service supports, a developer with whom you've perhaps had no direct interaction may identify a use for your service that you never originally intended. The result of this, of course, may not always be a net positive. If, however, the concepts your API unleashes to third-party interpretation allow people to do something they value in ways that no other service can match, you may soon witness other developers making your application ubiquitous for you in ways that literally change the world.

Before we move on, there is one crucial facet of this architecture that must be addressed: as you may have noticed, we have not yet discussed the format of the data being transported between a consumer of a REST API and the server hosting that API. This is because the REST architecture does not define a data format – only the protocol used to transport the data: namely, HTTP requests. Originally, every implementation of REST used XML. Today, JSON has become as widely used as – if not more than – XML, primarily because it is, quite frankly, simpler. Not only can languages such as JavaScript and PHP parse JSON with a single native function call into an object structure that is easier to interact with than traditional DOM/SAX parsers, but JSON simply allows the same information to be conveyed in less bytes. In order to cater to developers who prefer either format, however, many API services choose to support both formats, typically allowing developers to specify in request headers or parameters which format should be used. What is important to remember, however, is that, as the API provider, the data format is entirely up to you: you can use CSV, VCF, even a format of your own definition, if you prefer... again, REST defines the way that the data is sent, not what that data looks like. Our code examples will use JSON, but you should use whatever format(s) you feel will best accomplish your goals, such as ease of consumption or application response time.

Making a Token gesture

A significant factor in the wide adoption of the REST architecture is that it liberates Internet data from the browser: any application or device with the capacity to issue HTTP requests can consume a REST API, allowing its own users to interact seamlessly with the application providing that API. One obstacle, however, to implementation of any REST API provider is... that it liberates data from the browser. Why is this potentially problematic? Because an API provider cannot take for granted all of the security mechanisms that are now bundled with nearly every browser.

Arguably the biggest limitation is the possibility that the API consumer is unable to store session cookies – or, at the very least, will not respond to them in the same fashion. Domino, like many other web servers, relies upon session cookies to allow what is known as a "token" to be sent to the server in the headers of every request; this temporary token, issued to the user upon initial authentication, is encrypted data that allows Domino to securely identify the user without requiring their user name and password to be sent in plain text every time the user navigates to another page or submits information to the server. Because existing session cookies are automatically sent to the server with every request without requiring the design of the web page to specifically instruct the browser to do so, once a user is logged in, we know that their token will be sent every time and do not have to write any code to enforce this behavior. Without certainty that the API consumer supports cookies, however, we need another way to secure the application. The remainder of this article will describe one possible approach.

Before we proceed further, let's get something out of the way: several of the techniques we're about to explore take advantage of changes to the Domino platform planned for version 8.5.2, which at the time of this writing is scheduled to be released later this year. As IBM always reserves the right to remove pending features from the final release, some of what is discussed in this article may no longer be applicable once 8.5.2 is officially available. Assuming each of these features does make it into the product, however, their use in combination provides an easy and effective way of implementing reliable security for the type of API previously described.

Domino's built-in session authentication mechanism uses multiple layers of encryption to associate a temporary token with a single user. To be precise, an encryption key stored within an Internet Site configuration document allows the Domino server to decrypt the token being sent by the browser; once decrypted, the token contains both the name of the authenticated user and the expiration date-time of the token itself. Hence, Domino not only knows who the user is, but also whether the token is still valid. If the token has expired, was not valid to begin with, or identifies a user who has insufficient access to perform the operation associated with the request that included the token, Domino asks the user to attempt to authenticate again. Furthermore, the configuration document that stores the encryption key is itself encrypted; the server must be able to decrypt the document to even read the key that is then used to decrypt and create tokens. This mechanism is one of many reasons why Domino is more secure than most other web server platforms.

This article proposes a technique that does not involve true encryption; instead, it uses two different types of one-way hash encoding known, respectively, as MD2 and MD5. When encrypting data, a key is used to transform the original data into its encrypted representation. In order to decrypt the result, a key must be used to transform the encrypted data back into its original value. Hash encoding, on the

other hand, is designed to transform plain text data into an encoded representation that *cannot* be decoded. Instead, any implementation of a given hashing algorithm is designed such that encoding the same value will always return the same encoded result.

For example, encoding the word "Domino" using one implementation of the MD2 hash algorithm that is built into Domino will always return a value of "AEB5929ED34F602FD98DB7917098AC5B". Looking at that value, you're probably thinking what most Lotus professionals might: "That looks like a UNID." You're absolutely right: the result of MD2-encoding any string value – regardless of the length of that string value – is an uppercase 32-character hexadecimal string, which is the same format used for the UNID of any note in a Domino database. Those of you who have worked with older versions of Notes and Domino might recognize that this format was also previously used for something else: storage of a user's HTTP password... although when this format was used for that purpose, the hexadecimal value was surrounded by parentheses. In fact, even in the current version of Notes and Domino, the Formula function @Password returns an MD2 encoding of the value passed to the function, surrounded by parentheses... this is the built-in implementation referred to moments ago.

This result carries with it a fascinating implication: performing an MD2 hash always results in a value that is valid for use as a UNID. As you may already know, the UNID is a read-write property for a database note. This provides us the luxury of setting "meaningful" identifiers for our documents, as long as our applications are designed such that we have an opportunity to set the UNID before the document is originally saved (changing a document's UNID after it has already been saved to disk causes a new copy of the document to be created). This technique offers significant performance benefits compared to traditional index-based data structures: we need not search for data within a larger collection; if we know what information uniquely identifies that document, we "know" its UNID simply by hashing that value, and can therefore go directly to it. Accessing a document via its UNID is always faster than retrieving it via a view or even searching for it within a full-text index. Basing an entire application's data structure on this technique also relieves the server of the burden of maintaining unnecessary indexes. As you'll soon see, however, this technique can also provide us fine-grained control over the security of our application's data.

At first glance, the latter use may seem like a contradiction. After all, if hash encoding always produces a predictable result, won't the result be insecure? In reality, because hash encoding is one-way, the only way to "decode" the result is to perform the same hash operation against a series of candidate values until a match is found. In other words, the less likely the original value is to be guessed by whoever or whatever is attempting to decode it, the longer it will take for a match to be found because more obvious values will be attempted first. While this is not an absolute guarantee against brute force intrusion attempts by unsavory types, the way in which we'll be using it later will be perfectly sufficient for securing most data.

Anonymity is a warm blanket

Another aspect of this approach to authentication that may at first seem counter-intuitive is that, even if a valid token is passed with an API request, the user will not, in fact, be authenticated... that is, as far as Domino can determine. All data submitted and requested by consumer applications will be handled anonymously. You might well ask, "then what's the point?" Firstly, as was mentioned previously, we don't know that our consumers will handle cookies the same way that browsers do, and are therefore forced to circumvent Domino's standard authentication mechanism anyway.

Secondly, this allows us to discard our existing definition of a "user"; for most situations where we would use this technique, of course, one "user" will equal one human, but this alternate approach to authentication frees us up to muddy those waters, if desired, without risk of doing the same to our Domino Directory – or even an alternate directory defined via Directory Assistance. This becomes especially important if our application experiences viral adoption, because the performance degradation that would be experienced using traditional authentication against a Domino Directory containing millions of registered users would render the service unusable. The technique this article proposes, however, offers practically linear scalability when supporting an enormous user base.

Finally, ensuring that all data is accessed and submitted anonymously allows us to lock down our application completely. This is, of course, the most counter-intuitive statement of all. But this brings us to the first change planned for 8.5.2 that will be crucial to this technique: the ability to flag an XPage as available to "public access" users.

Domino has long supported a seemingly contradictory set of permissions in access control lists: an ACL entry can be listed as having no access, yet can also be granted the ability to read and/or write public documents (which includes design elements flagged for public access). This allows us to create a "welcome page" for users who do not yet have access to the application; this page typically describes the nature of the application and provides either a direct method for obtaining access to it or a description of that process. Even contact information for someone who controls access to the application is more useful than what the user experienced prior to the addition of public access ACL flags: being told simply to go away. Unfortunately, prior to 8.5.2, XPages could not be flagged as public access design elements, so an application relying primarily on XPage elements for its interface was forced either to use other design elements to provide such a home page or revert to the "just go away" approach to greeting users who did not already have access.

Because in 8.5.2 XPages can be flagged as public access elements, we can not only use them as welcome pages for authenticated and anonymous users alike, we even have the option of completely reversing our traditional application access model: we can, in fact, prevent *all* users from accessing the application and *only* permit access by anonymous users... but only to XPages flagged for public access. How does this make our application more secure? It forces all interactions with application data to go through the API that we provide, preventing all methods of direct access to data. Nobody but the server and the application signer has access to the underlying data, so no data is accepted or sent unless our code determines that the request to do so is valid.

This, in turn, brings us to the second change in 8.5.2 that is crucial to this technique: the new

`sessionAsSigner` global variable in SSJS (server-side JavaScript). In an XPage, all code runs under the permissions of the user accessing the page. This is convenient but limiting: unlike agents, which by default run as the signer but can be flagged to run on behalf of the user, XPage code does not even have the option of performing operations with the signer's permissions. In 8.5.2, however, a new global variable, `sessionAsSigner`, has been added. This variable is identical to the existing `session` variable, except that it provides an entry point to the Domino API that brings with it the current permissions of the ID last used to sign the XPage. As a result, any operations performed on any Domino objects accessed via the `session` variable still run as the user; any operations performed on any Domino objects accessed via `sessionAsSigner`, however, run as the signer. With this change, XPages have *less* limitations than agents, because user and signer permissions can be fluidly mixed, not only within the same design element, but within the same code event – indeed, if desired, even within the same line of code. This can, of course, cause disastrous problems if used unwisely, but also offers great flexibility when used appropriately.

For our purposes, this new capability allows us to rely upon the anonymous user's permissions only to tell us which database we are currently in, and immediately pass control to the signer to determine whether the API consumer will be permitted to perform the operation described in the API request. Again we are confronted with an apparent contradiction: as an anonymous user with no access to the current application, why would we rely on the user's permissions to identify the current database and not the signer's? As strange as it might seem, the `sessionAsSigner` instance of the Java `Session` class has no current database. While it might not be an entirely accurate depiction of what is going on behind the scenes of the Domino Java API, we can consider this variable to be an instantiation of a "new" session with no corresponding database. Our anonymous user, however, does have public access to the current database, which is sufficient to tell us its location – specifically, the server and filepath used to access it. As a result, the following SSJS code gives us a valid entry point to the current database that will access it using the signer's permissions:

```
var signerDatabase:NotesDatabase =  
sessionAsSigner.getDatabase(database.getServer(), database.getFilePath());
```

Once we have established that database handle, any methods performed against it that attempt to access or create documents within it will do so on behalf of the signer. The same exact methods performed against the global `database` variable, however, would continue to run under the user's permissions – which, in our case, would be denied access to any data.

We now have a way to ensure that anonymous users can interact with our application's data, but only to the extent that our code determines they should be permitted... so let's dive into how our code will make that determination.

Welcome to the club

Before we can allow users to log in, we must first, of course, have users. Since we're using a custom authentication scheme, we obviously need a custom registration mechanism as well.

For the purposes of this authentication approach, a user consists only of a document within our application that stores a single item value: a hashed representation of the user's password. It may seem strange, at first glance, that we're not storing the user's ID in the document that stores their password... there's no need, as you'll soon see.

Earlier we discussed an option built into Domino for obtaining an MD2 hash of any string value. Because this implementation is specific to Domino, it is actually a variation on the standard algorithm, and will not return the same encoded result that would be returned by other implementations. Furthermore, the token format used in the authentication technique this article proposes relies upon the use of MD5 hashes; not only is there no implementation of this algorithm built directly into Domino, but we need to be absolutely sure that the token value we receive from third-party applications is the precise value we have calculated in advance that we are expecting to receive. We need, therefore, a more generic method for hashing string values.

One of the key benefits of XPages is that it is now easier than ever to incorporate external code into our applications: specifically, Java code written by developers with no knowledge of Domino can be imported directly into, and leveraged within, Domino applications – nearly always without any additional modification needed. In many cases, features built into the base Java language can be very useful in our applications. As an initial example, let's take a look at a small custom utility class, called HashMaster:

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class HashMaster {

    public static String md2(String unhashed) {
        return encode(unhashed, "MD2");
    }

    public static String md5(String unhashed) {
        return encode(unhashed, "MD5");
    }

    private static String encode(String unhashed, String algorithmId) {
        String hashed = "";
        byte[] defaultBytes = unhashed.getBytes();
        try {
            MessageDigest algorithm = MessageDigest.getInstance(algorithmId);
            algorithm.reset();
            algorithm.update(defaultBytes);
            byte messageDigest[] = algorithm.digest();
            hashed = new BigInteger(1, messageDigest).toString(16);
        }
    }
}
```

```

    } catch (NoSuchAlgorithmException nsae) {
        nsae.printStackTrace();
    }
    return hashed;
}
}

```

This class not only allows us to generate an MD2 hash for reasons already mentioned, but the same class can also obtain an MD5 hash, which will be useful later, both in generating and in validating authentication tokens. In the meantime, let's take a look at a method from another class (`DominoUtil`), which will be used extensively in our API:

```

public Document getDocumentByPrimaryKey(Database source, String key,
    Boolean createOnFail) {
    Document result = null;
    String unid = "";
    try {
        unid = HashMaster.md2(key);
        result = source.getDocumentByUNID(unid);
    } catch (NotesException ne) {
        if (ne.id == 4091 && createOnFail) {
            try {
                result = source.createDocument();
                result.setUniversalID(unid);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result;
}

```

This method does the following:

1. Obtains an MD2 hash of the "primary key" passed to the method
2. Since the result of the hash is a syntactically valid UNID, it attempts to locate a document within the specified database that has the same UNID
3. Although the hash result might be the UNID of an existing document but also might not be, there is the possibility that an exception will be thrown. In that scenario we may optionally create a new document and set its UNID to match the hash. Take note that, if a new document is created, it is not yet saved. In either scenario, we end up with a document whose UNID is an encoded version of the text used to request it... in other words, the UNID is meaningful: if I pass "Tim Tripcony" as the key, the resulting UNID is an encoded representation of my name.

Finally, let's examine how this concept is put to good use within the `register()` method of our `UserRegistration` class:

```

public boolean register() {
    try {
        DominoUtil util = new DominoUtil(getDatabase());
        Document userRecord = util.getDocumentByPrimaryKey(getUserId(),

```

```

        true);
    if (userRecord.isNewNote()) {
        userRecord.replaceItemValue("userPass",
HashMaster.md5(getPassword()));
        userRecord.replaceItemValue("Form", "user");
        setSuccess(userRecord.save());
    } else {
        setError("User Id " + getUserId() + " already exists");
    }
} catch (NotesException e) {
    setError("An unexpected error occurred while attempting to register
user");
    e.printStackTrace();
}
return isSuccess();
}

```

As you can see, this method attempts to locate a document that has a UNID matching the MD2 hash of the user ID we are attempting to register. If the document it returns is not a new note, then the user has already been registered: the database already contains a document with a UNID representative of the specified user ID. If, however, the document is a new note, then we can safely create the new account.

As was previously mentioned, we're not storing the user ID in their account record – just their password. For the sake of convenience, we are writing a `Form` item, in case we want to create a view to display all user accounts, though it's not likely we ever would. Similarly, you may choose in your own application to store additional information about each user to allow users to search for other users and view information about them, such as a profile picture or contact information. For the purposes of authentication, however, we never need to search any index for a user's account record, because we can always obtain a hash of their ID and navigate straight to the account record via the resulting UNID.

One concept that should be noted is the format used for the user ID. This could be a canonical name (i.e. "CN=John Doe/OU=..."), a shortname (i.e. "jdoe"), an email address... when using this style of authentication, the name format is inconsequential as long as it's consistent. The key limitation is that – unlike traditional Domino authentication – this particular technique would not allow the same user to enter their canonical name, shortname or email address and still be authenticated as the same user... you have to choose one.

An additional implication of using a hash of the user ID as the UNID for the account record is that this makes the user ID case sensitive, which is atypical; most authentication systems expect the user's password to be entered using the same case as the stored value, but do not require the case to match for the user ID. Because an MD2 hash of a lowercase string will not return the same value as an MD2 hash of an upper- or mixed-case version of the same string, however, our implementation requires that the case of the ID match each time it is evaluated – at least, in the low-level code. You can certainly shield your users from this requirement by converting the case of the ID to ensure it is always the same prior to being hashed. If, however, you *prefer* that the ID be case-sensitive, hashing the ID as passed without case modification will automatically enforce that requirement for you.

The Keymaster and the Gatekeeper

Now that our application has users, we need to allow them to log in. As we discussed earlier, this process essentially just consists of two steps: generating temporary tokens, and matching tokens that come back in subsequent API requests to the corresponding user. The latter step also includes determining whether the token is even valid to begin with and, if so, whether it has expired. We'll now take a look at exactly how to do this.

Just as the data transport format you choose for your API is a decision you must make based on priorities you set for your application, you can choose any format you wish for constructing tokens. In order to provide one example of this technique, we will be modeling our authentication scheme based on some principles used by [Toodledo](#), a cloud-based task management service. To be precise, we will construct a temporary token upon request to send to the consuming application. Instead of simply sending that token back, however, the consumer will be required to send back an API "key" based on that token accompanying any subsequent API request.

The format that Toodledo uses for an API key is as follows:

```
md5 ( md5 ( password ) + token + userId )
```

Let's express this format briefly in paragraph form. First, the user's password is hashed using the MD5 algorithm. Next we append the temporary token and the user ID to that result. Finally, we MD5 hash the entire combination one more time. The result is a 32-character mixed case key that is a representation not only of the user's credentials, but also of the period of time during which the key is valid.

In order to illustrate how we can generate the temporary token that will be used to construct the API key, let's take a look at the constructor of a `TokenRequest` class:

```
public TokenRequest(String userId, Database database) {
    setUserId(userId);
    setDatabase(database);
    DominoUtil util = new DominoUtil(database);
    Document userRecord = util.getDocumentByPrimaryKey(userId, false);
    try {
        if (!(userRecord == null)) {
            DateTime loginTime = util.getDateTime();
            Document tokenRecord = database.createDocument();
            setToken(tokenRecord.getUniversalID());
            tokenRecord.replaceItemValue("Form", "token");
            tokenRecord.replaceItemValue("userId", userId);
            tokenRecord.replaceItemValue("tokenCreated", loginTime);
            String key = getAuthKey(userId, userRecord
                .getItemValueString("userPass"), getToken());
            tokenRecord.setUniversalID(HashMaster.md2(key));
            tokenRecord.save();
        } else {
            setError("User Id " + userId + " does not exist");
        }
    }
}
```

```

    } catch (NotesException e) {
        setError("An unexpected error occurred while attempting to request a
token");
        e.printStackTrace();
    }
}

```

This code attempts to locate the user's account record. Remember, we don't have to search a view to locate this document: because the UNID of an account record is predictable, we know how to locate it even though the document that we'll retrieve doesn't actually store the user ID. Hence, if the hashed user ID does not match the UNID of an existing document, the specified user does not exist. Otherwise, the handle we now have on the account record was retrieved rapidly, no matter how much data our application currently contains. This performance implication is fundamental to why this technique is so useful when designing an application to be as scalable as possible.

If the specified user ID is valid, we create a document that represents the temporary token that is being requested. As before, we set some convenience field values – this time, both the form associated with the document and the creation date of the token. Similar to our user registration process, however, we have no need to ever store the token. As you might imagine, we will be overwriting the UNID of the token document with a value of our own choosing, but until we do, it has a default UNID automatically generated by Domino; for the sake of convenience, this will be the token that we send back to the consumer. We could instead generate our own random or sequential identifier, of course, but Domino has already generated a sequential identifier for us, which we are about to discard anyway, so we may as well put it to good use while we have it.

Having set our return value to be the temporary, automatically-generated UNID, we retrieve the user's password from their account record, and pass it along with the user ID and token to a method that will determine what the corresponding API key must be when it is sent back by the consumer. Because we're already storing the MD5 hash of the user's current password, we don't need to hash it again when calculating the API key:

```

public String getAuthKey(String userId, String password, String token) {
    return HashMaster.md5(password + token + userId);
}

```

We don't *need* to calculate this yet, of course: it's the consumer's responsibility to provide this key, not the token generation process. However... immediately before we save the token document, we set its new UNID to be an MD2 hash of the expected API key. As a result, when we do receive an API request that includes a key, determining whether that key is valid – and, if so, the identity of the corresponding user – is easy:

```

public static String getUserId(String key, Database database) {
    String result = null;
    DominoUtil util = new DominoUtil(database);
    Document tokenRecord = util.getDocumentByPrimaryKey(key, false);
    if (!(tokenRecord == null)) {
        Date now = new Date();
        try {
            Date then = tokenRecord.getFirstItem("userId")
                .getLastModified().toDate();

```

```

        double timeDifferenceMinutes = (now.getTime() - then.getTime()) /
60000;
        if (timeDifferenceMinutes < 240) {
            result = tokenRecord.getItemValueString("userId");
        }
    } catch (NotesException e) {
        e.printStackTrace();
    }
}
return result;
}

```

We simply hash the key, and try to find a document with a matching UNID. If none exists, that key was never valid. If a document does match, we check the last modified date of the `userId` item; if it's older than 4 hours (here, again, is an arbitrary portion of this approach that can be changed to suit your own priorities), then it's expired. But if the document exists, and the token it represents hasn't expired, then the `userId` item tells us on whose behalf the action specified in the API request should be performed. Because each requested token is valid for a specific period of time, we can inform all API consumers how long they can cache tokens within their own application before having to request a new one; they need not request a new token prior to every API request.

In the next section, we'll pull all of these concepts together to create a functional API.

The sum of the parts

Thus far, we've only looked at the "back end" code that can be used within an XPage to register and authenticate users. So let's finish off by exploring how all the concepts this article has discussed can come together to create a fully functional application API.

The following is the entire source XML of `registerUser.xsp`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core" rendered="false">
  <xp:this.afterPageLoad><![CDATA[#{javascript:importPackage(com.puncrock.api);
try {
  var signerDatabase:NotesDatabase =
sessionAsSigner.getDatabase(database.getServer(), database.getFilePath());
  API.sendJson(API.registerUser(param.userId, param.userPassword,
signerDatabase).toJson());
} catch(e) {
  print(e);
}]]></xp:this.afterPageLoad>
</xp:view>
```

The first characteristic of this element that is atypical for an XPage is the `rendered` attribute of the `UIViewRoot` node: by setting it to false, we tell Domino not to send any markup of its own. This allows us to "print" to the browser only the specific data that we want to send. Since we are surfacing an API to consumer applications that will provide its own visual representation of the data, we can send just the data with no visual characteristics of our own. We do that using the `sendJson` method of our base API class:

```
public static void sendJson(String json) {
  XspHttpServletResponse response = (XspHttpServletResponse) FacesContext
    .getCurrentInstance().getExternalContext().getResponse();
  response.setContentType("application/json");
  response.setHeader("Cache-Control", "no-cache");
  try {
    PrintWriter out = response.getWriter();
    out.write(json);
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

This method obtains a handle on the servlet response, which allows us to not only write our desired content directly to the consumer, but also to set the content type of the response and set a header that prevents caching, just in case the consumer would otherwise choose to cache the response.

We use the `afterPageLoad` event of the XPage to call this method, but before doing so, we use the native `importPackage` function of SSJS to allow us to use an abbreviated syntax. Since all SSJS code is ultimately interpreted by Java at runtime, we would normally have to use the full Java namespace (`com.puncrock.api.API.sendJson`) to allow the runtime to load the correct method; `importPackage`

provides the class loader a shortcut for finding the method, so as long as we haven't defined any conflicting variables, we can simply call `API.sendJson()` and the correct code will be executed.

The `registerUser` method returns an instance of a `UserRegistration` class. We've previously examined its `register` method; let's take a look now at its `toJson` method:

```
public String toJson() {
    JSONObject json = new JSONObject();
    try {
        json.put("success", isSuccess());
        if (!isSuccess()) {
            json.put("error", getError());
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
    return json.toString();
}
```

This method uses the `org.json` package, which can be downloaded from (as you might expect) [json.org](http://www.json.org). Although the XPage runtime automatically includes multiple packages for generating JSON, when generating – and parsing – basic JSON structures, the `org.json` package is far simpler to use, as the above example illustrates.

If the registration request succeeded, the consumer will receive the following response:

```
{"success":true}
```

If the request was unsuccessful, however, the response will be similar to the following:

```
{"success":false, "error":"User Id timtripcony already exists"}
```

Either response provides the consumer all of the information it needs to provide appropriate feedback to the user.

The source of `getToken.xsp` is quite similar:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core" rendered="false">
  <xp:this.afterPageLoad><![CDATA[#{javascript:importPackage(com.puncrock.api);
try {
    var signerDatabase:NotesDatabase =
sessionAsSigner.getDatabase(database.getServer(), database.getFilePath());
    API.sendJson(API.getToken(param.userId, signerDatabase).toJson());
} catch(e) {
    print(e);
}]]]></xp:this.afterPageLoad>
</xp:view>
```

Once again, we block the default rendering of the page, and in its place, send back a JSON response – this time, by calling an almost identical `toJson` method of the `TokenRequest` class we examined

earlier:

```
public String toJson() {
    JSONObject json = new JSONObject();
    try {
        String token = getToken();
        if (!(token == null)) {
            json.put("token", getToken());
        } else {
            json.put("error", getError());
        }
    } catch (JSONException e) {
        e.printStackTrace();
    }
    return json.toString();
}
```

This time, if a valid user ID was provided, the consumer should receive a response similar to the following:

```
{"token":"AEB5929ED34F602FD98DB7917098AC5B"}
```

Finally, let's allow a user to actually *do* something in our application. To be precise, let's allow a user to store their current GPS position in their account record. So we'll create `setUserLocation.xsp`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xp:view xmlns:xp="http://www.ibm.com/xsp/core" rendered="false">
    <xp:this.afterPageLoad><![CDATA[#{javascript:importPackage(com.puncrock.api);
try {
    var signerDatabase:NotesDatabase =
sessionAsSigner.getDatabase(database.getServer(), database.getFilePath());
    API.sendJson(API.setUserLocation(param.apiKey, Number(param.latitude),
Number(param.longitude), signerDatabase));
} catch(e) {
    print(e);
}]]></xp:this.afterPageLoad>
</xp:view>
```

This page calls the `setUserLocation` method of our base `API` class, and passes it several request parameters, including the API key that should identify a valid user:

```
public static String setUserLocation(String apiKey, Double latitude,
    Double longitude, Database database) {
    JSONObject json = new JSONObject();
    String userId = getUserId(apiKey, database);
    if (userId != null) {
        Document userRecord = new DominoUtil(database)
            .getDocumentByPrimaryKey(userId, false);
        try {
            userRecord.replaceItemValue("latitude", latitude);
            userRecord.replaceItemValue("longitude", longitude);
            json.put("success", userRecord.save());
        } catch (NotesException ne) {
            try {
```

```

        json.put("success", false);
        json.put("error", ne.toString());
    } catch (JSONException je) {
        je.printStackTrace();
    }
} catch (JSONException je) {
    je.printStackTrace();
}
} else {
    try {
        json.put("success", false);
        json.put("error", "User not authenticated");
    } catch (JSONException je) {
        je.printStackTrace();
    }
}
return json.toString();
}

```

Because this is the longest single block of code we've yet seen, but also because it finally demonstrates a real use case for this authentication technique, let's examine it in excruciating detail.

1. We create an empty JSON object that will be sent back to the consumer to notify the user of the result of the attempted operation.
2. We call the `getUserId` method we examined earlier, passing it the API key received with the request. If the key is expired – or was never valid – we update the JSON object to convey that the operation was not successful because the user is not authenticated. This provides the consumer an opportunity to request a new token and try the same operation again.
3. If, however, the key we received was valid, we now know the user ID associated with the token used to generate the key. We can, therefore, rapidly obtain a handle on their account record.
4. Having now located their account record, we write the latitude and longitude data received with the API request and save the record.
5. Finally, we update the JSON object to convey that the operation was successful.

Expanding on this basic premise would allow your service to do more interesting things: instead of just storing the user's current location by overwriting item values in their account record, you could create a separate "check-in" document each time, storing the user ID, location data, and current date-time. This might allow consumer applications to display a chronological map of a user's travels, perhaps even calculate patterns of changes in physical distance between members of a given social circle.

What is significant about this example is not *what* is done with the data that was submitted to the application; rather, the most pertinent facet of the above method is how the application is determining whether the data should be accepted in the first place – and if so, on whose behalf.

Epilogue

In summary, let's recap some of the most unconventional characteristics of this approach to authentication:

- We're storing a hashed representation of the user's password directly within the application, but do not store their user ID within the same record.
- Despite not storing the user ID in the record that stores their password, we can rapidly retrieve the password when we need to because the UNID for that record adheres to a format that is both predictable and inexpensive to calculate.
- Once the user has been registered, their password never needs to be sent to our application again. As long as the consumer knows the user's ID and password, it can generate a temporary API key based on tokens it requests from the API.
- Our token and key format causes tokens to expire after a period of time of our choosing, allowing each consumer to cache tokens temporarily but not indefinitely. This allows users to perform multiple operations during a single session without having to request a new token for each. This approach minimizes both the total traffic to our application and the response time for each operation, but also minimizes the period of time during which an intercepted key can be used maliciously.
- We never store either the token that we send to the consumer or the API key we expect to receive in return, but because we know what that key must be, we assign the token record a predictable UNID, allowing both the validity and the identity of the user to be rapidly calculated.
- The use of a predictable UNID for both account records and token records removes the need to create views that display either document type. This reduction of the indexing burden that the application places on the Domino server, in addition to the rapid retrieval of data when requested, provides superior scalability and performance compared to traditional Domino data structures.
- Most importantly, by preventing all users – both anonymous and conventionally authenticated – from accessing any data or design elements other than the XPage design elements used to surface our API to consumer applications, we are able to keep our application's data entirely secure despite being unable to leverage the entire security model that is built into the Domino application model.

I hope that you've found this article informative, and that this approach to authentication gives you yet another way to use Domino to provide amazing application experiences to your users.

FIN