Video 1: What is HubDB and when would you use it?

Hi folks. It'm Jeff from HubSpot Academy. Let's take a look at HubDB. We'll start off with a broad definition and then wade into some uses and specifics to get a good feel for this unique tool.

So first thing's first: HubDB is a semi-relational data store. It's very similar to a spreadsheet. In fact, HubDB tables can be imported and exported from CSV files. Essentially, HubDB is a way to store page agnostic data in the CMS. You create tables with columns for things like text, images, and dates; you add content into rows; then you pull the content out with HubL into templates and modules. That's HubDB in a nutshell.

To get a bit more clarity, let's talk about some typical uses. Let's say you're working for a retailer with a number of locations. They want to create an interactive map where you can get info about all their locations. Each location will need coordinates and information about the location itself.

You could build a map like this in the HubSpot CMS using templates and modules, but if your client has a ton of locations, things could get unwieldy pretty fast. Also, if you decided after the map is built that you want to use these locations on another page or build subpages for each location, you'd essentially be out of luck. All that data exists on a single page.

This is where HubDB comes in handy. You can store these locations in a table. Then you can use this data whenever and wherever you need it. So you can build your map page project and use that same data for other future projects.

HubDB has a number of advantages in a scenario like this one, including: Separation or "normalization" of data, Ease of viewing and editing large amounts of data, and granular permissions.

HubDB has a number of advantages in a scenario like this one, including: Separation or "normalization" of data, Ease of viewing and editing large amounts of data, and granular permissions.

Let's talk through these briefly so that you can get an idea of when HubDB is a good choice for a project.

First off, there's a separation or normalization of data. This is essentially the technical term for the location scenario described earlier. It basically means that the table of data is self- contained. This is an important advantage because if the context where you use the data changes, the data is nice

and portable. You don't need to start from scratch. To use a word that executives get excited about: This means it's more "scalable."

To switch gears from the technical side: Viewing and editing large amounts of data is just more convenient in a table format. A table is very simple and concise. You can see a ton of records in a glance. This makes things easier for content editors.

Speaking of content editors: HubDB makes it possible to set fairly granular control for individual editors. For instance, you can allow an editor to edit the data in a table, but not the page where this data is used. As you can imagine, all this stuff makes delegating tasks and managing large projects much easier and more extensible.

Ok, those are some aspects of HubDB that make it a great candidate for handling projects with lots of data. In addition to those fundamental aspects of database tables, HubDB also has a unique feature that makes it a very good fit for a common website architecture challenge. If you recall from the map example a bit earlier, one of the likely additions to that example project was creating subpages for each location. HubDB makes this extremely simple with a feature called dynamic pages.

When you use a HubDB table for dynamic pages, each row becomes its own page. The URL and titles for these pages are set in their respective rows, and each page is automatically added to your analytics views. This saves a ton of time vs creating pages individually and it makes content maintenance much easier as your content grows.

Well, there you have it. Those are the major aspects of HubDB and why you might want to use it.

Video 2: Basic HubDB implementation

OK, let's take a look at how to use HubDB in a template. There are quite a few ways to use this tool, but in this video we'll focus on the most straightforward implementation so you can quickly get a feel for how it works.

To get started, navigate to HubDB by choosing HubDB from the flyout under Files and templates in the Marketing menu.

Click the Create table button to create a new table and give your new table a name. This name is for your internal reference and doesn’t need to conform to any standard.

By default, tables contain an “ID” and a “Name” column. Add new columns to your new table with the Add column button. For each column you’ll add a label, a name, which is the unique identifier for this column in HubL, and you’ll select a column type.

For the sake of this example, we're going to build a table to hold books, so we'll use the default name column for the title, a new text column for the author, a select column for book type, and a multi-select column for tags. For select and multiselect columns, you'll also need to populate the selectable options. In a real-world project you'll want to carefully plan out these columns and options ahead of time.

After you've got your columns created, you're ready to populate the table. Use the Add row button below the table to add rows one at a time. You can also import data using CSV files. Check out the resources tab for more details on that process.

Once you've set up your table click Publish to make the data public and accessible. This brings up an important point: HubDB is made to store public content for a website. Sensitive data like passwords shouldn't be stored here.

Ok, once you have a published table, you're ready to use this data in a template. The last thing you'll need before jumping into the design manager and building this template is the table ID. This can be found in the HubDB home screen. This is the same screen where you create new tables, so you can access it via the main menu as we did earlier, or by using the Back to HubDB tables link above your table.

Table IDs are listed in the ID column on the far left of this home screen table of tables. Copy the ID for your table onto your clipboard and navigate to the design manager by choosing Marketing > Files and Templates > and then Design Tools from the main menu.

Ok, we're going to use a barebones coded template for the sake of clarity and simplicity here. Since this process involves writing HubL you'll need to either use a coded template like we're doing here, or build a module.

Now, the exact details of how you pull data out of tables will vary from project to project, but we'll explore some basic concepts here and try to head off some of the most common questions.

Start off by simply looping through the table and grabbing some data. The quickest way to do this is using a for loop and the hubdb_table_rows function. This function takes two arguments, the table ID and an optional filter query, which we'll explore later in this video.

So now we're looping through each row in the table. To access individual fields, you'll use dot syntax with the column name. Let's start off with the name column. I've wrapped this in an H2 somewhat arbitrarily here to make the output more obvious.

OK, let's use live preview to see what this looks like. In the upper right-hand corner, click “Preview” and then ““Live preview with display options” to pop open a preview in a new browser tab. So far so good. We've got a bunch of h2s with the titles of the books in that table we built earlier.

If you wanted to add the author to this, you could do so with the same syntax.

And if we click back over to the preview, we’ll see those authors below the book titles.

Okay, now let's take a look at getting data out of multiselect columns. If you recall, the book table had a multiselect column for tags. Just for the sake of example, if you were to try to access this data as if it was text like so:

you would get this. This makes sense if you think about it. There can be any number of tags, so they're stored in a sequence. Note that there are two attributes for each tag here: an id and a name. In most cases, you're going to want the name, not the id.
[show preview]

So in order to list out these tags, we'll create another loop. So here we're looping through those tags and grabbing the name attribute for each tag. I've put these inside of an html i tag so the preview is clearer.

Also, if you'd like to separate these with commas, you can use a conditional and loop.last to insert commas for each tag that isn't the last one.

And if we check out the preview, we’ll see a nice little italicized list of tags below the author.

Ok. When we first created this loop with the hubdb_table_rows function, I mentioned the optional filter query parameter. In this example we're using a tiny bit of sample data. In a real-world scenario, you'll probably have a lot more than seven rows. You can reduce the amount of data you're pulling out of this table by filtering it. This topic can get quite deep, and there are many filters available. Check out the resources tab for links to the documentation.

For the sake of example, let's simply reduce this list of books to fiction titles. The syntax for these filters is the column name plus two underscores then the filter you're using, an equal sign, and then the value. So, to select only the fiction books from our table, we're using the column name "book_type" then two underscores, then the equals filter—which is eq—then the value we're looking for, which is fiction. Notice that value is case sensitive, so we're using a capital f.

If we check this out in the preview, we can see that the non-fiction books at the bottom of that list have been excluded.

Building on this, another common task in the filter query parameter is adjusting the order of the results. Filters are strung together with the ampersand symbol. The parameter for specifying the order of results is orderBy. So we can alphabetize these books by name like so.

And if we look at the preview, we'll see the books have been reordered.

OK, let's take a look at one final thing. So far we've focused solely on how to grab data from tables, but we haven't talked about any best practices. This can be a pretty deep topic, but essentially you want to limit the number of calls you're making to the database. This topic is a little different if you're using asyncronous Javascript, but if you're using HubL, you'll want to use one query to grab all the data you'll need on a page and then create subsets afterwards.

A moment ago we used a query parameter to limit our database query to fictional books. If we were building a page where we only wanted to display fictional books, this would be the most appropriate way to write our HubL.

If however, we were building a page with both fiction and nonfiction but we want to separate these book types, we should grab all of the books, and then create subsets in HubL. The select attribute and reject attribute filters can be very handy here.

Here's the syntax for using select attribute to reduce the table rows to only fictional books. We've removed the query parameter from hubdb_table_rows and we're storing all the rows from our table in a variable named books.

Then we create a new variable for the fictional books. We assign the result of the select attribute filter to this new variable. This filter takes three arguments: the attribute we're filtering on, the comparison operator, and the value. Attribute in this context is essentially column name, but you'll notice that we we're using dot syntax here since this is a select column with both a name and id for each value.

Once we have our subset of table data, we can loop over it just like we saw earlier. This is the same loop we used when we looped over all the rows, we've just replaced the table query with our new subset of fictional books.

It's worth mentioning that despite syntax differences, select attribute works a lot like the query parameter we used earlier. Once again, the major difference is that we're filtering this data within the page instead of limiting the data we're pulling from the table.

Ok, now you've seen all of the essential elements of working with HubDB and HubL. The exact strategy you use for pulling data out of HubDB tables will vary with the needs of your projects.

Check out the resources tab for links to documentation and dedicate some time to practicing both query parameters and the select and reject attribute filters.

Video 3: Creating dynamic pages with HubDB

So. One of the handiest features of HubDB is the ability to use tables to generate dynamic pages. Let's take a look at how this works. This video assumes you're already somewhat familiar with creating tables and looping through data with HubL. If you're new to HubDB and haven't watched my video on basic HubDB implementation, you should check that out before watching this one.

Ok, let's first take a look at the finished product so we're clear on what we're going to build here.

This isn't the most glamorous page, but this example will allow us to focus on the essential elements of this functionality. As you can see, we have a listing page with a number of books, and we can click through to sub-pages for each book.

OK, let's build this out. First let's start with the HubDB table.

This is a simple table full of books. To make a table dynamic, click the "Settings" button and select "Use for dynamic pages." This will add two new columns to the table: page title and page path. These columns will be the URL and title tag for your dynamically generated pages. For the sake of example, we'll simply reuse the name of each book for each of these fields.

That's all we need to do on the table side of things. Let's jump into a template and take a look at how we set this up in HubL.

We're using a blank coded template here for the sake of example. Before we start adding code, I should mention that dynamic pages require publishing a page from a template. You can't preview your rendered code directly from the template. So we'll code this out first and then walk through creating a page and seeing our results in the browser.

Ok. This template will handle both the listing page where we view all the rows in the table, and the sub pages, where we view single rows as dynamic pages. If you're familiar with marking up a blog with HubL, this is a very similar pattern.

This pattern hinges on a conditional that looks like this. In the first condition we have the dynamic_page_hubdb_row. This will be the markup for our dynamic pages. In the second condition, we have dynamic_page_hubdb_table_id. This will be the listing conditional where we loop through the table rows.

Let's go ahead and add that loop. So here we're looping through that books table and outputting the name column into an h2, the author into a paragraph, and then the tags into an i element.

One more thing we'll want to add to this loop is a link to the dynamic page. Remember from our preview that we want to be able to click through to the subpages from each of these. So we'll wrap the title in an anchor. The URL here will start with the request path. This is the root relative link for the current page. Then we add the segment for our dynamic page with hs_path. This is the page path column in our dynamic table.

Ok. Now, let's add a little markup for the dynamic pages. We'll access the columns by name the same way we did in the listing section, but we'll access them as attributes of dynamic_page_hubdb_row instead of iterations in a loop. So the name column wrapped in an h1 will look like this. And the author column wrapped in an h2 will look like this.

So that's a good bit of markup without seeing any output, but now we've got enough to move on. As I mentioned earlier, we'll need to publish a page to see this in action.

[add step here for publishing template]
So let's go to website pages and publish a page from this template. To create a page we'll navigate to "Marketing," "Website," and "Website Pages." Then we'll click "Create website page."

We'll search for this template by name and then click the thumbnail to select it and give it a name.

Now, we're going to get a blank page here to start. This makes sense because our template is nothing but barebones HubL to render our dynamic content. The important thing to concentrate on here is the settings tab. There are a couple things we need to do in there to get this up and running. First we need to fill in the required title and URL fields in order to publish the page. With that taken care of, the most important aspect of this dynamic page implementation lives in the "Advanced Options" dropdown menu. Click this dropdown menu and scroll down to the HubDB section. This is where you connect your table to this page. So we'll select the books table we used in the template. This is an important point though: The table you select here must correspond to the one you're using in your template.

Ok. Once that's all set, we'll click "Publish." Then we can click the page URL in this next screen and finally see this in action.

And now we've got that delightful page we saw at the beginning of the video. As we click into each of these books, we access the dynamic pages marked up within that dynamic_page_hubdb_row condition in our template. Hopefully your production pages will be much better looking than this one, but now you know the essential elements of getting this up and running.

Video 3: Database schema and linking tables

OK, let's talk a little bit about database schema. This is a deep topic that transcends most common HubDB implementations. That being said, the high level aspects of database best practices are very relevant to HubDB, and anyone working with this tool should spend some time on this subject to avoid common pitfalls. Restructuring database tables can get extremely expensive if you don't plan carefully.

So in this video, we'll briefly outline some best practices and then we'll take a look at a couple features of HubDB that can help you implement these best practices, including using foreign IDs to tie tables together.
[Photo by rawpixel on Unsplash]

OK, let's dive into some best practices. First off, the most important element of designing databases is a clear purpose. This may seem obvious, but it has a very special meaning in this context. In large-scale projects, clearly defining purpose is frequently a formalized development phase. Once again, this might be overkill for the typical HubDB project, but you should make sure you meet with stakeholders and get on the same page about the content you'll be storing in these tables. Sketch this stuff out, put it in writing. Some good questions to ask and answer are
What is this content? Talk about content creators, scope of content. You need to figure out where this content starts and ends.
How is it going to be consumed by visitors? You don't necessarily need high-definition designs of individual pages for this, but the better you understand this, the better you can plan.
How is it going to be updated? To some degree, this is a matter of creating a convenient editing experience for contributors, but more importantly, you want to get a feel for how this content will evolve over time.

This brings me to the bottom line: Plan well beyond your immediate needs.

The goal is to future proof your tables. Now, this is never 100% possible, but there are a few things you can do to get a bit closer to this goal.

The first thing to mention here is normalization. This is a fairly universal standard for database design. It's a very deep topic, so I'd recommend researching it separate from this brief video. Essentially though, it's a standard for keeping tables self-contained so that your data can grow without having to be restructured from the ground up. Since we can never really predict the future, the best we can do is to try to make sure that our data structure allows us to access our content in totally unforeseen ways.

Beyond this deep topic, there are some features of HubDB you can use to build tables that grow elegantly without needing restructuring after the fact.

There are a wide variety of column types that can help ensure that data is predictable and standardized. Checkbox, select, and multiselect columns are especially helpful here. Try to look for opportunities to substitute these columns for more open-ended options like text or number fields. If there's any possibility that a value will be repeated, use these columns to maintain consistency. Never rely on content editors to enter data "correctly."

To take this one step further, foreign ids are one of the most powerful features of HubDB. Use these columns to link multiple tables together. The best way to keep your data structures evergreen is to make sure that individual tables only handle one thing at a time. Related data should be organized in separate tables whenever possible.

In this example table, the authors column is drawing from a foreign table. In some cases, a list or multiselect might be a good option. In this case, a foreign table is handy so that we can store data related to these authors.

Here's a look at that authors table. The bio column here is a great example of when you'd want to create two separate tables. This data is related to the authors not the books themselves. If we were to store this data in the books table, we'd open ourselves up to inconsistencies the moment we had two books with the same author. Content editors might be willing and able to copy/paste that content in a small table, but what about one with hundreds or thousands of records?

It's up to you to determine when to break data out into multiple tables, but always try to imagine your current data growing by 100 or 500 percent when you're making these decisions. Check out the resources tab for more info on foreign ids.

Video 4: Using HubDB with JavaScript and the API

Since using HubDB often involves working with large amounts of data, this is an area of the CMS where you're likely to stray beyond HubL for more complex projects. Let's take a couple minutes to explore the available options.

There are essentially two tiers of common HubDB implementations that don't use HubL: authenticated access and public access. Both of these use the HubDB API. Let's take a look at each of these.

Published HubDB tables are public. This is an important aspect of HubDB. It means you should never store sensitive data there, but it also means you're free to use client side code to pull data from HubDB.

Let's review some details of unauthenticated access:

First off, you can only use GET requests here to pull data from tables. If you need to write data to tables, you'll need to authenticate.

Also, to be clear, you can access HubDB this way either inside or outside of the CMS.

We won't go into specific implementations here, but you can use AJAX and JS frameworks to build single-page, app-style displays of any variety. You'll want to use the API docs as reference for building calls.

Check out the resources tab for links to these docs. It's also worth mentioning that there's a Javascript client for making these public requests even simpler. Once again, there's a link in the resources tab.

OK, to modify HubDB tables using the API, you'll need to authenticate via an API key or OAuth. So if you're trying to build something that involves collecting user input and storing it in HubDB, you'll need this level of access.

Technically, you can use Javascript to authenticate with an API key inside the CMS, but this will be client side code and therefore totally public and incredibly reckless in a production environment.

So you'll need to either mask your authentication with a third-party service or proxy server, or build your project outside of the CMS. Since this subject reaches beyond the limits of the CMS, we'll leave it there, but once again, check out the resources tab for some options for further exploration.

Video 5: Importing and Exporting tables with CSV files

Ok, let's take a look at importing and exporting a HubDB table. This workflow will look different depending on exactly what you need to do, but in this example we'll take an existing table, export it to a CSV file, import it to google sheets and edit, then export and import back into HubDB. While you can use Microsoft Excel for this, it's important to mention that excel has a nasty habit of modifying special characters. So we recommend google sheets if you have a choice.

So here's a little table with a list of books. We have two text columns for name and author, a select column for book type, and a multi-select column for tags. To Export we'll click the Export button and choose CSV from the drop down. This will trigger a download. Now let's jump over to google sheets to import this CSV.

Under the file menu, we'll choose import. Then we'll navigate to our recently downloaded file and drag it in. In google sheets we get a dialog box where we can choose a few options for this import. We're going to leave all of these settings at default for this example and click Import data.

So now we've got our data in google sheets. Now, there are a couple of unexpected extra columns here for path, created at, name, and child table. These are part of the dataset on the HubDB side for things like dynamic pages. They aren't relevant to our task at hand so we'll ignore them for now. That being said, it is worth noting the id column all the way on the left. We can also leave this blank and let HubDB assign an id when we re-import this data.

Ok, we'll add a new row here with another book. Let's take special note of these last two columns. If you recall, in our HubDB table these are select and multi-select columns respectively. I'm going to add Fiction for book type, which is used throughout this column in other rows. But I'm going to add Magic Realism as a new tag in this multi-select tags column.

Now, let's go ahead and shuffle this data back into HubDB. Let's give this a name for the sake of clarity, then we'll go up here to File, choose download as and then choose the CSV option from the flyout. This will trigger a download and we'll get a file with the name we just specified.

To bring this back into HubDB, we'll jump back over to HubDB and choose import. This will open up an importing wizard. The first screen of this wizard gives us a couple options. We can append this data to our current table, or we can replace the table. In this case, we'll choose Replace since we've been editing the existing dataset not creating completely new content.

After clicking next, we can select our downloaded CSV file and click next again to see the map fields screen. Once again, we'll simply ignore the columns with the hs prefix. If you did happen to create new columns in your spreadsheet application, you could create new columns in this screen to map that data to. In this case, things look good as-is. So we'll click Import. This will give us a little warning since we're about to overwrite an entire table. If you're not feeling confident about this, you can backup the table you're about to overwrite by Exporting it here.

Click Finish import to complete the process. We get a little success message, and then we can click done to see the results.

Now. If we scroll down, we can see that last record has been added to this table. As promised, the ID column has been updated for us by HubDB. Let's take a look at those select and multi-select columns though. Here we see that Fiction has been successfully imported as book type, but that Magical Realism tag was not imported. The lesson here is that we can't create new values for select and multi-select columns in the spreadsheet, but we can use existing values.

This brings up the larger topic of strategy and prep work. HubDB provides a lot of intuitive control over this import/export process, but you'll want to use caution when shuffling around data like this. If you're using this as a collaboration tool, make sure you're prescriptive with your collaborators about things like adding new columns or multi-select values. It's a good idea to do tests on cloned tables and backup data before you overwrite it.