

LITERATURE 101

STUDY / REPORT

JONNY F. BRATEN

Tables of Contents

Introduction & Goal	2
The assignment	2
My Website	2
The Website's Design: Aesthetics	4
Picking the Right Tools for the Task	4
Layout	5
Styling	7
The Database Design	9
The Contents of my Database — Users	9
The Contents of my Database — Literary Devices & More	10
Normalization	12
Logic & Scripting	16
HTML	16
PHP & MySQL	17
JavaScript	18
XML	21
Editing Content	22
Users	22
Adding, Editing, & Deleting Content	23
The Result	25
A Dynamic User-Driven Website	25
References & Resources	26
Styling & CSS	26

Introduction & Goal

The assignment

The general, overarching goal of this assignment was to develop a dynamic web page that also connects to a database, in which some kind of data suited to the type of web page were to be stored and retrieved; the key words being *dynamic*, and *database*.

The theme and subject of the web page itself was left up to each individual student to decide (much to my delight) assuming it could somehow be made to fit and meet the requirements above. That said, the assignment also encompasses the consideration of a user-base or audience—hypothetical or otherwise—that the constructed site may seek to attract and please with what it has to offer. The audience or customer(s) may still be pretty much any hypothetical human being, including the student attempting to create the page, but the idea behind this was to consider how to best reach out to target-audience and deliver what they'd want, regardless of how strange or niche that thing might be.

For the technicalities of the assignment, it was divided into two parts, where each part would need to be completed on two different dates. The initial part was simply to specify what kind of web page to create; this means what the web page aims to display by the end of the project, what it's meant to store in its database, who the potential users are, and explain the general gist and purpose of the website. This part was good for getting ideas flowing and forcing the brain to try to get into the whole data-base mind-set, but also had the added function of getting some feedback and confirmation from our lecturer that our idea wasn't somehow going to entirely miss the point. The things specified in the first part will also be covered here and greatly expanded upon.

The second and by far biggest part of the project was to actually create a webpage from scratch, learning the required methods for doing so along the way. Each student received a domain with the necessary preparations made for supporting a database, and the task from there was to connect to the domain in all ways necessary to both upload a functioning webpage and to interact with the database, and then of course to actually develop the web content to be uploaded.

My Website

Seeing as we were at such liberty to pick and mold what the goal for our websites would be, I spent a lot of time racking my mind for suitable ideas. Ask anyone who knows what a website is, and you could probably also get an answer if you were to ask them what specific subject they would like to see a website portray, and I myself was at no shortage of ideas. What I found to be the limiting factor was the database-side of the assignment's focuses.

The inclusion of a database in itself is obviously not the problem, as the use of databases is so wide-spread it seems like nothing short of a necessity in most cases, but I found myself

spending most of my thinking-time piecing together how I could create something that I wanted to make and have it utilize a database and a user-system, and have it utilize it in a way that would make sense, both for the assignment and for my website. Granted, the assignment does not specifically state that the combination of the database and its user-base *needs* to make sense in the context of the website, but given how we were given time to try to think of ideas for what to make, it would be the natural way to approach it.

Personally I had no actual clients in mind, nor did I have any particular website in mind that I felt like was lacking from the world wide web and at the same time would fit what I needed to make, therefore my mind quickly strayed to personal interests. Finally I decided upon creating a website focusing on literature, with a focus of individual terms and aspects within literature, aiming to structure in a way that would be complimented by the use of a database. This began with the idea of adding literary devices into a database along with definitions and information, which then later expanded to include punctuation, and then literature/based videos as an attempt to include a dynamic media type as well.

The Website's Design: Aesthetics

Picking the Right Tools for the Task

Despite how the heart of a dynamic web-page lies in the features and services it offers, and despite how the website's usefulness will ultimately be decided by its ability to deliver those features effectively, the visual design of it floats alarmingly fast to the front of the mind when trying to take your target audience into consideration. The most obvious reason for this is of course that if you start designing the actual navigation of the page, you've already begun to define its usability, seeing as the navigation is one of the things that would also define whether a website *is* able to offer its services effectively, but the aesthetics alone are also hugely important. You want something that's visually appealing, but not something that distracts from the site's content; you want something that stands out, but not in a way that disorients your user; you also want to capture the target audience's attention, assuming it needs catching, which bring us onto the use and choice of media types.

While it's true that we were originally tasked with creating a dynamic web-page, the dynamic nature was something that was required of the actual data-content on the website. When speaking of media types on the other hand, the word dynamic takes on a very different meaning, and whether to use it or not—or often more importantly *how* to use it—becomes a question of design and what audience you're attempting to appeal to. Here, dynamic (or temporal) media types refers to media “in motion”, such as moving images, videos, music, etc. while static or (or non-temporal) media types are—much like the name implies—more still media, such as regular images and text. While blocks of texts may be utmost informative, it's not nearly as exciting, or at least not nearly as naturally attention-grabbing, as a playing video. On the contrary, the video itself might be far less informative and structured despite its visual appeal, and so the question becomes what you ultimately want to achieve. Do you need to grab your audience's attention with a firework of motion, or do you need your website to read like a textbook? The answer, I believe, often falls somewhere in-between.

In my case, the people I've sought out to appeal to are people who are naturally interested in literature and reading, and as such, text becomes a very big part of my design. Not to say styling the website was not prioritized, but just that the goal I set out to achieve was to supply resources to those who were already interested and came seeking it, as opposed to attempting to establish an interest to begin with. I did however consider the latter issue, simply because I found the potential issue of having to grab someone's attention through dynamic media types in order to get them interested in a type of media that at its core is so very static to be interesting. Despite this not being an issue for me, had it been so, my conclusion is that one good solution would be to put dynamic media on the fore-front of the website in order to pull in attention, but perhaps doing so by using videos explicitly focused on reading and writing

(much like those I later added to my own website) and then using those videos as introductions to the more static media in the background. In my case however, my media types remains largely static, much as the media it aims to inform about.

In the end I do believe my addition of a third media-type benefits the page, and it certainly adds more diversity to its appeal. I stayed away from any sound/music (aside from that included in the videos) because I did not find any particularly natural need for them, and I wanted to force as little unneeded media onto the page as possible. Besides, I think most would agree that websites that automatically begins playing music and other sounds without the user's go-ahead does not usually make up the ideal web-browsing experience.

I do however my media-types are appropriately chosen, and besides the fact that the videos originally began as more of the challenge of simply trying out a third media type and using it in the design, I think my website is richer and more diverse for it. My goal is ultimately to inform those who are already interested, but with more dynamic media there's always the potential for catching the attention of those originally not interested in literature, and I truly do believe that—and I believe the videos I've picked are great examples of this—subjects and topics, no matter how static and unmoving it may be in its nature, can benefit from dynamic media, because even if you can't/won't make the thing you're presenting dynamic in itself, dynamic content can always be a way into the static ones.

Layout

I realized as soon as I sat down to create a storyboard for my site that the things I wanted to portray, I wanted to show in a somewhat condensed form, in the sense that all you needed was all in one place, simply waiting for you to grab ahold of it and read/watch it. I did consider splitting up my design, separating the categories of my contents throughout pages, with each category having its own hub of some sort, but it seemed like a far messier solution to me, and I didn't really feel like the content should be separated to such a degree in the first place.

The key to my design basically became having one shared and consistent interface that the user would use to both navigate and view the content of the website, ensuring that the moment the site clicked for them at whatever piece of content they were viewing, they would know and understand how to navigate the site in its entirety.

I decided on a design that consists of two hierarchical navigational-menus to browse content, where one menu determines the available options in the second menu.

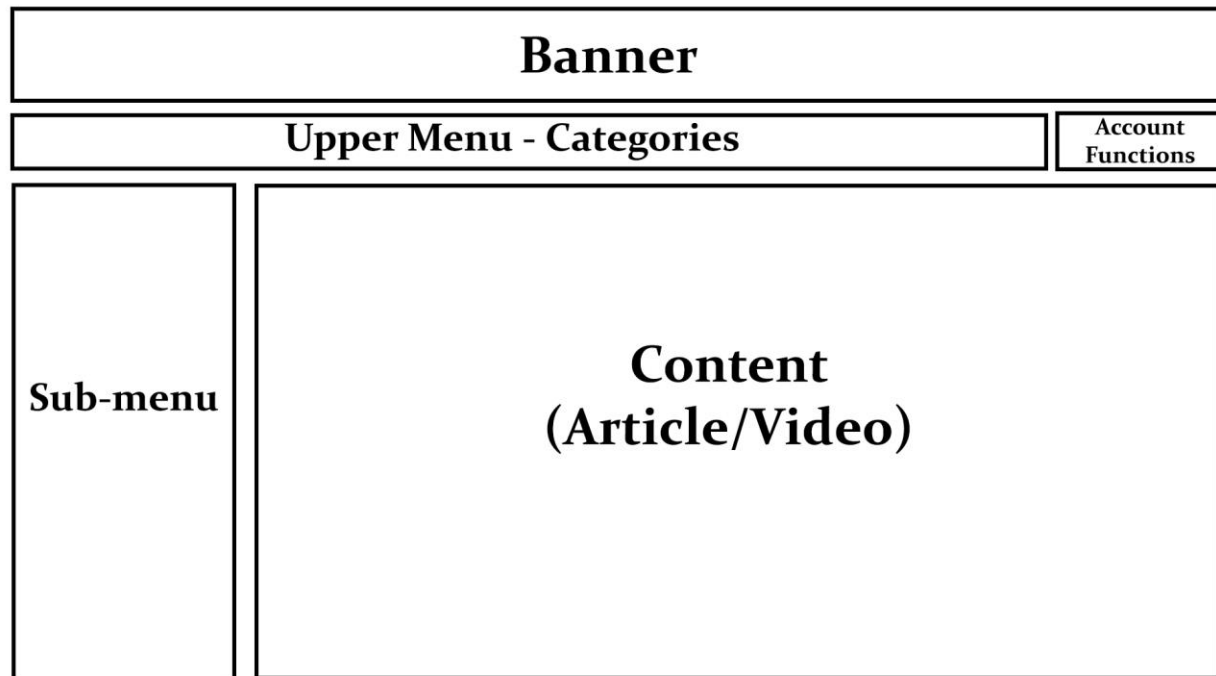


Figure 1: Storyboard — Layout

The topmost navigational menu is used to swap in between the major categories, while the lower, vertical menu is used to select what exact piece of content you want to view. I turned a few different layouts over in my head, but the most natural thing seemed to have the parent navigational-bar on the top of the screen, and then make it stand out a bit through styling, basically having users “work their way down” the page when navigating. The placement of the menus remain the same, and the contents of the upper menu that represents the major categories always stays the same content-wise. The options in the sub-menu on the other hand shows the content available within each selected category, so selecting a new category in the top-bar will make all content within that category appear listed alphabetically in the sub-menu, allowing the user to brose them. Clicking on a piece of content-name within the sub-menu will again fill and replace the content-section of the webpage with the content that corresponds to the selected subject. This way, the layout and navigation feels consistent and coherent, and it also allows users to browse content without actually replacing the article they’re currently reading.

Keeping with my intention to stick mostly to a single central hub for the page, my plan was to have log-in, register functions, content-editing, etc. in pop-up forms above the main site, but due to time constraints and technical difficulties I went with an alternative solution where these forms have their own page they appear on. All of them do however have the same layout and styling in order to keep with the consistency I wanted.

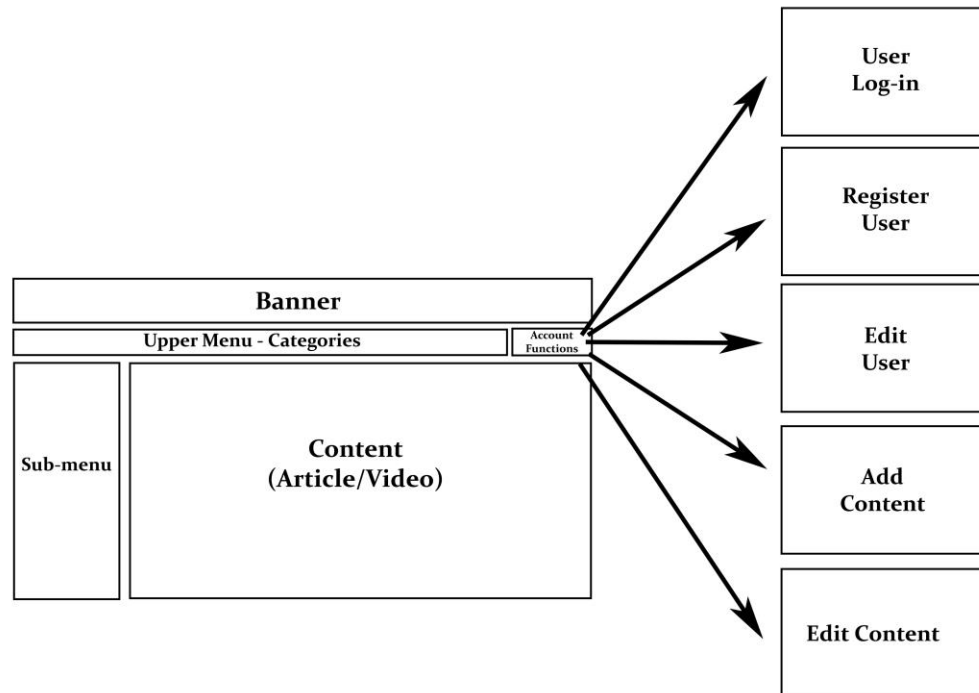


Figure 2: Storyboard — Site Map

Styling

Much like the navigation of the page, I wanted to keep the styling of the page simple, but also make it appealing to an eye that may attract some attention. Text in itself is not very attention-catching, and a couple of bright colors can often help with that, at least to a small degree, and do so without distracting the user from the content itself. I used some graphics from a series of animated literature-focused educational videos, and edited them to suit my needs. Aside from a few hints and nods to books and literature in general, and perhaps a few references to specific literature, I wanted to keep the graphics mostly clean and abstract.

For the color schemes I went with something a bit vibrant to contrast the plain text, using a couple complementary colors, giving the site a more energetic look.

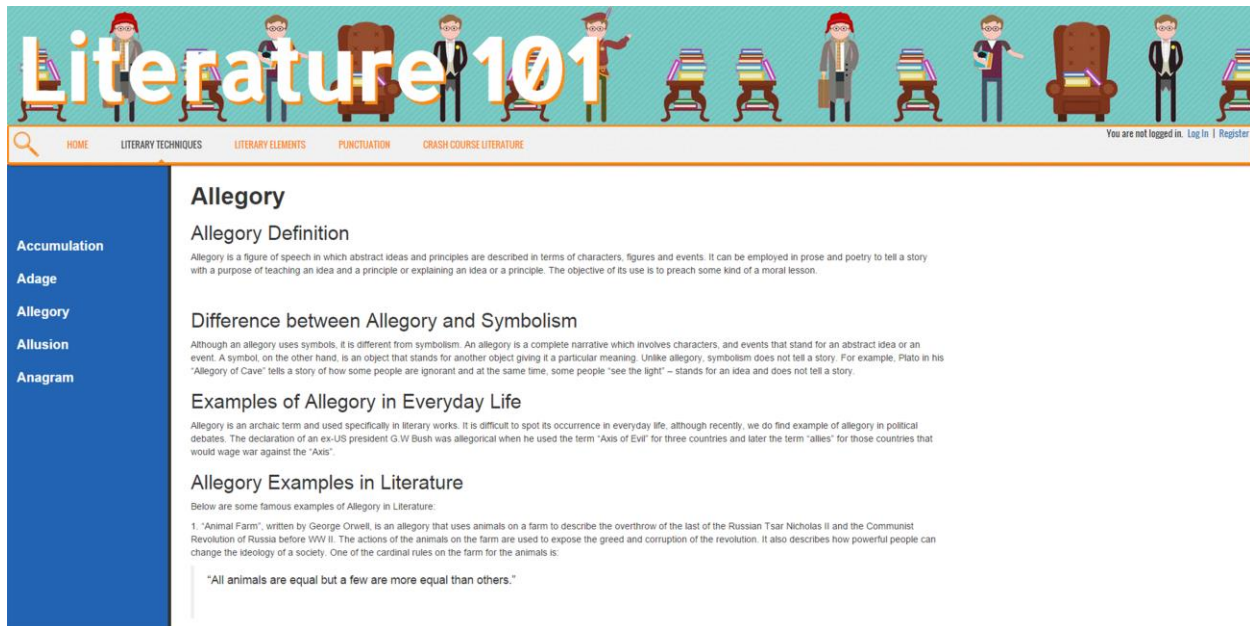


Figure 3: The Web Page

The forms that are used to register/edit/log-in users, as well as the larger forms for editing/inserting new content, were styled to go together with the main-page, especially because they were originally meant to pop-up as an overlay on top of the main page, but the styling was kept even though the forms were moved to separate pages. Sticking to similar colors also helps to give users a feeling of consistency across the pages.

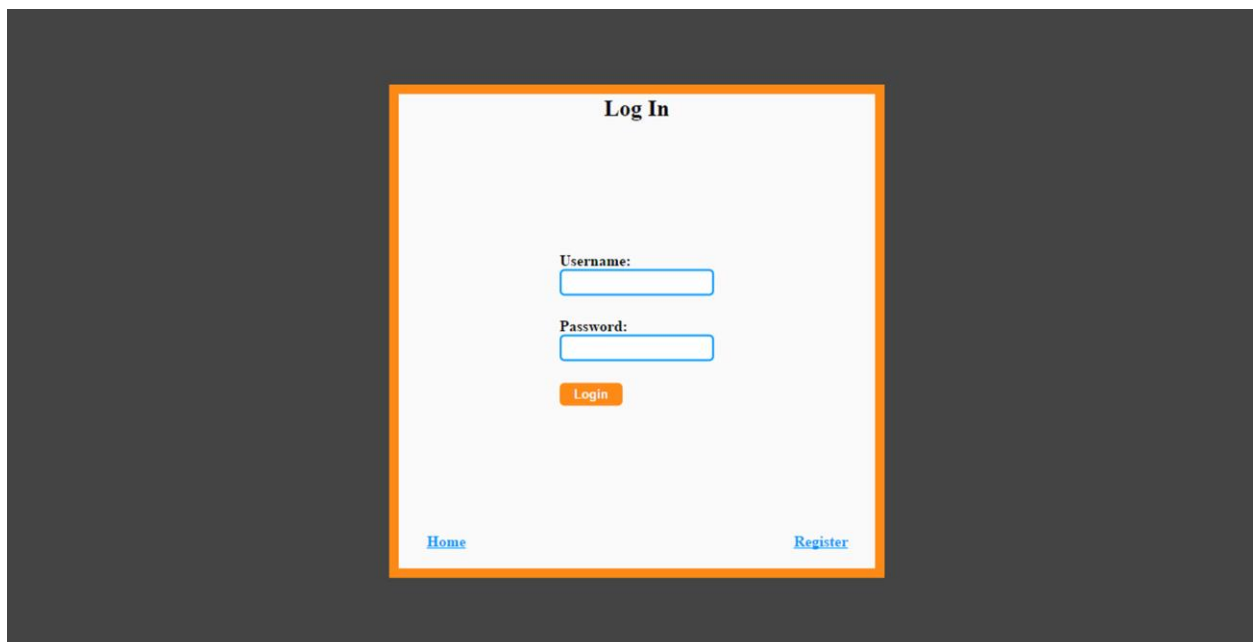


Figure 4: Form Example

The Database Design

The Contents of my Database — Users

Being able to register users and access them was one of the requirements from the get-go, so this is the more obvious piece of contents within my database. My website doesn't really revolve around users as much as it does actual content though, and as such the user-data itself does not have much unique attributes to speak of.

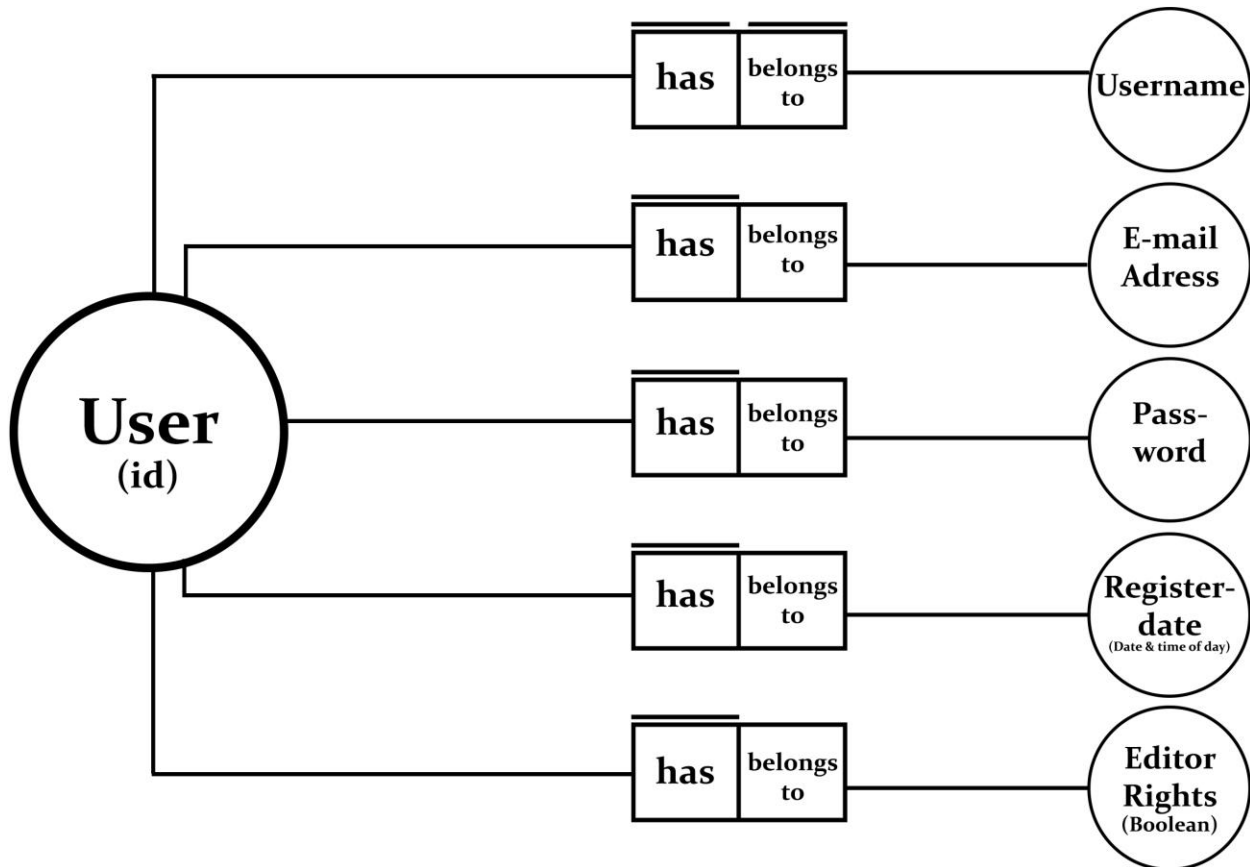


Figure 5: Database entity relationship diagram — Users

The users' are stored in their own table, with each row corresponding to a single user account. Most importantly, every user gets a username and a password, where—following common norms—the username is used to identify and distinguish the users, and the password is a private part of information that each individual user uses to prove their identity upon logging into the page. Both username and password must be given during the process of registering a user, along with an e-mail address (admittedly, I do not currently make any use of this e-mail in my code, aside from storing it as a user's contact information). My user-table also utilizes an user-id for various reasons. For starters I wanted to make an attempt at implementing and using an auto-increment value in my system, but functionality-wise I also wanted something that would allow me to refer to users without identifying them by public usernames, and I also

thought it wise to have an easier way of comparing and retrieving users than searching for and comparing potentially large strings.

Finally, each user has their own Boolean value that tells the system whether a specific user has the rights to add, edit, and delete content from the site. It's obviously not appealing to have every single user be able to edit the content, and as such this value is used to distinguish those users trusted with the right to edit content from the rest.

The Contents of my Database — Literary Devices & More

The more unique data in my database (unique in the sense it's there as a part of the specific self-tailored part of my project) is of course the literature-based content, which mostly takes the shape of literary devices. I later expanded upon the contents a bit, but the general layout remains the same.

The idea stemmed from the want to store literary devices in my database. I deemed them rather fit for a database structure seeing as they are rather modular in the way that they may each have a name/identifier, and then content that defines and explains it.

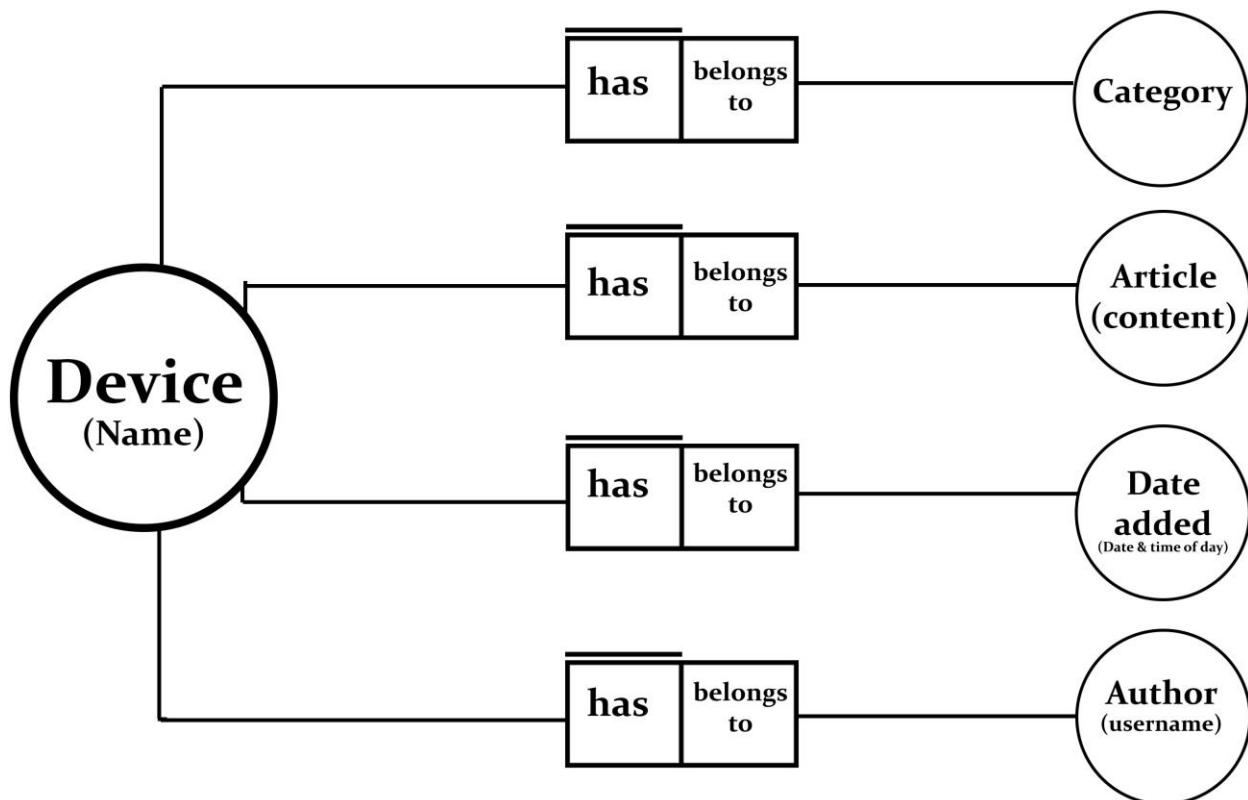


Figure 6: Database entity relationship diagram — Literary Devices

Note that while my relationship diagram states that several literary devices may have the same content, this obviously wouldn't make much sense, as every article would be tailored to to what

they're explaining, but it also doesn't make sense to enforce non-duplicate articles in my opinion, so while I don't see why anyone would do so, it is technically possible to add the exact same article for two different literary devices.

The extensions I later made introduced punctuations and literature-based educational videos to my database, but their set-up and relations remains largely the same. It's only their placement in the database and their content that truly differs, and so when I refer to literary devices, the same can now generally be said for punctuations and videos as well.

Each literary device has a name or term that distinguishes it, for example "metaphor".

Metaphor is a literary device, but can further be specified as a literary technique. Hence, each literary device or piece of content in general has a category in which it belongs, dividing the content into relevantly sorted batches. Next, each literary device has a short-ish article that defines the device and explains how it is used. I say article, but as I introduced videos towards the ends, "content" might have been a better term looking back on it. In the end, the videos themselves are simply embedded using a string of text, so it may still be written/pasted into a form much like you would an article.

This is really the meat of the content stored within the database, and it could potentially store a very large amount of text if you wished to write a long article. If you were to write a long enough article you might even find this property to seem too large and messy, at least the thought crossed my own mind as I was designing it. I contemplated diving the article into multiple parts, such as "definition", and perhaps "examples of use", but the issue with that would be to have the stored content suit those defined parts. As I later discovered, I did indeed end up adding things into the database that I originally wasn't expecting to add, and it goes to show that I was able to do that quite easily for the very reason that I kept "content" as a single field. A user could just as easily add and embed videos in place of an article all on their own without me actually adding a video-category, although it wouldn't be quite as tidy. In the end though, I would probably further divide the content into parts and customize them to suit their respective tables, but seeing as they were all a part of a single table for a long time, this option did not occur to me as feasible (more on that in the normalization sub-chapter).

Moving on, each literary device has a time-stamp connected to it that tells when the device was registered and added to the database. This is simply for the reason of being able to see how old or new a device might be in the context of the database. Similarly, each device has a column to tell who authored that piece of content, or more specifically, what the username of the account that added the content is. This is also where the tables of content connect with the user-table, giving the option to reference to an account if you're curious about the account that added a piece of content.

Normalization

The process of normalizing my database was at the forefront of my mind when designing my database almost from the start, as this was one of the early subjects we tackled during lectures, but I also found the act of doing so becoming more and more important as I went on with the assignment. I'll admit that more foresight and knowledge when starting out would have helped in the matter, but my database also grew as evolved as I developed my website, and as a result of this, my database only appeared to becoming ever less normalized. My table used for storing users remained largely the same, but the literary devices and other content began causing problems as their content grew more diverse.

My original idea centered around having one table dedicated for my literary devices, and as I went on I soon realized I would like to divide my literary tools into categories: literary elements, and literary techniques.

Name	Category	Article	Date	Author
Metaphor	Literary Technique	Metaphor is a l...	2/9/2015 21:00	Jonny
Narrative	Literary Element	The term narra...	2/9/2015 22:00	Jonny
Theme	Literary Element	Often confused...	2/14/2015 13:00	Tobias
Moral	Literary Element	In literature, m...	2/16/2015 12:00	Tobias
Simile	Literary Technique	When speaking...	2/20/2015 8:00	Orlando
Hyperbole	Literary Technique	Often used in...	2/22/2015 10:00	Vincent
Allegory	Literary Technique	An allegory is a...	2/25/2015 22:00	Jonny

So far so good, at least going by the rules of normalization. The 1st rule of normalization states that there is to be no repeating groups, which is taken to mean that no attribute is allowed to have more than one single value associated with it. Take for example the "author" attribute; its purpose is to supply the username of the account that registered the literary device to the database, but let's say we'd like to know the username of every single person who has ever edited the literary device, given that there are more than one. In this case, we'd wind up with several authors per row, and a literary device such as "theme" might be associated with both Tobias and Orlando, preventing my table from attaining first normal form. With this set-up on the other hand, this does not become an issue.

There is however some that would define repeating groups differently, and claim that my category-attribute has repeating groups merely because its contents are hardly diverse. There is

something to this, which I'll get back to, but at the very least going by the definition referenced above, the table is in the first normal form.

In 2nd normal form, a table's attributes are dependent on the *entirety* of the key. My original table also fulfills this requirement, as none of my attributes are dependent on the subset of the key, which would commonly happen if I was to use a composite primary key. A composite primary key would be a key consisting of multiple columns, say, both a device name and an ID. In a case like that, I'd risk having the attributes be dependent on either the id or the name *alone*, contrary to them being dependent on the both of them combined. Seeing as I don't use composite keys though, I get off rather easy on this one.

For the 3rd normal form, the requirement stated is that every single attribute on a row *must* be dependent on key attribute, and they key alone (and also already be in second normal form). That's to say, if the "name" in the table above functions as the key, then every other attribute must dependent on the name of a literary device. Put differently, the other four attribute needs to derive from what the name is, which they do in the table above. Now for the sake of this report, let's say this was not the case; let's assume that every single literary device in the "literary element" category was written by Tobias, while every device in the "literary technique" category was written by Jonny. In that case, the author attribute would be dependent on the category attribute, which again would be dependent on the name—a so called "transitive dependency" between the author and the key.

The table above on the other hand already meets this demand, as the name alone decides what category it belongs to, what article it contains, the date it was registered, and the username of the person who wrote it. In any case, any hypothetical emerging patterns that could hint at transitive dependencies would at the very best only be coincidences (one user may for instance happen to prefer to add articles exclusively within the "literary elements" category).

What I did come to eventually realize, was that because I only had two types of categories, those two categories would fill up a whole lot of columns despite there only being two of them. I later introduced two new categories—punctuations and videos—and while you'd maybe think having more variation within the categories would make the table seem more normalized, I found it only helped emphasize the messiness of the design, and point out the redundancy in my database.

Name	Category	Article	Date	Author
Metaphor	Literary Technique	Metaphor is a l...	2/9/2015 21:00	Jonny
Narrative	Literary Element	The term narra...	2/9/2015 22:00	Jonny

Theme	Literary Element	Often confused...	2/14/2015 13:00	Tobias
Moral	Literary Element	In literature, m...	2/16/2015 12:00	Tobias
Simile	Literary Technique	When speaking...	2/20/2015 8:00	Orlando
Hyperbole	Literary Technique	Often used in da...	2/22/2015 10:00	Vincent
Allegory	Literary Technique	An allegory is a...	2/25/2015 22:00	Jonny
Em-dash	Punctuation	Not to be confu...	2/25/2015 23:00	Jessica
Hamlet pt.1	Video	Embedded...	2/26/2015 8:00	Orlando
Frankenstein	Video	Embedded...	2/26/2015 10:00	Vincent
Comma	Punctuation	The comma is...	2/25/2015 23:00	Orlando

Let's say my database was to grow beyond my expectation, potentially reaching several hundred pieces of columns. In that situation, the category attribute will suddenly seem infinitely more redundant with the same categories repeating over and over. To combat this, I decided to re-do my database design and the way I accessed it by separating each category into their own table.

Name	Article	Date	Author
Metaphor	Metaphor is a l...	2/9/2015 21:00	Jonny
Simile	When speaking...	2/20/2015 8:00	Orlando
Hyperbole	Often used in da...	2/22/2015 10:00	Vincent
Allegory	An allegory is a...	2/25/2015 22:00	Jonny

Name	Article	Date	Author
Narrative	The term narra...	2/9/2015 22:00	Jonny
Theme	Often confused...	2/14/2015 13:00	Tobias
Moral	In literature, m...	2/16/2015 12:00	Tobias

Name	Article	Date	Author
Em-dash	Not to be confu...	2/25/2015 23:00	Jonny
Comma	The comma is...	2/25/2015 23:00	Tobias

Name	Article	Date	Author
Hamlet pt.1	Embedded...	2/26/2015 8:00	Orlando
Frankenstein	Embedded...	2/26/2015 10:00	Vincent

This solution optimizes my database and eliminates what would potentially develop to a lot of redundancy. The result is four separate tables in the third normal form, and an overall more tidy database. When adjusting my php-code and my SQL-queries to suit this design, it also quickly became even more apparent that this design was a natural choice, also in terms of accessing the database. On my website, the content is visually divided into categories in the first place, and as I was making the logic behind shuffling through the content, I became aware that I really wanted to access my devices based on their categories in the first place. Designing it this way ultimately makes my php and JavaScript far easier to deal with, as taking care of categories on an database- and SQL-level relieves me of the quite more complicated task of taking care of that logic with my other scripts.

Logic & Scripting

HMTL

The basic form of my HTML is rather straight-forward and logical when inspected independently of related php/JavaScript code in my opinion, and as such I do not find that I have a need to explain it in depth. Using design-sketches to pinpoint me, I simply made sure to group all of my HTML into <div> container elements, grouping them together in the way that seemed to make the much sense. What in general made the most sense in my opinion, correlated very much with the elements I had drawn out on my storyboard. Generally, everything related to the banner would go into a banner-container, everything on the topmost navigation bar would go into its own container, and if I found it necessary I would further divide sub-elements within each container into yet smaller container, ensuring that each container encapsulated a common type of content/elements. The most obvious examples of this making sense is in the way my sub-navigation bar to the left in my website basically lists my content in a list-form, making each piece of content-name into a element. In a case like this, it not only makes sense within the context of the content I display and their relativity to each other, but it also makes sense simply within the context of how you'd usually write good HTML.

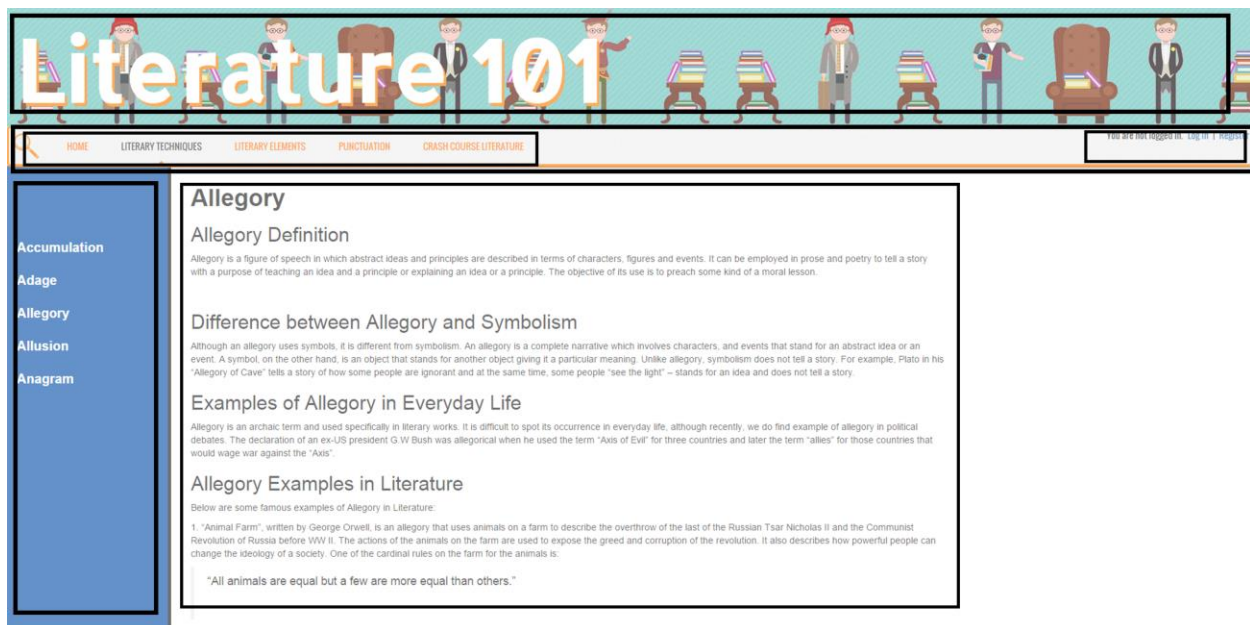


Figure 7: The webpage layout, derived from the storyboard

PHP & MySQL

While I have made note of and is very much aware that it's of no interest to include large chunks of code in this report, I do believe it would be in its place to explain my code in more general terms, and then refer to a few cases of code where I believe it either helps me make a point, or where I believe the code is unique and central to my design. I should preface this by saying that working on this project has taught me a whole lot about creating and structuring code in the context of webpages, and for whatever webpage I begin creating next, I believe the making of this one will hugely influence and also improve the way I set out to build and structure my code from the get-go.

Nevertheless, I also tried to make out a structure that seemed neat to me when I began creating my first real website—this one.

The general idea for my structure is to have a clear hierarchy between HTML, PHP, MySQL, and JavaScript. The most reoccurring hierarchy though, is that between HTML, PHP, and MySQL, with the database at the top. Granted I had no experience in what would be considered clean or even commonly good website-code was extremely limited, I knew I didn't want to mix the three to the point where I would suddenly place a MySQL query in the middle of my HTML code out of nowhere. To avoid this, I created a small group of PHP-classes for my required purposes, and then created the functions in those classes that would make the general kinds of queries to my database necessary, and return the result in the format I'd need given the context.



Figure 8: General php-classes for database utility

The “DB” (or Database) class takes care of the most basic database functionality, such as connecting to the server. This is essentially the part of my code that is the “closest” to my database in terms of its functionality and its relations, and it is the class that establishes and

retains the link between the database and the rest of my website. Obviously, I want this connection in just about every context of my website, considering its dynamic, and as such it only seemed natural to separate the most general database functionality into a class of its own, in which I make the things I know I will always be in need of.

The “user” and “device” classes are simply just classes to represent my database entities in an object-oriented fashion. When dealing with a user or device, I can therefore simply create or use an object of one of these classes in order to easily apply whatever operations I want. The “-tools” classes are the ones that handle the actual functions I might want to apply to a user/device or groups of those.

To give an idea how this benefits me, let’s assume we’re registering a new user. The user opens the register-form, fills each field successfully, then submits the form. Handling this request becomes very simple and neat, as I simply create a “user” object, which have properties equal to its obligatory properties (name, password, e-mail, and the current date/time). The constructed user-object is filled with the submitted data as the form is validated, and when the user-object has all the properties it needs filled, a “save” function from user-tools is called, which takes the object as an argument and stores it in the user-table in the database, using the values now contained within the object.

The literary-device and their tools follow the exact same principle, only the devices-objects naturally have properties that corresponds to its own columns in the database, and some of the functions in the user-tools and device-tools vary, as the operations I need to perform on the two differ.

The functions of these classes are also where all my MySQL queries are made, meaning that whenever I want to make queries to the database, I go through these classes, *not* directly from the scripts displaying my website or directly controlling the logic for it. I find this to be a good way of building tidy layers and a sort of hierarchy in my code that prevents queries to pop up in unexpected places. Likewise, I know that if my queries are not acting the way I’ve expected them to do, I simply need to go into one of these classes and inspect the function containing the query. An example of such a function might be a function from literary devices, which takes the name of a “category” as an argument, and then fetches every element of that category and returns them in an array.

JavaScript

I utilize some JavaScript on my website in order to achieve the interactivity I want, and to make the menus behave the way I intended them to do. All the JavaScript for my page is contained within a separate js-file that is simply included in the header of my main site, neatly separating the JavaScript from the rest of the code. The fact that it also only handles my two navigational menus and the things necessary to operate them makes it feel very modular, enabling me to

make quite a variety of changes where I needn't worry about something potentially breaking in other parts of the code as I make them.

Both my navigational bars, albeit separated, are made up of a list of elements, with each element corresponding to a "button" from the user's perspective; the user clicks an element, and the page responds accordingly. These clicks register as so called "on-clicked events", which are handled by the JavaScript.

Now we obviously want a different response for each specific button we click, and we want the response to correspond to whatever we clicked, so the goal essentially becomes to trigger a different on-click event for every single option there is to click. This might sound fair for the top navigation-menu. There are only five options to switch between after all, including the search-option, which amounts to setting up five separate on-click functions that trigger when their corresponding button is clicked, and then performs the requested operation. But then there's the sub-menu, and this is where things get tricky. Each choice in the topmost navigational bar corresponds to a category, and needs to display the elements contained within that category. Considering we could potentially have a long list of elements, and then that we might have an equally long list of every category, the number of on-click events and functions grow exponentially. Creating one for every single element isn't just unfeasible, it's nearly impossible to do in such a straight forward way if you want it to function with every new piece of content the users add without having to change the JavaScript. It needs to be *dynamic*.

The solution to this becomes a combination of JavaScript, php, HTML, and SQL all coming together. The top navigational bar is mostly all JavaScript, where clicking one of the options exchanged one sub-menu for another category's sub-menu. The sub menu on the other hand is dynamic, as it directly acquires its content from the database. For the sub-menu, my php classes uses an SQL query to select every element within a category in an alphabetical order, and each element is then added into the HTML as a list-element which is displayed in the sub-menu. This SQL query originally selected every literary device "WHERE CATEGORY='clicked category'", but my final design further simplifies this query as every category is in its own table. The query to fetch the list with, say, the literary techniques category, would therefore be no more complicated than: **"SELECT * FROM `category` ORDER BY name"**, where "category" would equal "literary_techniques", which is the name of the table containing all the literary techniques. Every registered literary technique's name would then be displayed in the sub-menu.

Still, even for the top-menu, creating one function for every on-click event would retain the need for five separate functions to react to my buttons, and like I mentioned, doing so for every element in the sub-menu would be both tedious and static. My solution is that I utilize php to set HTML elements based on the data received from the SQL query. Of course, the contents gotten from the query is used for the information that is actually displayed to the user, but it also sets the *class*- and the *id-attributes* using that same information. For example, the element

in the HTML list representing the literary technique by the name of “Allegory” will be displayed by that name in the visual list, but the HTML element will also receive an id corresponding to its name, allegory. The top-menu follows the same principle, except the issue is far less complicated to solve, seeing as the list contents remains the same, corresponding to each table in the database.

The core of the script—which is somewhat clever, if I may be so bold to say—looks like this:

```

27      //-----Sub Menu -----
28
29      $( ".sub" ).click(function() {
30
31          //Change the active sub-menu element to the one that was clicked
32          var tempDom = document.getElementsByClassName('active');
33          var aNode = tempDom[0].className;
34          var activeClass = aNode.replace("active", "");
35          localStorage.setItem("activeClass", activeClass);
36
37          //Use the id of the clicked element to fill and send a form to the server
38          var subject = $(this).attr('id');
39          document.getElementById("currentArticle").value = subject;
40          document.getElementById("articleQuery").submit();
41      });

```

Figure 9: JavaScript Example: Dynamic Menus

This specific code is for the sub-menu, but the top menu follows largely the same principle. The HTML-class attribute is basically used to keep track of what elements in each menu is “active”, meaning it is currently selected and is displaying its contents. For the top-menu, being active would mean that the sub-menu related to that category is active, so selecting the “videos” category would make that element and its sub-menu (the one containing all the video-elements from the database) active.

For the sub-menu, as shown above, when *any* single element within the list is clicked, my JavaScript retrieves the currently “active” element by name, which would be whatever element that is currently having its article/video displayed on the page. The function then takes the classes attributed to that element, and removes the “active” attribute (it also stores the currently active category to retain it after refreshing the page). After this, the currently active element is set to the newly select one, and its HTML-id is retrieved, which as a reminder contains the name of the clicked device. That id is then submitted through a form which is handled by the php/SQL side of things, and the name/id is used to retrieve the article (or video and other content) related to that specific database entry, voila!

The name of the clicked element is also used to display an HTML-header above the article or video retrieved from the database.

The *search*-function also utilizes some JavaScript simply to retain the current state of the site

between submitting the search-form and refreshing the page. The database query for the search function utilizes the **“UNITY”** statement to search/select elements from each of the category tables, and the **“LIKE”** statement to provide search results that are similar, but not necessarily entirely like the given search term. The currently active category is saved, as is the currently active literary device, ensuring the current article/video remains there for the user to see regardless of the page refreshing for the search.

XML

It's regrettable that I didn't reach a point where I could fully integrate XML into every part of my site, because ever since our lectures on XML and the benefits that comes with it, I truly did want to use it in practice for my project to get a good handle of how exactly to use it in a way that lets you reap the benefits from it.

Granted, the contents of my database is simply not the most ideal for the use of XML, and can't quite begin to use it on any more than a basic level unless I were to enforce additional info just for the sake of practice. As mentioned, I did originally want to further divide the contents of each element in my database, but as this also became far less ideal, I was having a harder time to find good ways that I wanted to utilize XML.

One piece of XML-use that I've added that is in fact visible to the user of the web-page itself, is the option to download an XML-file contained the name of the literary device they are currently reading about, as well as the article belonging to it. This XML-file is located on the server, but its contents are dynamically generated based on how the user is navigating the page. Selecting any article at all will alter the XML accordingly, filling in the info for whatever element the user is viewing. Selecting an article will also make the “download”-option appear at the very bottom right-hand side of the page, and clicking it offer a downloadable XML-file with the contents of literary device currently being viewed.

Editing Content

Users

Every person who visits the site is allowed to become a registered user by choosing the “register” option from the top menu bar (or alternatively be directed there through the login screen. Doing so will display the registration-form for the user to fill in.

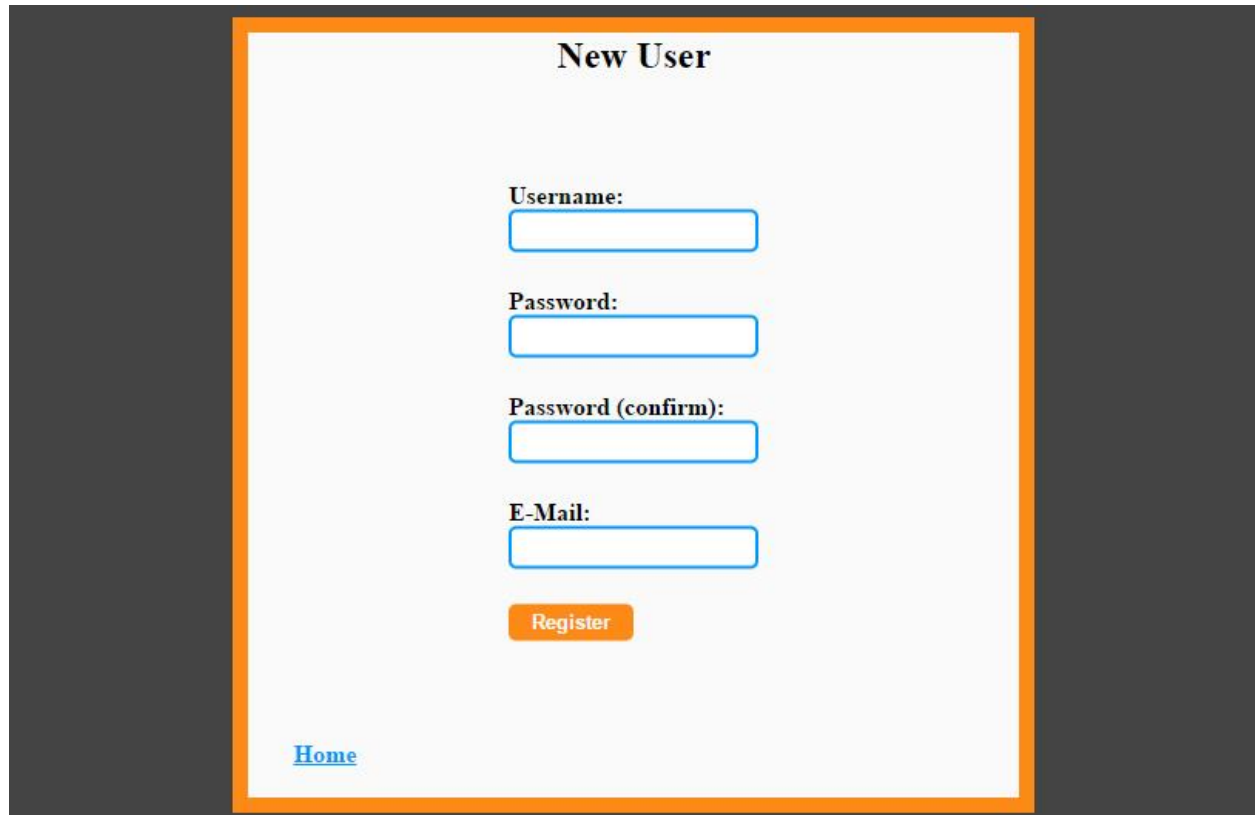
The image shows a web form titled "New User" centered on a light gray background. The form is enclosed in a white box with an orange border. It contains four input fields: "Username:", "Password:", "Password (confirm):", and "E-Mail:", each followed by a white rectangular box with a blue border. Below the input fields is an orange button with the text "Register" in white. At the bottom left of the form, there is a blue link labeled "Home".

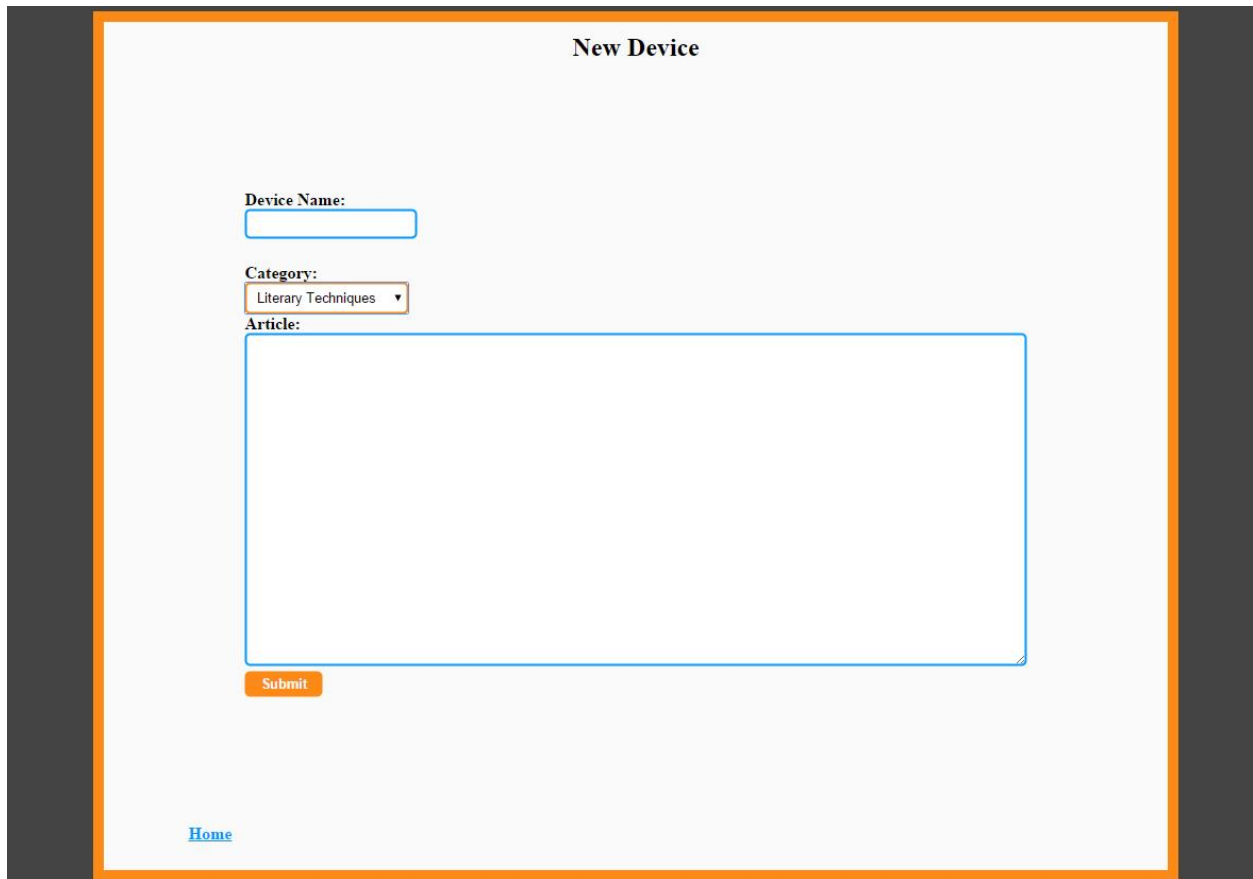
Figure 10: User Registration Form

Becoming a user requires the visitor to enter a unique username, a password, and an e-mail. Their password will be encrypted upon the form being submitted and accepted, and the time of their registration is automatically stored in the database along with their entered information. The e-mail address may also be changed in hindsight, should the user find the desire to do so.

In a part of the database invisible to the user, their editing-rights are set to false by default. Should a user be granted the right to edit however, additional options will appear on the top menu, providing them several possibilities that can be used to alter the content of the database concerned with storing the articles and literary content.

Adding, Editing, & Deleting Content

When a user with editing-rights is logged into the site, additional editing options will be displayed alongside the usual user-options in the top-bar. The option that will always be present in this state will be “Add Device”, which as the name suggests allows the user to add a new literary device, or alternatively a new article on punctuation or a video-focused article. Choosing the option will bring up the “New Device” form:



The screenshot shows a web form titled "New Device". It is set against a light gray background with a dark gray border. The form contains the following elements:

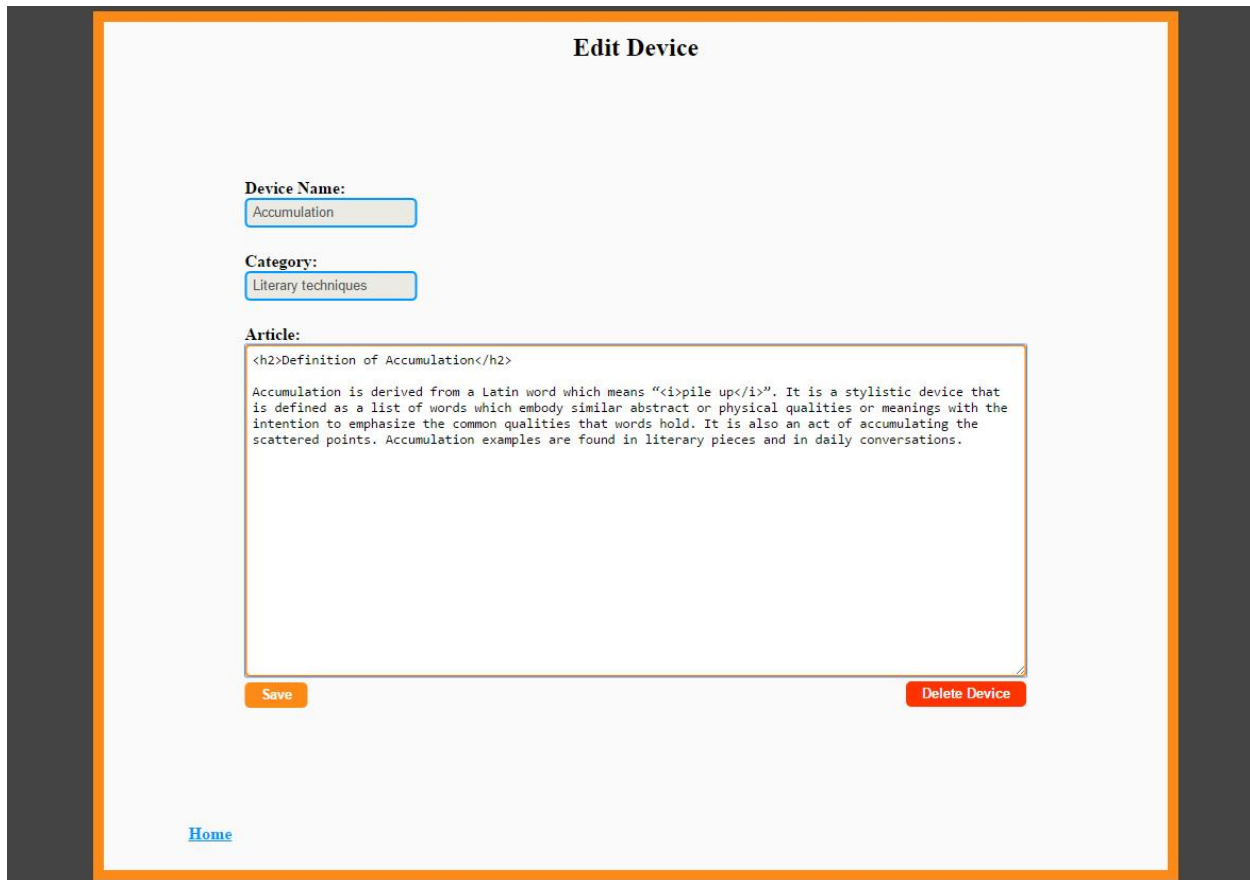
- Device Name:** A text input field.
- Category:** A dropdown menu with "Literary Techniques" selected.
- Article:** A large, empty text area for writing the article.
- Submit:** An orange button located below the text area.
- Home:** A blue link in the bottom left corner of the form area.

Figure 11: New Device/Article Form

When adding a new literary device or other article, the user must first enter a valid subject-name, which will be used to sort the article into the database and display it as an option in its respective menu on the site. What menu/table it belongs to depends on which of the four categories the user selects, which they do using the dropdown menu seen above. Finally, the article must be written as the user sees fit.

If the user is on the main-page and is currently viewing a device, the option to “Edit Device” will also appear in the top menu. This will bring up a similar form, only allowing the user to edit the article of the selected subject/device. This also allows the user to completely remove the device, should they no longer wish to have it in their database.

The device-forms also allows the users to enter HTML-tags for custom-formatting if they'd like to differentiate their text further than the automatic styling of their entered headline. Some other tags on the other hand, such as <div>, is not accepted, and some additional measures are taken when displaying the user-inserted data to ensure user-inserted HTML won't affect surrounding elements.



The screenshot shows a web form titled "Edit Device" with a light gray background and an orange border. The form contains three input fields: "Device Name:" with the value "Accumulation", "Category:" with the value "Literary techniques", and "Article:" with a text area containing HTML code and a paragraph. At the bottom of the form are two buttons: "Save" (orange) and "Delete Device" (red). A "Home" link is visible in the bottom left corner of the page.

Edit Device

Device Name:
Accumulation

Category:
Literary techniques

Article:
<h2>Definition of Accumulation</h2>
Accumulation is derived from a Latin word which means "<i>pile up</i>". It is a stylistic device that is defined as a list of words which embody similar abstract or physical qualities or meanings with the intention to emphasize the common qualities that words hold. It is also an act of accumulating the scattered points. Accumulation examples are found in literary pieces and in daily conversations.

Save Delete Device

[Home](#)

Figure 12: Editing existing devices and articles

The Result

A Dynamic User-Driven Website

The result of my work as I see it is a website that successfully utilizes its database to create a dynamic web-page that allows users to alter its content. All actual content is stored within the database, while the main page located directly on the server is merely an empty shell to be filled with the database's content. The menus themselves only contain the overarching categories that the database content is divided into, and the rest of the menu is generated using the current content of the database.

The JavaScript does its part using HTML-classed and id's generated using the names of the database content as well, and through the interface, users may select the subject they want to view in order to have it displayed in the content pane. Furthermore, it allows users to keep browsing the menus while retaining their current content on display, and the search-function allows users to search all the content-tables in the database to find specific devices without browsing through the menus.

With their user account registered and editor rights attained, the users may freely use and update the site as they see fit, with no assistance required, and no changes having to be made to code, nor to the database externally aside from the forms that are integrated into the site. The exception remains granting a user right to edit content. This is something I did as a safety measure. You obviously don't want anybody to be able to mess with the contents of your site, and the sure-safe way to prevent rights to wind up with the wrong people is to require the change to be made directly to the database. Granted, you could enable existing users with editing-rights to grant that right to others as well, but such a system may also quickly spiral out of control. The very best option might have been to have a selected few "super-users" or administrator accounts of some kind who could both monitor other users and grant or take away the right to edit content.

As for other lacks and improvements, further customizing and dividing up the tables in the database would be my first step to improve the site's modularity, and then to more effectively divide each property into XML-nodes and display them, visually much in the same way I already do. The way my forms function worked out better than I had initially thought, but having users utilize HTML tags is not the most ideal way to edit content. I've experienced pages with very large user-bases utilize the same concept, and it provides the user's with a lot of choice, but in the end I believe the best solution would be to create a full-fledged text-editor box that could replace the current HTML-text field, but I digress.

Ultimately I'm happy with the way my sites function overall, and I find it fulfills the general needs of the assignment satisfactory, albeit I would have liked to further integrate XML into my page and have its design be more considerate to ever-growing infrastructure of the internet, if not quite to the point of using Hadoop.

References & Resources

Styling & CSS

I utilize a few of Bootstrap's built-in features for my styling in order to style my menus in a tidy way.

Admittedly I hardly even scratch the surface of all of its features, but I found it suited my purpose neatly:

<http://getbootstrap.com/> (last visited 23/3/2015)

The template and styling for my topmost menu can be found here:

<http://cssmenu maker.com/menu/responsive-menu-bar> (last visited 23/3/2015)