We began project development with work on the back end. After discussing the process, we came to the decision that building the system from the ground up—from the structure of our database, to building the interfaces between the database and the presentation of the information within, to the actual display and layout of the pages—would allow for easier integration and ease of development as we moved forward with each phase of development.

The first step was to decide the structure of our database. Using MySql, we came up with a database consisting of five tables: User, containing the values ID, username, password, first name, and last name; Calendar, containing the values ID, user id, name, and visibility; Calendar Item, containing the values ID, calendar ID, name, description, location, and time; List, containing the values ID, user id, name, and description; and List Item, containing the values ID, list id, and description. Certain values of these tables can be used to trace and create a tree structure of elements for ease of organization. For instance, if a User creates a calendar, the user ID is stored as a value in the calendar table. From there, the user will be able to create events (calendar items) under that calendar. The ID of the calendar is stored as a value in the calendar items table. With this information, to display a calendar, we only have to select the id’s of the calendars where the user id corresponds to the id of the user, and then select the calendar items that correspond to the id’s of our selected calendars. Technical language aside, we believe this structure will be enough to accomplish our goal of implementing a system that allows for a compiled method of organization for the users. If not, further structure of the core systems will allow for easy manipulation, should we need to add or remove any components.

The next step was to create classes using PHP to facilitate the storage and organization of data within the system. All tables in the database use auto-incrementing integer values as their ID’s, so with this exception, each class corresponds exactly to their table within the database. Each class consists of getters and setters for each member value, as well as appropriate constructors. The classes themselves do not interact with the database (i.e., no User.store() methods to store a user into the database), but they do implement an interface called persistable, indicating that they represent objects that are meant to interact, through some form, with a database. The interface consists of three methods dealing with manipulation of object ID’s. In short, if an object has NOT been stored to a database, its ID is set to -1, while if it has been stored, its ID corresponds to its entry in the database.

From here, our task was to implement a system that interfaced between the classes and the database. The purpose of this system layer is to create a level of abstraction between the actual classes and the technicalities that are involved in storing, deleting, and retrieving from the database. In other words, we separated the logic and language (sql statements, for example) from the actual representations of the objects we are using in our system. Called the “Persistence Layer,” this script is responsible for the actual storing and retrieving of database information, and contains separate store, retrieve, and delete functions for each of our five object types. In addition to the abstraction, this creates a fairly intuitive usage environment—all that needs to be done to add an object to a table is to call the corresponding object function from the persistence layer class.

These systems define the core of our application. With this completed, we could move on to designing an interface to interact with and manipulate the data. It was decided that we would base the entire interface around a single html file with a single JavaScript file that defined the functionality of the interface (there are, however, several other JavaScript files linked in the main html file—these are classes used to define the objects used in the system; they are more or less exactly the same as their PHP counterparts). The reason for this centrality was to support one of our main goals: to keep everything in one place. A single page containing all the information a user needs eliminates the overhead of having to navigate between web pages to accomplish tasks. With our design, everything, from object creation, to modification, to viewing is kept easily accessible and right in front of the user through their whole experience.

The first challenge was figuring out how to commute our data from the PHP persistence layer to out JavaScript controller. The solution ended up being a div element with an ID of ‘dom-target.’ This div runs a script upon loading of the page that retrieves all necessary data from the database, then encodes it via JSON and echoes it in a paragraph element to the page. The display type is ‘hidden,’ so these elements are not seen and are only accessed by the JavaScript controller, which extracts the information, parses it via JSON, then creates corresponding objects and stores them in global arrays for use when necessary. The advantage here is, again, all the information is centralized, and is only passed once per page load. It also keeps stray calls for information from appearing in unnecessary places—i.e., after a database update occurs, we don’t need an update\_data function or anything of that sort since the data is reloaded upon reloading the web page anyway. This structure also provided a means of having to worry about populating our data—once it happens, it’s all available and accessible in the JavaScript controller, and the arrays containing that information are free to be utilized without having to make calls to update or change information through the database.

Of the elements central to the html file, there are a few that are central to the functionality of the interface. First, of course, there’s the elements that make up the weather icon. This display pulls information from the weatherunderground API to populate the information it uses to display the weather. Having the weather as a component was a central idea of our design concept—not being prepared for the weather on a daily basis can make or break a day. If it’s raining, you may think to leave an extra few minutes early to account for the inevitable traffic. If a cold front came through, and now it’s 40 degrees in the morning instead of 70, you would want to know so you can bring a jacket with you. If some light showers came through and the temperatures were below 32 degrees, you would want to know so that you could be cautious of ice on the roads. Weather awareness is as central to an organized life as having an organized calendar and organized list of tasks.

The other important element is the dashboard. It consists of a toolbar at the top, containing lists of calendars and lists, and then a large, empty area that makes up the majority of the display. This area is the area targeted by the JavaScript controller when most events occur. Listeners are attached to every element in the calendar and list lists that prompt creation of elements via DOM operations when they’re clicked. In short, for either of the two objects that can be displayed, a parent element is created and added to the DOM as a child to the dashboard, and other children are added to that parent elements from there. The resulting element displays the object that was the target of the click—a calendar, or a list. These are entirely new elements that do not appear anywhere in the html file—their creation and deletion is handled entirely through the JavaScript controller. This allows maximum flexibility, as every element is created with its own unique ID based on the type of the object and the actual ID of the object. This keeps us from having to keep track of existing elements or ID’s for elements and greatly simplifies the understanding of the code behind the display. If we want to delete an object, we simply have to get the children of the dashboard, search for the corresponding ID, and remove it from the DOM.

As far as functionality of the calendars and lists themselves, they maintain everything you would except from a calendar or a list. For calendars, you can create new calendars, add and delete events, and edit existing events. You can even display events as a weekly calendar if you’re a little more “now” focused and don’t care to look at an entire 30 days at a time. For lists, again, you have expected functionality: create new lists, and add or remove list elements. These are all persistent database objects, of course, and handling of the database information is dealt with through forms and calls to separate PHP files for each action. In these cases, data passing is simply handled through POST methods. If the database operations are successful, the app directs the user back to the main page, which results in a call to obtain the manipulated information from the database, as discussed above.

With all this in place, we concluded the development of our centralized organization application. We truly believe that we have created an application that streamlines the method of daily organization. Your calendars, your task lists, and your weather are all centralized in one simple, easy to use application, and with the technology available today, this application could be available to the user essentially anywhere. The alternative to our system would be a secretary, perhaps, but computer systems are not subject to the same inconsistencies as humans, and you won’t have to pay our system to keep track of your life for you. Now, all of your daily planning is right in front of you, and you can be ready for whatever life may throw at you next.