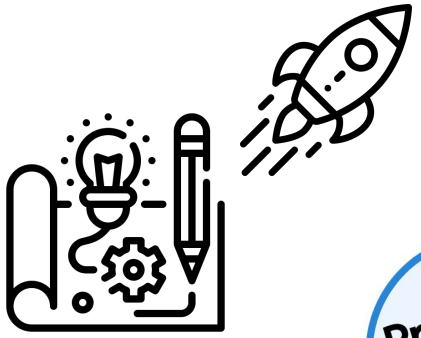
CMS Website Builder

Identity Design Evolution



Project Wixs

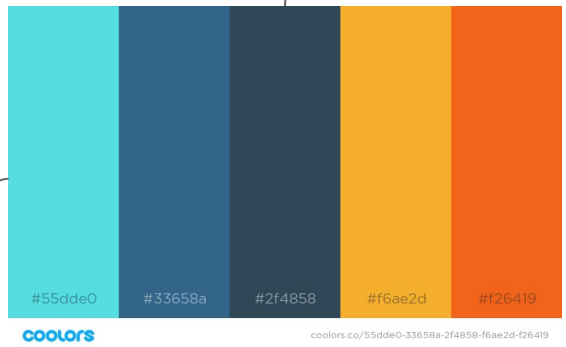
Project Wixs

Project Wixs

Project Wixs

Project Wixs

Project Wixs



Initial Tech Stack

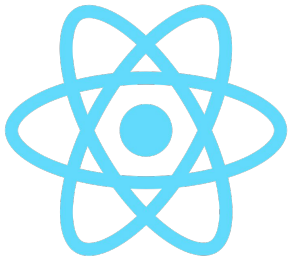
PHP - Server Scripting

PostgreSQL - Relational Database

PostgreSQL

React.js - Responsive front-end

- JavaScript
- HTML
- SASS/CSS



Node.js - Development server | Dependency Management

- NPM

Parcel.js - Development Hot reload | Production Bundling

Team Role Management

Drayton Williams: Team Lead | Front-End | Back-End

Jasdeep Grewal: Back-End - PHP Scripting

Cameron Hammel: Front-End - Dashboard Operations

Ian LeMasters: Front-End - Component Testing | System Walkthrough - Usage Guide

Nathan Hellinga: Front-End - Lead Editor Design & Functionality

Liam Howes: Front-End - Admin operations | Back-End - Database Administration

Kieran Colaco: Front-End - Editor Component Creation

Curtis Honsberger: Front-End - CSS Styling & Editor Design

Each member actively participated in Documentation

Internal Documentation

- We created internal documentation to help set up standard practices and keep the group organized

ProjectWixs Roles

We are likely to bounce between all these roles so nothing is set in stone but if we can have people who can really 'master' some role(s) and become a "go-to" guy for a role, that will make you the most invaluable.

Front-End:
- Knowing general structure of React and its addons (React-Router, React-Bootstrap, Parcel.js)
- HTML, CSS, and JavaScript: all required to fully utilize React so this is a given.
- Mainly dealing with the .html .js .scss files

Back-End:
- Postgres database development and administration
- PHP (and how to communicate with Postgres)
    - The creation of webpages displaying info from Postgres database relevant to front end developer (returning a 1 if user was authenticated or returning the list of users for the admin page)
- Mainly dealing with the .php files and basically creating REST API (based on above point)

Documentation and Structure (The clean-up crew)
- Organizing the GitHub based on versioning, releases, and issues – keep it clean and organized so the devs can easily pull and get the right files. Know how GitHub and or Git works
- Active code commenting: Because there is never too much commenting when trying to explain how functions/files work. Also maintaining that each file follows a similar structure. Make shit pretty – basically.

Testing People
- This is a fluid role that we all actively participate in
- Whether testing certain features so they fall within our set parameters (password length, max/min number of users, exception handling). This is all crucial.
- Finding and using a test suite (e.g. Jest -- a JavaScript testing framework, works with React and other JS frameworks)

Most Importantly – Template Functionality
- This will likely combine a few of the above-mentioned roles so this is an important section. Will need at least 3 of us dedicated to really understanding how we are going to implement the template building/publishing aspect.
- Will require React.js and PHP and Postgres knowledge and interaction

We will add roles and things to do as we go of course but hopefully this helps as a reference point beyond any chats we have.

How to run the current system locally
Monday, Feb 17th, 2020

1. Download and install Node.js and Npm. Found Here: https://nodejs.org/en/
2. Download/Clone GitHub repo and locate wixs-tester folder. Found Here: https://github.com/DrayWilliams1/ProjectWixs
3. Cut/Paste 'wixs-tester' folder into somewhere easily accessible. Root folder is recommended.
4. Open terminal or command line and 'cd' into wixs-tester folder.
5. Enter 'npm install' and it should load/build up a folder called "node_modules"
6. Enter 'npm run-script dev' and it should bundle up the files (through Parcel) and load it onto localhost: 1234. Go to that link.
7. Profit.
8. Bonus: You can have your own method of accessing/manipulating the files, but I use VSCode (right now) and have a couple extensions that make it nicer to deal with – ask me if you want to know more). Remember it's all good because it's your local test server

If any of this doesn't work let me know. This worked for me.

How to get into Postgres

Group Account Details
Username: c4f00g02
Pass: A9b3f(Z2V$

There's 2 ways I get into Postgres:

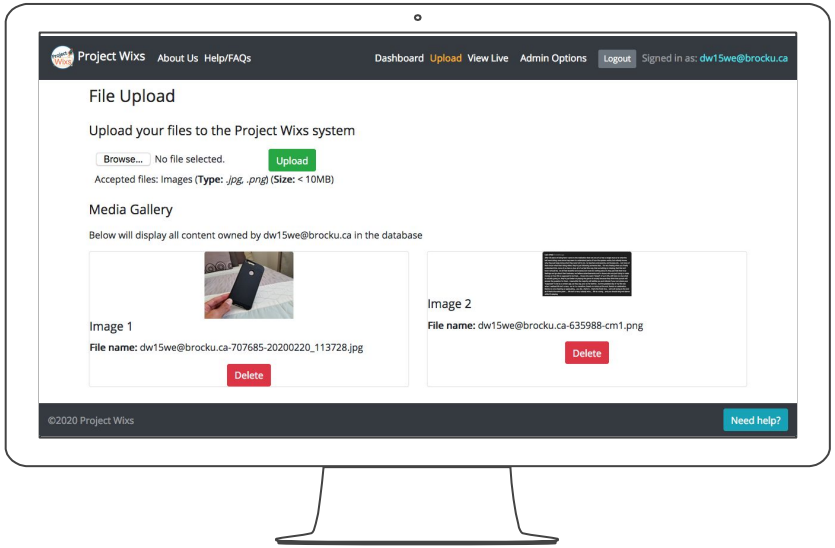
1 – Through MobaXterm
- Ssh into sandcastle.cosc.brocku.ca but with our username and password combo
- Enter 'psql' into command line and you will enter our Postgres database. Here you can type help to find a bunch of the functions or just search them online. I usually use \l to see if the tables I created are there. I would suggest trying some of the commands at this link: https://gist.github.com/Kartones/dd3ff5ec5ea238d4c546

2 – Through PgAdmin
- I used PgAdmin to have a better visualization of our database. Download Postgres software here: https://www.pgadmin.org/download/
- Left click on 'Servers' on the left hand menu and Create → Server.
    - Under General: Set Name as whatever (I used "Brock Sandcastle")
    - Under Connection: Set Host name as sandcastle.cosc.brocku.ca. Set username as ours and password as ours
    - Under SSH Tunnel: Enable it first. Set the host as what I said before, username as ours, and password as ours.
- When you hit save it should connect you to the server. You'll find ours if you go to the databases field on the left menu and it will be a big list with all the databases in the server (and yes you can look at the other groups/individuals databases – I already saw them)
- To find and edit our tables go to Schemas → Tables and right click for more options.

Other than the above steps I would say just play around because I am new as well and I am sure there's features/software I haven't explored yet

Other helpful links:
https://www.postgresql.org/docs/9.1/sql-commands.html - postgres sql commands
https://www.postgresql.org/docs/9.2/app-psql.html -- psql commands
https://www.zentut.com/php-pdo/pdo-connecting-to-postgresql/ -- initial connection
https://www.postgresqltutorial.com/postgresql-data-types/ -- postgres data types

Front End - Responsive Components



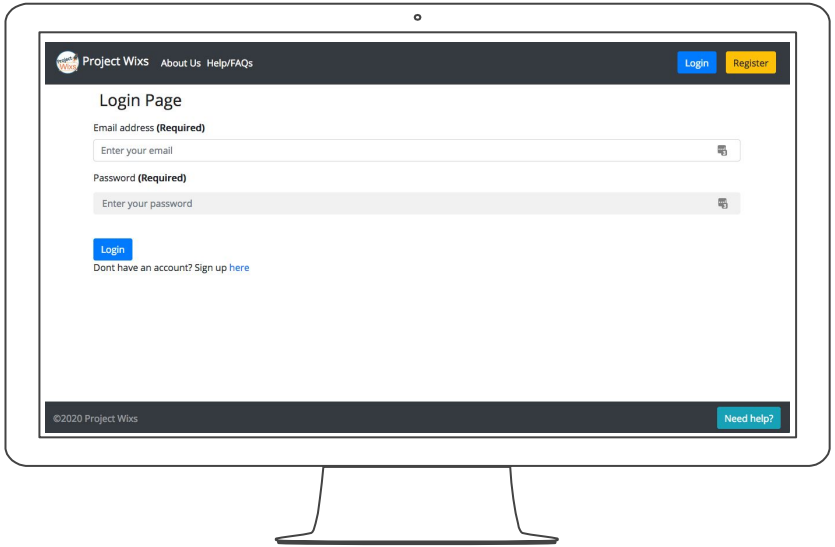
- Basic render function
- State and props
- Responsive ‘reactive’ data

```
async onSubmit(event) {
   event.preventDefault();
   if (this.state.selectedFile) {
     this.uploadFile(this.state.selectedFile);
   } else { // no file selected
     alert("No file selected. Try again.");
   }
 }

render() {
   return (
     <div>
       <Container>
         <h1>File Upload</h1>
         <h2>Upload your files to the Project Wixs
system</h2>
       </Container>
         ....
     <div>
   );
 }
}
```

```
const params = {
      email: this.state.email,
      password: this.state.password,
      usid: this.state.usid
    };

    axios
      .post(LOGIN_USER_URL, qs.stringify(params))
      .then(response => {
        console.log(response);

        if (response.data["success"] === true) {
          auth.setCookie("usid", this.state.usid, 7);
          auth.setCookie("user", this.state.email, 7);

          window.alert("Sign in successful.");

          window.location.href = "#/dashboard";

        } else {
          window.alert(response.data["message"]);
        }
      })
      .catch(error => {
        console.log(error);
      });
```

Front End - Back end Transfer

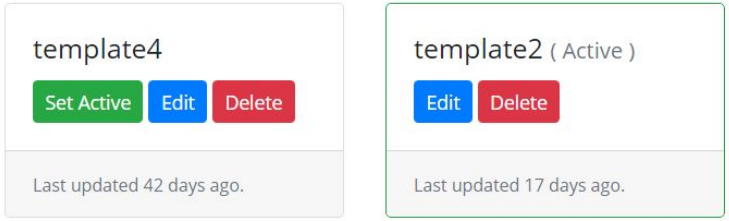
- Web development needs AJAX
- React uses Axios
- Most components follow this

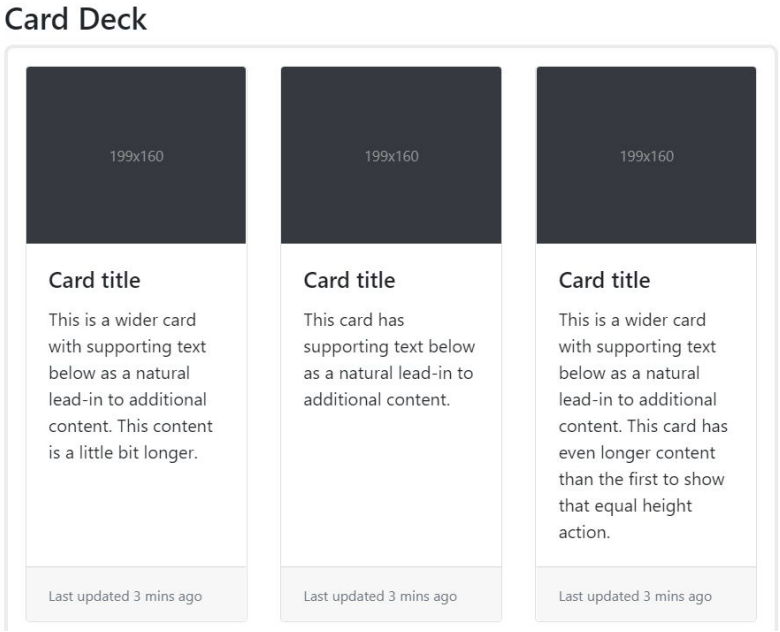


Front End - Template Management

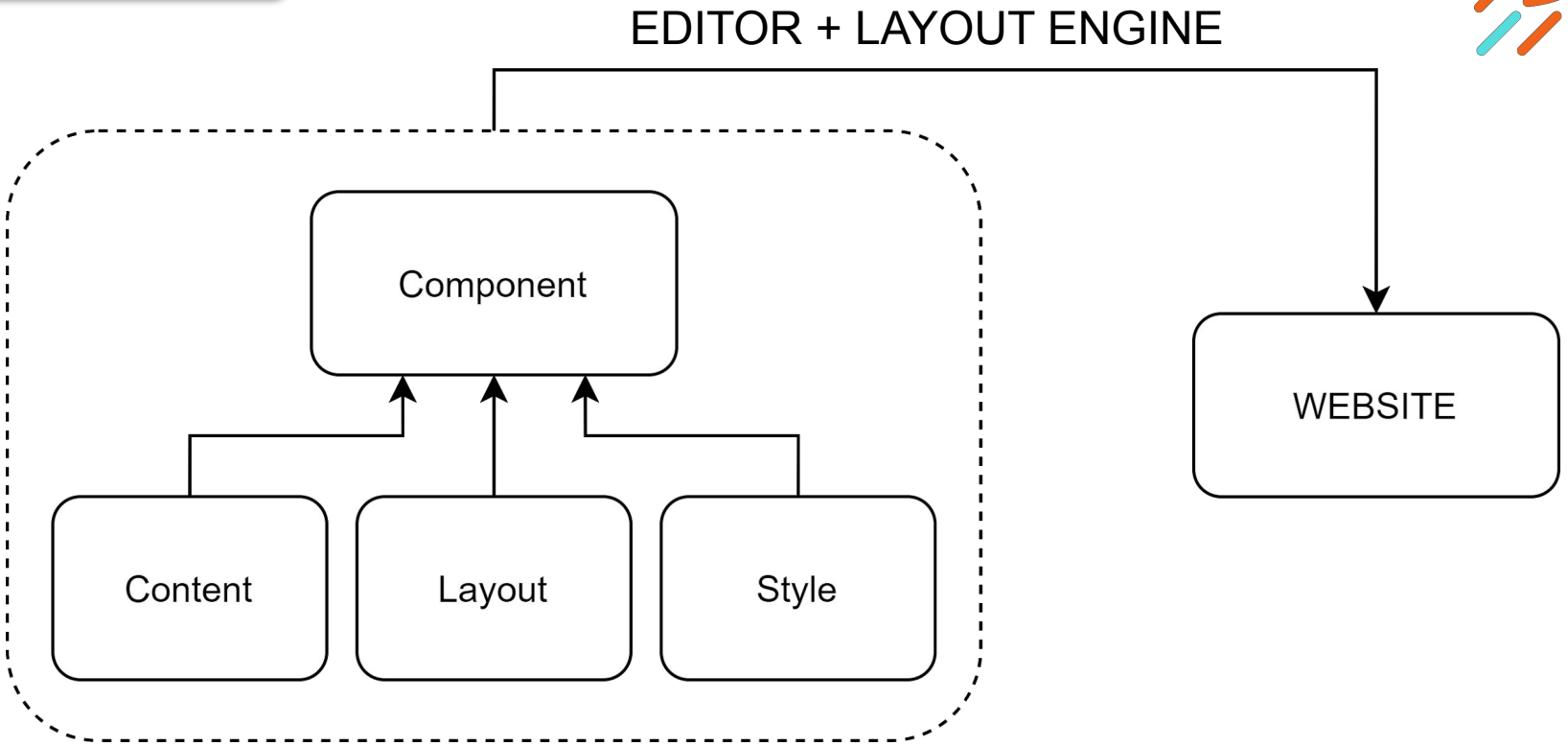
React-Bootstrap - Components

- Card deck
- Cards
- Buttons
- Custom “Active” Card





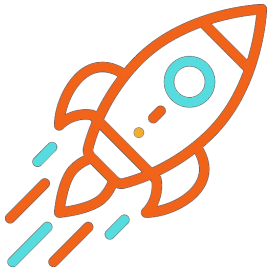
Editor


EDITOR + LAYOUT ENGINE
Component
Content
Layout
Style
WEBSITE

Editor - Components

- Individual self contained page elements
- Isolated components prevents hard to track down bugs
- Allows for quick/easy development of additional components
- Component code is entirely separate from editor code

Editor - Components



- Each component can be parsed as a JSON object which allows for the editor to convert it’s entire state to an immutable string
  - Base Component
  - Custom Props
  - Inline Styles
- Implements/provides a data schema that allows the editor to build custom forms for data manipulation

Editor - Layout Engine

- The system that supports resizing, dragging and dropping components in the editor.
- This system is integrated with the editor to also provide component properties which allow adding resize limits.

Editor - Engine

- Uses a custom data schema to allow all custom implementations of any component
- Creates forms on the fly from component schemas
- Applies custom component styles inline
- Handles saving and loading layout
  - Saving happens automatically whenever a page change is made. This ensures the user does not forget to save changes and is a simple addition because of our simple JSON system save states
- A stripped down version of the editor is what renders the JSON data for the published page

Editor - Component Schema

- Provides a simple and extensible interface for the editor and components to interface
- Components do not need to provide any editor code
- Reduces code reuse in components
- Compartmentalizes functional components to allow parallel development

```
export const SCHEMA = {
 type: RichTextBox,
 gridOptions: {h: 2, w: 4, minW: 2, minH: 2},      //
grid options parameters: minW, maxW, minH, maxH,
isDraggable, isResizable
 title: "Rich Text Area",
 iconName: "plus.svg",
 desc: "A Text Component that Allows for Text Styling
such as Bold, Underline, Bullet Points, etc",
 iconPathName:
require('../../../assets/icons/other/011-lines.svg'),
 props: {
   content: {
     type: "RichText",
     name: "Content",
     value: null,
   }
 }
};
```

Back End - Postgres connect

- Credentials are obtained and connection is established

```php
header("Access-Control-Allow-Origin: *");
require_once 'dbConfig.php';
$dsn = "pgsql:host=$host;port=$port;dbname=$db;user=$username;password=$password";
try {
   $pdo = new PDO($dsn); // create a PostgreSQL database connection

   if($pdo) // connected successfully
      { /* execute logic here */ }
} catch (PDOException $e) {
   $responseObject['message']=$e->getMessage(); // report error message
}
```

Back End - Data Binding

- Necessary values are collected from POST request and locally binded to be used

```
if($pdo) { // connected successfully
  if (($_SERVER['REQUEST_METHOD'] == 'POST')) {
            // Using empty test instead of isset function
            $email_post = empty($_POST['email']) ? null : $_POST['email']; // set email to form submission
            $password_post = empty($_POST['password']) ? null : $_POST['password']; // set password name to form submission
            $sessionID_post = empty($_POST['usid']) ? null : $_POST['usid']; // retrieve session id

            if (validInputs() && accountExists() && updateSession()) {
                $responseObject['success']=true; // echoing a response that can be used to redirect page after AJAX call
            } // otherwise, error, response message is displayed in alert
        } else { // request method is not POST
            $responseObject['message']="Invalid request. ";
        }
  }
}
```

Back End - Query Execution

- Values are then bound to a SQL query and executed.

```
function getUser() {
    global $pdo, $email_post, $responseObject, $user;
    $sql_select = "SELECT user_id, email, first_name, last_name, admin FROM users WHERE email = ?";
    $stmt = $pdo->prepare($sql_select);
    $stmt->bindValue(1, $email_post, PDO::PARAM_STR); // pass and bind value to the statement

    if($stmt->execute()) { // The query has executed successfully
        if ($stmt->rowCount() > 0) {
            $user = $stmt->fetch(PDO::FETCH_ASSOC);
            $responseObject['user']=$user;
            return true;
        } else {
            $responseObject['message']="User with email {$email_post} does not exist in database. ";
            return false;
        } ...
    } else {
        $responseObject['message']="Error querying users table. ";
    }
}
```

Overall Functionality

Make/Login Account → Create/Manage Templates → Upload/Manage Content → Edit Template (w/ Content) → Publish Template → Share Site

System Limitations/Future Improvements

JEST - JavaScript testing framework (for React components)

Stress Testing Server - for more accurate limits
- Videos (outside hosting)

Multi-paged routing (templates)

Login Tests/Challenges

Improved Alerts/Toast(y)