CS Capstone Final Document

May 25, 2018

Linking Seasonal Weather Data to AgBizClimate^{TM}

Prepared for

Oregon State University

Clark Seavert

Signature Date

Prepared by

Group26

Signature Date

Abstract

The purpose of this document is to provide documentation regarding the *AgBizClimate* project. We will start off the document by giving a general overview of the *AgBizClimate Project.* This will inclue information about the goals of the project, and information about the project stakeholders. Next we have our requirements document. In this section we will also discuss how our requirements have changed over the last several terms. Next we will be discussing the design document. In this section we will first display our original design document. We will then discuss how our design has changed over the term. After the design document we will display the tech review. Next, we will also display the project poster. Finally we will provide some project documentation regarding how project setup, running the project, how the project works and guids for any API's we are using. After this we will discuss any technical resources for learning more about the technologies that our project uses. Finally, We will end this document with our conclusions and reflections section. This section will involve reflecting on this project and discussing what went well and what didn't go well.

Contents

1 Introduction

1.1 Purpose

This SRS describes the requirements and specifications of the AgBizClimate^{TM}web application. This document will explain the functional features of this web application. This includes the interface details, design constraints and considerations such as performance characteristics. This SRS is intended to outline how we will proceed with the development of the *AgBizClimate* system.

1.2 Scope

This project is a part of a much larger AgBiz Logic^{TM}program. However, the purpose of this project is to add a short term climate tool to the *AgBizClimate* module. This limits the scope of the project to the *AgBizClimate* Module. Additionally, we will only be adding the short term climate data tool as the long term climate data tool already exists.

1.3 Definitions, Acronyms and Abbreviations

REST - Representation State Transfer, This is a type of architecture that manages preforms operations on the state of the program. This is especially popular in web development.
API- Application Programming Interface. This is a piece of software that allows a connection to another piece of software providing some sort of service.
Thredds Data Server - This is a web server that provides meta-data and data access for scientific data sets using OPeNDAP along with some other remote data access protocols.
OPeNDAP - Open-source Project for a Network Data Access Protocol. This is the protocol we will be using to retrieve the data sets from the Thredds data server.
NMME - North American Multi-Model Ensemble. This is a data set that brings together a variety of different weather models into one data set.
Climate Scenario - This is a theoretical calculation of yields, inputs and of the overall budget for one situation based on the climate data.
NETCDF - This is a file storage format for large scientific data sets especially good for any data that is referenced on a grid and related to is geo-location.
SQL Database - This is a relational database that makes storing and accessing data easy.
Container - A virtualized operating system that is used to host and deploy web development projects. This allows projects to be easily portable between different operating systems and platforms.
Mount Bind - This is the practice of mounting a directory from the native os to the container such that if either the container or the native operating system. This allows changes in the files to be reflected in both the container and the native operating system.

1.4 References

1.5 Overview

Seasonal climate is one of the essential factors that affects agricultural production. As a module of *AgBiz Logic*, *AgBizClimate* delivers essential information about climate change to farmers, and help professionals to develop management pathways that best fit their operations under a changing climate. This project aims to link the seasonal climate data from the NMME data set to *AgBiz Logic* so that it can provide changes in net returns of crop and livestock enterprises through powerful graphics and tables.
Currently *AgBizClimate* has a long-term climate tool but no such tool exists for short term climate data. We will implement a tool to extract short-term climate data from the NMME data set, display it to the user and allow the user to adjust crop and livestock yields or quality of products sold and, production inputs. Moreover, a landing tool

will be developed to allow users to switch between short-term seasonal tool and long-term climate data tool.

1.6 Product Functions

AgBizClimate is a web based decision tool that will allow users to gain specific insight on how localized climate data for the next seven months will affect their crop and livestock yields or quality of products sold and production inputs. The *AgBizClimate* tool will allow users to input their location (state, county) and a budget for the specific crop or livestock enterprise. *AgBizClimate* will select climate data for the next seven months for that location and provide graphical data showing temperature and precipitation. Users will then be able to change yields or quality of product sold by a percentage they think these factors will affect and modify production inputs. Finally the tool will calculate the net returns.

1.7 User Characteristics

AgBizClimate users can be split into two subgroups, agricultural producers and climate researchers. The first subgroup, agricultural users who use this product tend to be between fifty and sixty years old of mixed gender. Their educational background ranges from high school to the completion of college. The primary language this group uses is English, but there are some Spanish users as well. Most of the users in this group tend to have novice computational skills. The primary domain for these users is agricultural and business management. Most agricultural producers who use this product are motivated by the potential profit that the decision tool *AgBizClimate* could potentially offer. The second subgroup, climate researchers range from ages twenty to forty and are of mixed gender. The educational background for most climate researchers exceed the postgraduate level with their primary language being English. These users generally have advanced computational skills and are motivated by the easily accessible climate and weather data.

1.8 Constraints

There are several key constraints that this product has to work within. Firstly, We are limited by the availability and completeness of the data from the NMME data set. The NMME Data set does not have data for every point on in the country. Secondly, we must use the Threadds server hosted by University of Idaho to get the data in the NetCDF format. The tooling provides a variety of access methods however, the only method that currently works is downloading the whole file. Fourthly, we dont have access to any of the hardware that *AgBizClimate* is exists on as it is being managed by a third party. This will prevent us from improving the hardware or cause roadblocks if their servers are having issues. Lastly, we are limited to using the languages Python and JavaScript since we are integrating our product into an already existing project.

1.9 Assumptions and Dependencies

We are assuming that the NMME Thredds data base will allow us to pull location based temperature and precipitation data. This data will come in the form of a NetCDF files which we will then read and format a JSON response. Due to the fact that we are writing an addition to an existing project we do not need to interact with the user budgets as these have already been defined. This fact extends to the calculations portion of the *AgBizClimate* product. Our team will simply be accessing data via the NMME threadds database, will then format the data, store the data, and hand the data over to the front end or some other sort of client.

2 Requirements Document

2.1 Original Document

CS Capstone Software Requirements Document

November 3, 2017

Linking Seasonal Weather Data to AgBizClimate$^{\mathrm{TM}}$

Prepared for

Oregon State University

Clark Seavert ______________________ __________
Signature *Date*

Prepared by

Group26

Thomas Noelcke ______________________ __________
Signature *Date*

Shane Barrantes ______________________ __________
Signature *Date*

Shengpei Yuan ______________________ __________
Signature *Date*

Abstract

This Software requirements document will cover the requirements for the AgBizClimate project. We will first give a general introduction of the project. This section will provide some context for why we are doing this project and what this project hopes to accomplish. Next will give an overall description of what our application does. This section will what the application does in broad no technical terms. Then we will discuss specific Requirements. This section will provide the technical details of the application including the functional requirements for the application. In the next section there is an approximate project schedule displayed in a Gantt chart. Finally, The last section provides the Tables and Figures referenced throughout the document.

Contents

List of Figures

1 Introduction

1.1 Purpose

This SRS describes the requirements and specifications of the AgBizClimate^{TM}web application. This document will explain the functional features of this web application. This includes the interface details, design constraints and considerations such as performance characteristics. This SRS is intended out outline how we will proceed with the development of the *AgBizClimate* system.

1.2 Scope

This project is a part of a much larger AgBiz Logic^{TM}program. However, the purpose of this project is to add a short term climate tool to the *AgBizClimate* module. This limits the scope of the project to the *AgBizClimate* Module. Additionally, we will only be adding the short term climate data tool as the long term climate data tool already exists.

1.3 Definitions, Acronyms and Abbreviations

REST - Representaional State Transfer, This is a type of architecture that manages the state of the program. This is esspecially populare in web development.
API- Application Programming Interface. This is a peice of softwared that allows a connection to another peice of software providing some sort of service.
NWCTB - Northwest Climate Toolbox. This is the database we will be connecting to that will provide the short term climate data we plan to use.
Climate Scenario - This is a theoretical calculation of yields, inputs and of the overall budget for one situation based on the climate data.
SQL Database - This is a relational database that makes storing and accessing data easy.

1.4 References

[1] C. F. Seavert, "Negotiating new lease arrangements with the transition to direct seed intensive cropping systems," 2017.

[2] S. Y. Thomas Noelcke, Shane Barrantes, "Problem statement," 2017.

1.5 Overview

Seasonal climate is one of the essential factors that affects agricultural production. As a module of *AgBiz Logic*, *AgBizClimate* delivers essential information about climate change to farmers, and help professionals to develop management pathways that best fit their operations under a changing climate. This project aims to link the crucial seasonal climate data from the Northwest Climate Toolbox database to *AgBiz Logic* so that it can provide changes in net returns of crop and livestock enterprises through powerful graphics and tables.
Currently *AgBizClimate* has a long-term climate tool but no such tool exists for short term climate data. We will implement a tool to extract short-term climate data from the Northwest Climate Toolbox database, display it to the user and allow the user to adjust crop and livestock yields or quality of products sold and, production inputs. Moreover, a landing tool will be developed to allow users to switch between short-term seasonal tool and long-term climate data tool.

2 Overall Description

2.1 Product Functions

AgBizClimate is a web based decision tool that will allow users to gain specific insight on how localized climate data for the next seven months will affect their crop and livestock yields or quality of products sold and production inputs. The *AgBizClimate* tool will allow users to input their location (state, county) and a budget for the specific crop or livestock enterprise. *AgBizClimate* will select climate data for the next seven months for that location and provide graphical data showing temperature and precipitation. Users will then be able to change yields or quality of product sold by a percentage they think these factors will affect and modify production inputs. Finally the tool will calculate the net returns.

2.2 User Characteristics

AgBizClimate users can be split into two subgroups, agricultural producers and climate researchers. The first subgroup, agricultural users who use this product tend to be between fifty and sixty years old of mixed gender. Their educational background ranges from high school to the completion of college. The primary language this group uses is English, but there are some Spanish users as well. Most of the users in this group tend to have novice computational skills. The primary domain for these users is agricultural and business management. Most agricultural producers who use this product are motivated by the potential profit that the decision tool *AgBizClimate* could potentially offer. The second subgroup, climate researchers range from ages twenty to forty and are of mixed gender. The educational background for most climate researchers exceed the postgraduate level with their primary language being English. These users generally have advanced computational skills and are motivated by the easily accessible climate and weather data.

2.3 Constraints

There are several key constraints that this product has to work within. The first constraint is that we only have access to two data parameters from the North West Climate Tool box, precipitation and temperature. Secondly, we only have access to their data via the NWCTB API which could have additional restrictions such as limited usage per day, mislabeled data, or poor documentation. Thirdly, we dont have access to any of the hardware that *AgBizClimate* is exists on as it is being managed by a third party. This will prevent us from improving the hardware or cause roadblocks if their servers are having issues. Lastly, we are limited to using the languages Python and JavaScript since we are integrating our product into an already existing project.

2.4 Assumptions and Dependencies

We are assuming that the Northwest Climate Toolbox is a functional API that will allow us to pull location based temperature and precipitation data. This data will most likely come in the form of a text body of which we will then format into a JSON object and store in a Mongo database for future use. Due to the fact that we are writing an addition to an existing project we do not need to interact with the user budgets as these have already been defined. This fact extends to the calculations portion of the *AgBizClimate* product. Our team will simply be accessing data via the NWCT API, then format the data, store the data, and hand the data over to the tool while will provide some additional front end support.

3 Specific Requirements

This section contains all of the functional and quality requirements of the *AgBizClimate* System. We will give detailed description of what's being added to the *AgBiz Logic* system along with the features we will implement.

3.1 External Interface Requirements

This section provides a detailed description of all inputs into and outputs from the short term climate tool for the *AgBizClimate* system. This section will also provide descriptions for the hardware, software and communication interfaces. Additionally we will provide detailed prototypes for the user interface for the short term climate data tool.

3.1.1 User Interface

When the user first navigates to *AgBizClimate* from the *AgBiz Logic* main page the user will be directed to a landing page. This landing page will allow the user to either select the existing long term climate tool or the short term climate tool that we will be developing. On this page there will be a brief description of the tool and what does. This description will also describe the difference between the long term climate data tool and the short term data climate tool. Below this description will be two buttons one to run the long term climate data tool and one to run the short term climate data tool. Clicking the long term climate data tool button will take you to the long term climate data tool page. Clicking the short term climate data tool button will take you to the short term climate data tool page. For a prototype of this page see section 5 Figures and Tables figure 2.

After clicking on the short term climate data tool page you will be directed to a page that will allow you to create a new climate scenario. This page will allow the user to chose which budgets they would like to adjust and make notes about this scenario. This page also allows them to cancel or delete the scenario they are currently working on. For a prototype of this page see section 5 Figures and Tables figure 3.

Once the user Selects their budgets, makes notes on the scenario and clicks continue the will be redirected to a page where they will be prompted to enter their location. This page will take a county and a sate as input. This page will have a drop down menu for the state and the county. Before the user can enter a County they will be forced to enter a state. Once a state is entered the county drop down menu will auto populate and the user will be allowed to enter a county. For a prototype of this page see section 5 Figures and Tables figure 4.

After clicking continue the user will be taken to the plots for the location they selected. This page will initially have the 7 month average precipitation plot selected as the default. However, the user can select from four options via a drop down menu. Changing what is selected in the drop down menu will change which plot is displayed. Once the user has reviewed the plots they can then enter in how much they think their yield will be effected based on the climate data. For a prototype of this page see section 5 Figures and Tables figure 5.

Next the user will be directed to the budget review page. This page will display a summary of the budget. The summary of the budget will include Income, General Cash Costs and A total summary of the budget. This page will allow the user to adjust the cost per unit, input costs, and quantity sold and will adjust their budget in near real time. This page will also allow the user to remove or add inputs to their general costs. This page will also allow the user to save their budget and output it as a PDF. For a prototype of this page see 5 Figures and Tables figure 6.

3.1.2 Hardware Interfaces

This project requires no designed hardware and the hardware this system is running on is managed by the Operating system as a result no hardware interfaces are necessary. Additionally the connection the the NWCTB is managed over the network and requires no hardware interface.

3.1.3 Software Interface

There are several software interfaces in this application, one between the back end and the front end, one between the back end and the Database and finally a software interface between our RESTFUl API and the NWCTB. The majority of the connections between the front end and the back end are managed for us the the current *AgBiz Logic* system. However, we will still need to connect our API to the back end to provide the data from the NWCTB. To make this connection we will use a RESTFULL API. This will allow for the back end code to make a direct call to our API to extract the data.
We will also need a connection between the NWCTB and our API. At the time of writing this document we have not received any information from the NWCTB regarding the API they have promised access too. However, We do know that we will need to authenticate the connection with the data base, send a request and then receive the result. Authenticating the connection will involve some sort of hand shake where we pass the NWCTB a key. Once we authenticate the connection we can then send a request. A request will contain a variety of information including a location for which we want the data. Once we have sent the request we will then wait for the response and accept the data. We may also need to store this data is a data base. If this is necessary we will need an additional software interface between the database and the API.

3.1.4 Communications Interfaces

This project will require communication between its various parts. One key lane of communication is between the front end and back end of the application. Another key lane of communication will be between the NWCTB and our API. Most of the communication between components will be carried out through HTTP requests. These are managed by the operating system and will not effect the way our application works.

3.2 Functional Requirements

In this Section we will list and discuss the functional requirements for this project. Each functional requirement shall have an ID, Title, Description and Dependencies. The ID shall be unique for each requirement. The description will give a detailed expiation of the requirement. The dependencies section will list the requirements that will need to be complete before starting work on that requirement.

3.2.1 Functional Requirements For API

Functional Requirement 1.1 ID: FR1.1
Title: Request for Users Location Data
Description: The API shall take an HTTP Post request with the users State and County as parameters. Upon receiving the HTTP Post request the API will strip the parameters off the Request and store the values in variables for later use.
Dependencies: FR2.1 and FR2.2

Functional Requirement 1.2 ID: FR1.2
Title: Transforming Users Location data.
Description: The API shall take the users Sate and county information and transform it into latitude and longitude. It shall then store the results in a variable for later use.
Dependencies: FR1.1 and FR2.1

Functional Requirement 1.3 ID: FR1.3
Title: NWCTB Authentication
Description: The API shall authenticate the connection with the NWCTB by sending a request to connect to the NWCTB. This request will include a validation key to ensure that the connection is valid.
Dependencies: None.

Functional Requirement 1.4 ID: FR1.4
Title: Requesting data
Description: After the correct user information has been gathered and the connection with the NWCTB has been authenticated the API shall send a request to the NWCTB API for the information the user requested. This request shall be made via an HTTP Get request.
Dependencies: FR1.1, FR1.2, FR1.3 and FR2.1

Functional Requirement 1.5 ID: FR1.5
Title: Receive Response from NWCTB
Description: After sending the request for the data to the NWCTB API the API shall wait for a response to the request. Once a response has been send the API will receive this response through an HTTP Response. If the request for the user data results in an error the API shall send the appropriate response code along with a useful error message. This shall be done through an HTTP Post response.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4 and FR1.5.

Functional Requirement 1.6 ID: FR1.6
Title: Processing the Data to JSON
Description: Once the request has been successfully received the API shall process the raw data in the response body of the HTTP response sent by the NWCTB API. The raw data shall be placed into a JSON object and useful labels shall be applied to the JSON object so it can be easily displayed later.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5 and FR2.1

Functional Requirement 1.7 ID: FR1.7
Title: Send the Data to the Front End
Description: Once the data has been place into a JSON object and formatted the API shall send the resulting data to the front end application so it can be displayed. This shall be done via an HTTP Post response. The JSON object that contains the formatted data shall be placed in the response body. The appropriate headers for the Response shall be set to indicate that the response contains a JSON object.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR1.6 and FR2.1

3.2.2 Functional Requirements For Front End

Functional Requirement 2.1 ID: FR2.1
Title: Landing Page.
Description: Upon loading the *AgBizClimate* application the user will be directed to a loading page this page will allow the user to select between the short term climate tool and the long term climate tool.
Dependencies: None.

Functional Requirement 2.2 ID: FR2.2
Title: Request data from API
Description: After selecting the short term data tool the user will be directed to a page where they will be required

to enter the location they would like to analyze. This page will require the user to first enter a state via a drop down menu and then a county via a drop down menu. Clicking the continue button will send an HTTP request to the API for the climate data for the location the user requested. The Front end will then wait for a response to the request sent to the API. This request shall be made through an HTTP Post request.
Dependencies: FR2.1

Functional Requirement 2.3 ID: FR2.3
Title: Plot the Data
Description: After the API processes the request for the data for the location the the user specified the front end will receive the response from the API. The front end shall then take the data out of the response body and use a plotting library to plot the data and display it to the user.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR, 1.6, FR1.7, FR2.1 and FR2.2

Functional Requirement 2.4 ID: FR2.4
Title: Entering Adjustments to yield
Description: Once the data has been plotted and displayed the the user front end shall allow the user to make adjustments to the expected yield of the crop they are analyzing. This shall be done via a text box. The user input shall be limited to a decimal number with no more than one decimal place.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR, 1.6, FR1.7, FR2.1, FR2.2 and FR2.3

Functional Requirement 2.5 ID: FR2.5
Title: Redirect To Budget Tool.
Description: Once the user has entered the adjustments to the yield for the crop they are analyzing they shall then be redirected to the existing *AgBizClimate* budgeting tool. This tool will be sent the data they entered in the previous step and shall reflect the adjustments to yield that they made.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR, 1.6, FR1.7, FR2.1, FR2.2, FR2.3 and FR2.4

3.2.3 Functional Requirements for Testing

Functional Requirement 3.1 ID: FR3.1
Title: Unit Tests For API Routes.
Description: We shall write unit tests to cover 100 percent of our API routes. Each route shall have a test case that is non-trivial and seeks to emulate how the route will be used by the application. These test cases shall test that the correct response is returned from each API route.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR1.6 and FR2.1

Functional Requirement 3.2 ID: FR3.2
Title: Unit Tests for User Actions on Front End
Description: We shall provide unit tests for the various user actions on the front end of the application. The front end tests shall ensure that clickable objects on the page perform the correct action. We shall provide tests for all clickable objects that redirect the application to another page. Our tests shall ensure that if a clickable action is clicked that the correct action is taken for each clickable object.
Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR, 1.6, FR1.7, FR2.1, FR2.2, FR2.3, FR2.4 and FR2.5

Functional Requirement 3.3 ID: FR3.3
Title: Unit Tests for Budget Save
Description: We shall provide unit tests that ensure that the budget is being correctly saved once the user is all

done making adjustments to their budget. This test case shall ensure that the values sent to the back-end of the application are posted to the database. The test will also ensure that the values posted in the database are correct. Dependencies: FR1.1, FR1.2, FR1.3, FR1.4, FR1.5, FR, 1.6, FR1.7, FR2.1, FR2.2, FR2.3, FR2.4 and FR2.5

3.3 Performance Requirements

In this section we will list the performance metrics for our application. This performance metrics will describe acceptable performance for our application. Each requirement will have a title and a description that provides details for how that metric will be measured.

3.3.1 Performance Metric 1

Title: Run Time Performance
Our application shall provide the plotted data in a reasonable amount of time 100 percent of the time. Reasonable being defined as less than 5 seconds. This time shall be measured from the time the user presses the next button after selecting their state and county to the time the web page is loaded with the plot being displayed.

3.3.2 Performance Metric 2

Title: Reliability
Our application shall provide the correct result to the user 100 percent of the time. This result can include an error if the user enters a location that doesn't exist or if the data the user request is unavailable.

3.4 Design Constraints

The constraints of this project can be divided into two parts. Software constraints and hardware constraints. In this section we will discuss these constraints.

3.4.1 Software Constraints

This project shall be implemented using Django, Angular JS and SQL databases. What we add must also be done using Django, Angular JS and SQL.

Additionally, what we create must work with the the existing *AgBiz Logic* system. This means we will be forced to use software interfaces and databases that have already been implemented that we have no control over.

3.4.2 Hardware Constraints

Another factor that will effect how our software works is the hardware that this system is running on. The hardware this system runs on has already been determined and is managed by the university. We don't expect that the hardware will affect our project. However, we have no control over the hardware the *AgBiz Logic* system is running on.

3.5 Other Requirements

This section will cover any additional information pertinent to this project but not covered in other sections.

3.5.1 Stretch Requirements

In this section we will discuss requirements not apart of our contract with our client. These goals will not be required as apart of this project but are objectives that we would like to complete.

Stretch Requirement 1.1 ID: SR1.1
Title: Location Via a Map
Description: In requirement 2.2 instead of using two drop down menus to select the location the user would be able to use a pin on a map of the US to select their location. This would be done through an existing library that would allow us to take a pin on a map and produce a latitude and longitude.
Dependencies: FR2.1 and FR2.2

4 Gantt Chart

An approximate schedule for the AgBizClimate project is shown below. Note that this schedule is subject to change.

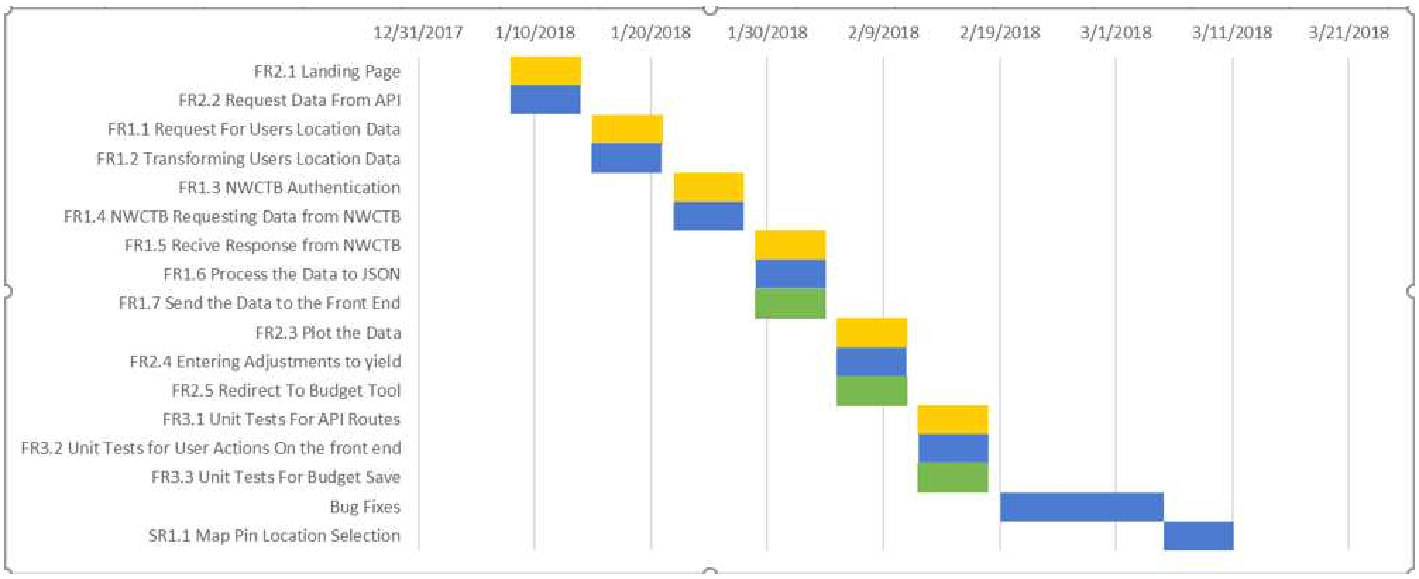
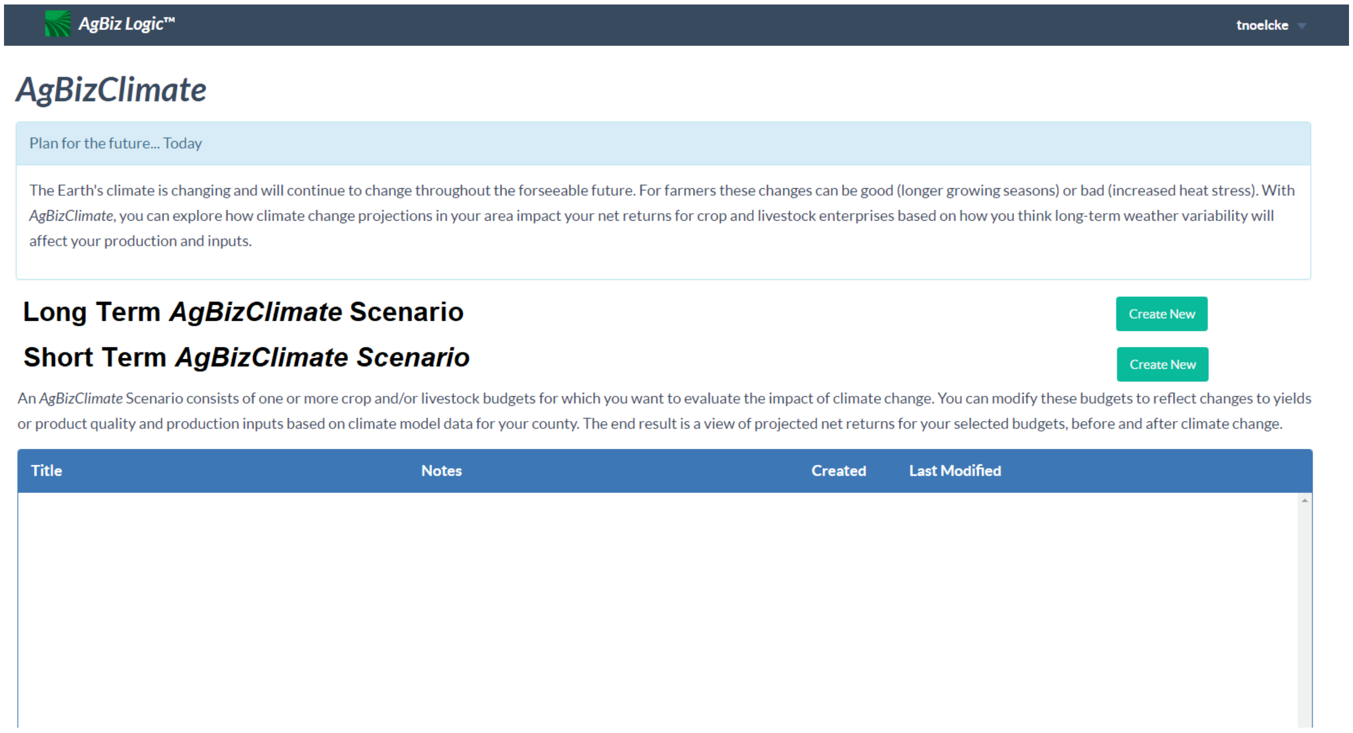


Figure 1: Project Schedual

5 Figures and Tables


AgBiz Logic™

tnoelcke

AgBizClimate

Plan for the future... Today

The Earth's climate is changing and will continue to change throughout the forseeable future. For farmers these changes can be good (longer growing seasons) or bad (increased heat stress). With *AgBizClimate*, you can explore how climate change projections in your area impact your net returns for crop and livestock enterprises based on how you think long-term weather variability will affect your production and inputs.

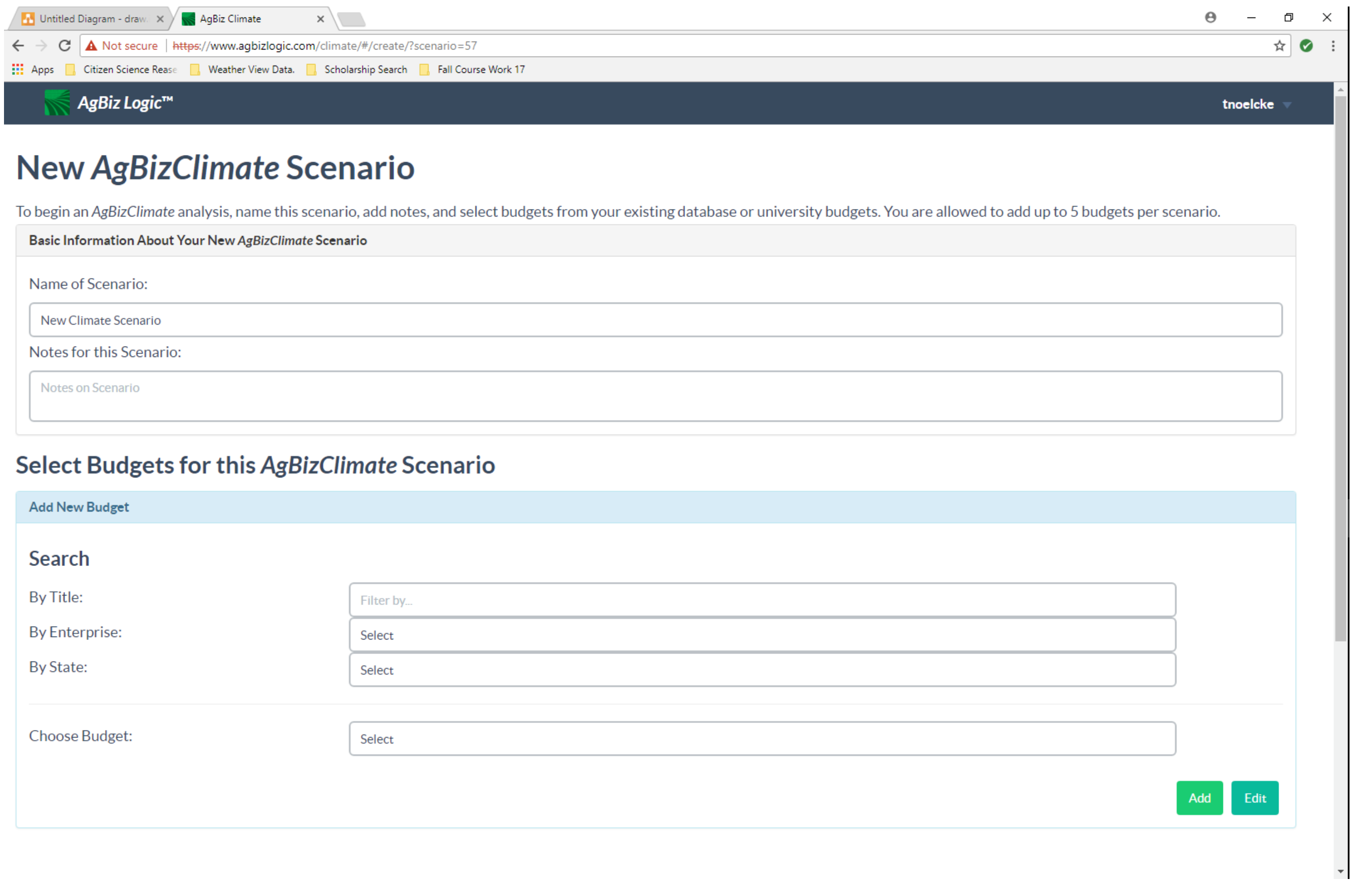
Long Term *AgBizClimate* Scenario — Create New

Short Term *AgBizClimate Scenario* — Create New

An *AgBizClimate* Scenario consists of one or more crop and/or livestock budgets for which you want to evaluate the impact of climate change. You can modify these budgets to reflect changes to yields or product quality and production inputs based on climate model data for your county. The end result is a view of projected net returns for your selected budgets, before and after climate change.

Title	Notes	Created	Last Modified

Figure 2: Landing Page



New *AgBizClimate* Scenario

To begin an *AgBizClimate* analysis, name this scenario, add notes, and select budgets from your existing database or university budgets. You are allowed to add up to 5 budgets per scenario.

Basic Information About Your New *AgBizClimate* Scenario

Name of Scenario: New Climate Scenario

Notes for this Scenario: Notes on Scenario

Select Budgets for this *AgBizClimate* Scenario

Add New Budget

Search

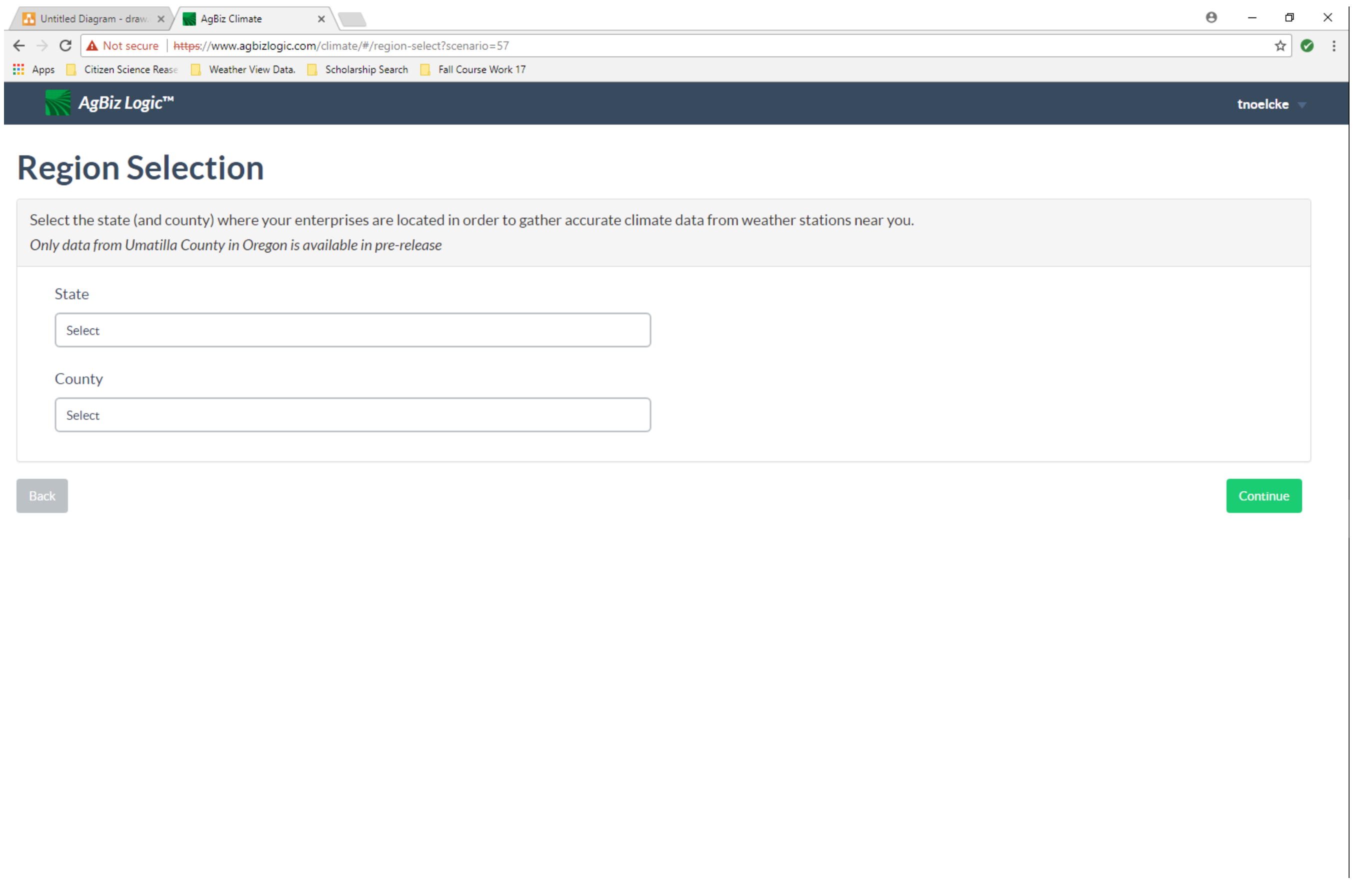
By Title: Filter by...

By Enterprise: Select

By State: Select

Choose Budget: Select

Figure 3: Allows user to select budgets and make notes



Region Selection

Select the state (and county) where your enterprises are located in order to gather accurate climate data from weather stations near you.

Only data from Umatilla County in Oregon is available in pre-release

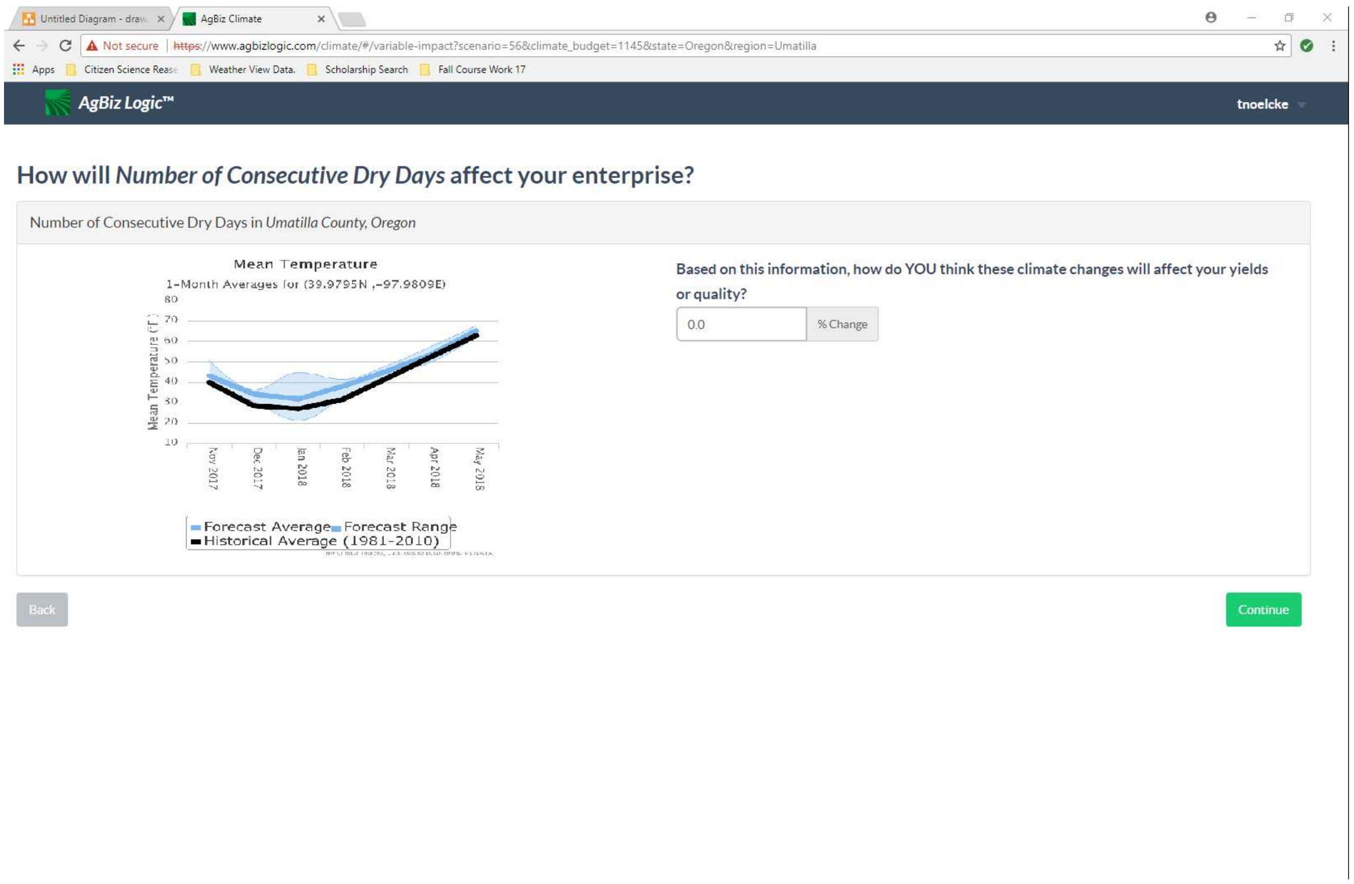
State

Select

County

Select

Figure 4: Allows User to select location



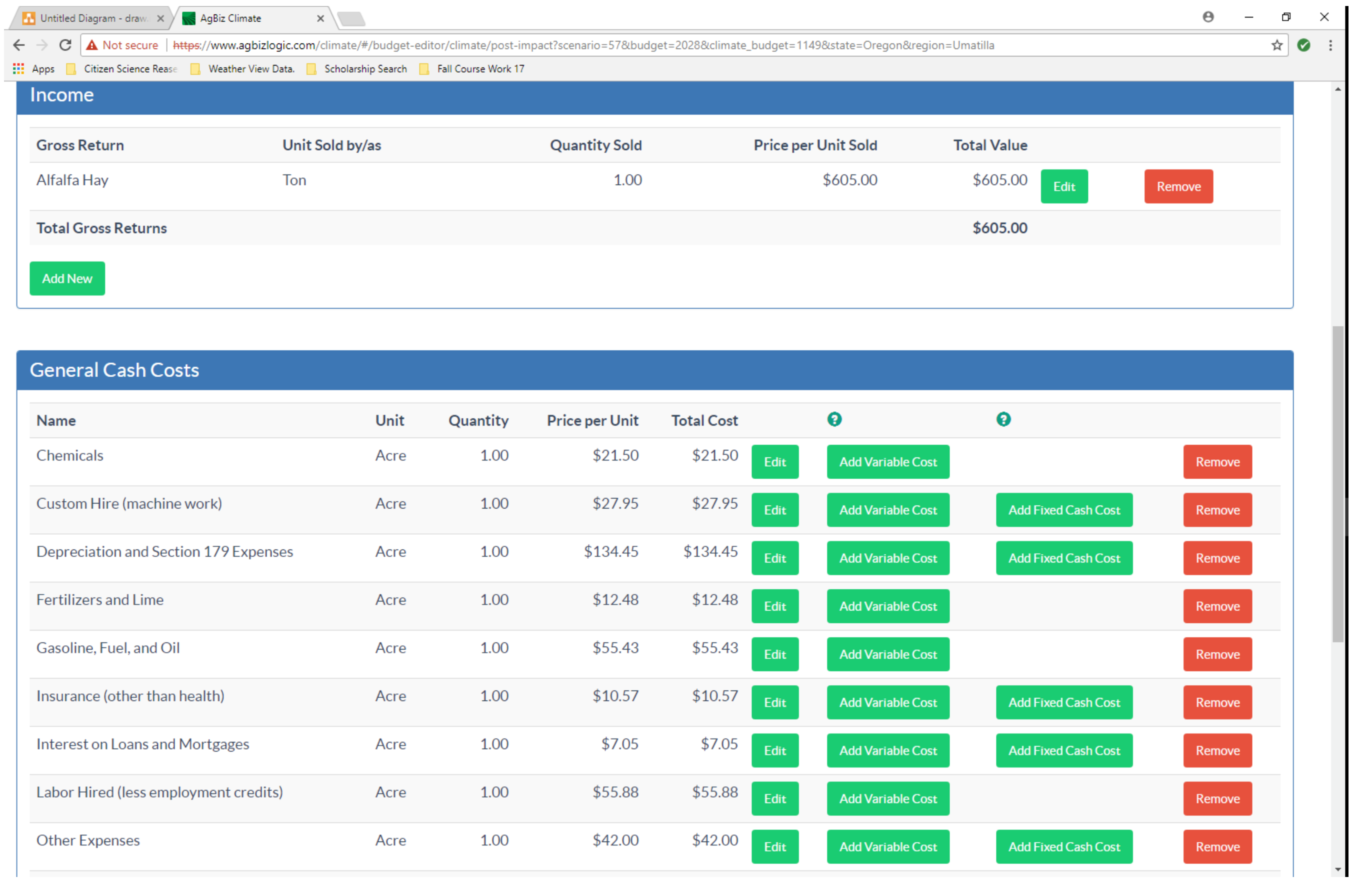
How will *Number of Consecutive Dry Days* affect your enterprise?

Number of Consecutive Dry Days in *Umatilla County, Oregon*

Based on this information, how do YOU think these climate changes will affect your yields or quality?

0.0 % Change

Figure 5: Displays the Data in a plot



Untitled Diagram - draw
AgBiz Climate

Not secure | https://www.agbizlogic.com/climate/#/budget-editor/climate/post-impact?scenario=57&budget=2028&climate_budget=1149&state=Oregon&region=Umatilla

Apps | Citizen Science Rease | Weather View Data. | Scholarship Search | Fall Course Work 17

Income

Gross Return	Unit Sold by/as	Quantity Sold	Price per Unit Sold	Total Value		
Alfalfa Hay	Ton	1.00	$605.00	$605.00	Edit	Remove
Total Gross Returns				$605.00		

Add New

General Cash Costs

Name	Unit	Quantity	Price per Unit	Total Cost				
Chemicals	Acre	1.00	$21.50	$21.50	Edit	Add Variable Cost		Remove
Custom Hire (machine work)	Acre	1.00	$27.95	$27.95	Edit	Add Variable Cost	Add Fixed Cash Cost	Remove
Depreciation and Section 179 Expenses	Acre	1.00	$134.45	$134.45	Edit	Add Variable Cost	Add Fixed Cash Cost	Remove
Fertilizers and Lime	Acre	1.00	$12.48	$12.48	Edit	Add Variable Cost		Remove
Gasoline, Fuel, and Oil	Acre	1.00	$55.43	$55.43	Edit	Add Variable Cost		Remove
Insurance (other than health)	Acre	1.00	$10.57	$10.57	Edit	Add Variable Cost	Add Fixed Cash Cost	Remove
Interest on Loans and Mortgages	Acre	1.00	$7.05	$7.05	Edit	Add Variable Cost	Add Fixed Cash Cost	Remove
Labor Hired (less employment credits)	Acre	1.00	$55.88	$55.88	Edit	Add Variable Cost		Remove
Other Expenses	Acre	1.00	$42.00	$42.00	Edit	Add Variable Cost	Add Fixed Cash Cost	Remove

Figure 6: Allows user to view their budget

2.2 Changes

3 Design Document

3.1 Original Document

CS Capstone Software Design Document

January 20, 2018

Linking Seasonal Weather Data to AgBizClimate™

Prepared for

Oregon State University

Clark Seavert

Sean Hammond

Prepared by

Group26

Thomas Noelcke

Shane Barrantes

Shengpei Yuan

Revision History

Revision	Date	Author(s)	Description
1.0	1.12.17	Thomas Noelcke, Shane Barrantes, Shengpei Yuan	Preliminary Draft

Abstract

This design document will cover the proposed design of the AgBizClimate™project. We will first give a general introduction to the project. This section will provide some context for why we are doing this project and what this project hopes to accomplish. Next we will talk about Architecture design. This section will describe a high level structure for the project. After that we will discuss the data for the project and how it will be structured. Then we will discuss in detail the design of each component in the *AgBizClimate* system. We will then discus the

design of each view in the user interface. Finally, we will provide a requirements matrix which will show how each component fulfills the functional requirements outlined in the requirements document.

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of this Software Design Description (SDD) is to describe the architecture and system design of the *AgBizClimate* project. This document will provide a high level design for the *AgBizClimate* short term climate tool. This document will also provide a detailed description of the design of the data for this project. We will also break down each component and discuss the design of each component in detail. After that we will discuss the design of each view in the user interface. Finally, we will have a requirements matrix. The requirements matrix will show how each component fulfills the functional requirements for this project. This document is intended for the project owners and software developers of the *AgBizClimate* system. This document is intended to be a guide for the implementation of the *AgBizClimate* short term climate tool.

1.2 Overview

Seasonal climate is one of the essential factors that affects agricultural production. As a module of *AgBiz Logic*, *AgBizClimate* delivers essential information about climate change to farmers, and help professionals to develop management pathways that best fit their operations under a changing climate. This project aims to link the crucial seasonal climate data from the Northwest Climate Toolbox database to *AgBiz Logic* so that it can provide changes in net returns of crop and livestock enterprises through powerful graphics and tables.

1.3 Scope

This project is a part of a much larger AgBiz Logic$^{\mathrm{TM}}$program. However, the purpose of this project is to add a short term climate tool to the *AgBizClimate* module. This limits the scope of the project to the *AgBizClimate* Module. Additionally, we will only be adding the short term climate data tool as the long term climate data tool already exists.

Currently *AgBizClimate* has a long-term climate tool but no such tool exists for short term climate data. We will implement a tool to extract short-term climate data from the Northwest Climate Toolbox database, display it to the user and allow the user to adjust crop and livestock yields or quality of products sold and, production inputs. Moreover, a landing tool will be developed to allow users to switch between short-term seasonal tool and long-term climate data tool.

1.4 Definitions, Acronyms and Abbreviations

REST - Representational State Transfer, This is a type of architecture that manages the state of the program. This is especially popular in web development.
API- Application Programming Interface. This is a piece of software that allows a connection to another piece of software providing some sort of service.
NWCTB - Northwest Climate Toolbox. This is the database we will be connecting to that will provide the short term climate data we plan to use.
Climate Scenario - This is a theoretical calculation of yields, inputs and of the overall budget for one situation based on the climate data.
SQL Database - This is a relational database that allows for storing and accessing data.
NOSQL Database - This is a non-relational database that allows for data storage and data access.
UI - User Interface, This is a piece of software that allows a human to interact with the software. Often this is what the user sees while using software.

1.5 References

[1] C. F. Seavert, "Negotiating new lease arrangements with the transition to direct seed intensive cropping systems," 2017.

[2] S. Y. Thomas Noelcke, Shane Barrantes, "Problem statement," 2017.

2 System Overview

2.1 Product Functions

AgBizClimate is a web based decision tool that will allow users to gain specific insight on how localized climate data for the next seven months will affect their crop and livestock yields or quality of products sold and production inputs. The *AgBizClimate* tool will allow users to input their location (state, county) and a budget for the specific crop or livestock enterprise. *AgBizClimate* will select climate data for the next seven months for that location and provide graphical data showing temperature and precipitation. Users will then be able to change yields or quality of product sold by a percentage they think these factors will affect and modify production inputs. Finally the tool will calculate the net returns.

2.2 User Characteristics

AgBizClimate users can be split into two subgroups, agricultural producers and climate researchers. The first subgroup, agricultural users who use this product tend to be between fifty and sixty years old of mixed gender. Their educational background ranges from high school to the completion of college. The primary language this group uses is English, but there are some Spanish users as well. Most of the users in this group tend to have novice computational skills. The primary domain for these users is agricultural and business management. Most agricultural producers who use this product are motivated by the potential profit that the decision tool *AgBizClimate* could potentially offer. The second subgroup, climate researchers range from ages twenty to forty and are of mixed gender. The educational background for most climate researchers exceed the postgraduate level with their primary language being English. These users generally have advanced computational skills and are motivated by the easily accessible climate and weather data.

2.3 Constraints

There are several key constraints that this product has to work within. The first constraint is that we only have access to two data parameters from the North West Climate Tool box, precipitation and temperature. Secondly, we only have access to their data via the NWCTB API which could have additional restrictions such as limited usage per day, mislabeled data, or poor documentation. Thirdly, we dont have access to any of the hardware that *AgBizClimate* is exists on as it is being managed by a third party. This will prevent us from improving the hardware or cause roadblocks if their servers are having issues. Lastly, we are limited to using the languages Python and JavaScript since we are integrating our product into an already existing project.

2.4 Assumptions and Dependencies

We are assuming that the Northwest Climate Toolbox is a functional API that will allow us to pull location based temperature and precipitation data. This data will most likely come in the form of a text body of which we will then format into a JSON object and store in a MongoDB database for future use. Due to the fact that we are writing an addition to an existing project we do not need to interact with the user budgets as these have already been defined. This fact extends to the calculations portion of the *AgBizClimate* product. Our team will simply be accessing data via the NWCTB API, then format the data, store the data, and hand the data over to the tool while will provide some additional front end support.

3 System Architecture

3.1 Architectural Design

Shown Below is the architectural design for the *AgBizClimate* project. This UML diagram shows the high level components of this application. This Diagram also shows how these components will interact.

Figure 1: System Architecture Design for *AgBizClimate* project

3.2 Decomposition Description

The *AgBizClimate* project can broken down into six components. The six components are the Backend Controller, The Front End Controller, UI, Existing *AgBiz Logic* Modules, Climate Data API, and the NWCTB API. In this section we will describe each models function and how it interacts with the other modules.

3.2.1 Backend Controller

This component is responsible for connecting the dots between the rest of the components. Generally, the backend controller will handle incoming requests from the front end controller and return the requested content. This component will also interface with the existing *AgBiz Logic* modules and Climate Data API so it can provide all the requested information.

3.2.2 Front End Controller

The Front End Controller will act as an interface between the UI, the existing *AgBiz Locic* modules and the Backend Controller. This component will handle UI action made by the user and will use those to make requests to the Backend and existing *AgBiz Logic* Modules. This component will then take the result of these requests and display the relevant information to the user. This component will also handle user inputs such as a button push or clicking on a drop down menu. This component will take these actions and modify the UI to reflect the actions the user preformed.

3.2.3 UI

This is the portion of the application that the user will see and interact with. The primary responsibility of this component is to display information to the user. This component will also be responsible for interacting with the front end Controller to ensure that user actions so the program responds correctly to user action.

3.2.4 Existing *AgBiz Logic* modules

This is not a component but rather a collection of components that already exists as part of the *AgBiz logic* system. We will use these components to preform a variate of actions including, retrieving budget data, managing user information, making modifications to budget data and saving budget data back to the database. We will interface with these component from the Front End Controller to handle budget data. We will also interface with these components from the back end to handle user data.

3.2.5 Climate Data API

The Climate Data API component will interface withe the NWCTB to provide long term forecast data. This component will take requests, with location data, from the Backend controller and will respond with the formatted data from the NWCTB. To do we will interface with the NWCTB API to retrieve the data. Then we will take the data from the NWCTB, parse it into JSON, apply some formatting and pass it back to the Backend. For the purposes of this project this component is only going to interface with the Backend Controller. However in the future this API maybe used by other sections of the application as well.

3.2.6 NWCTB API

The NWCTB API will be our data source for this project. This component will provide the climate data by interfacing with the back end controller. Currently we are not sure how we will interface with the NWCTB as the NWCTB has not responded to our requests for API access.

3.3 Design Rationale

We've chosen to design this system this way in part because of the nature of our system. We need a front end controller to handle the clients interaction with the server because this is a web development project and the front end will be separate from the server. The front end controller will facilitate the communication between the client and the server.
We also chose to use a Backend Controller so we can facilitate the communication between the front end controller and the various components on the backend. This makes the application easier to build, test and maintain. This also allows for one line of communication between the backend and the front end. This is necessary to keep the interactions between the backend code and front end code simple. This allows for large changes to be made to both the front end and the backend with out causing them to impact each other.
We also chose to create the Climate Data API as its own service. We chose to do this because it's easier to test and then the Climate Data API can be reused in future projects and with the other components. If we had built the Climate Data API into the backend controller this would not have been possible.
The NWCTB and Existing *AgBiz Logic* components have already been implemented out side the scope of our project. However, we are still planning to use them in our project. This is why we have created these modules in our design because we will be using them as part of the design of our system but do not want to tightly couple our project with these existing components.
More generally the components in our system use the REST API architecture type. We chose to do this because it allows for flexible reusable modules. The REST API Architecture also allows us to break our application into independent modules that are easier to develop, test and maintain. This division of our application also makes it more scalable allowing our system to keep up with future demand.

4 Data Design

4.1 Data Description

In this section we will discuss the design of the data required for this system. The data needed to implement this system includes the user data, climate scenarios, Budget Data and the climate data. It should be noted that this project adds the climate data to the system. User data, climate scenarios and budget data have already been implemented as part of the existing *AgBiz Logic* system. However, since we will be using this data as part of our project I've included their design in this section.
Shown below is the design for the data we will use in this program. In this UML diagram are all the various entities required by this system. Additionally we also show the relationship between different entities. Shown below are the UML diagrams for User Data, Climate Scenarios, Budget Data, Climate Data and Related Entities.

Figure 2: User Data Currently Implemented in the *AgBiz Logic* project

Figure 3: Design For the Climate Data for the *AgBizClimate* project

4.2 Data Dictionary

In this section we will describe each piece of data and what it represents. We will also discuss possible values for each piece of data.

4.2.1 User Data

In this section we will define the entities and fields that relate to user data and user budgets.

4.2.2 User

This entity represents a user along with the data we will need for each user.
Name - This is a string that stores the user name of a user.

Address 1 - This stores the first line of a users address.

Address 2 - This stores the second line of a users address.

zip code - This stores the zip code for the address the user entered.

city - This stores the name of the city for the address the user entered.

state - This stores the name of the state that the user entered. The state must be a valid US state as climate data is not available for out side the US.

Industry - This stores what industry the user is involved in. Currently the user may choose from two options Agriculture or Non-Agriculture.

Primary Business - This stores what sort of business a user is employed in. For instance if a user were a farmer they would select Producer.

Secondary Business - This is an optional parameter that stores what if any other business that the user may engage in. For instance if a user is a producer but also packages their product them selves they would select packager for this option.

4.2.3 Budget

This entity represents a budget and associated data. It is important to note that Budgets are related to users by a many to many relationship. It is also important to note that this entity is related to the costItem entity and the IncomeItem entity. This relation ship is a many one to many relationship.
User - This is a reference to the User entity because a budget should be associated with a user.

Created Date - This stores the date that this budget was created.

Modified Date - Keeps track of the last time the budget was modified.

Title - The user entered title for the budget. This helps users keep their budgets organized.

notes - Users entered notes about a budget. This helps a user keep track of their budgets.

enterprise - The enterprise type that this budget represents. If it is a conventional crop we would say conventional. If it were an Organic type crop this would be Organic.

descriptor1 through descriptor6 - Generic descriptor for the budget. May be used might not be used. This item will hold descriptions relevant to the budget we are storing.

state - Stores the state that this budget is intended for.

region - Stores what region in the state that the budget is intended for.

time_unit - Stores how the user would like to measure time for that budget. Some budgets may be measured in years but some may be measured in months or weeks.

farm_unit - This item stores how we plan to measure how much land we have to work with. Most farms will choose to use acres. However for some crops its more useful to use a different type of measurement.

farm_unit_quantity - This describes how large our farm is based on the farm_unit.

expense_unit - This item stores the unit of measurement for an expense. This is similar to a farm_unit.

expense_unit_quantity This item stores how many of the expense_units we have in our budget.

total_cost Total cost represents the total cost of the crop we are using this budget for.

total_variable_costs This represents the total variable cost for this budget. Variable costs are cost that can change and depend on other variables such as fertilizer use.

total_fixed_costs - Represents the the total fixed costs for this budget. Fixed costs are costs that will be the same regardless of other variables such as land lease.

total_general_costs - This is the total general cost associated with this crop. This represents how much it will cost to produce this crop.

total_income_less_variable_cost - This is total income with out subtracting the total costs.

profit - This is how much money we will make after we adjust for the total cost of producing the crop.

breakeven_yield - This represents the quantity of a crop needed to off set the total cost.

breakeven_price - This represents the price we will have to sell our crop at given that the quantity is the breakeven_yield to off set the total cost.

total_yields This represents how many units of a crop we produced.

primary_income_quantity - This represents the quantity of the most profitable crop in my budget.

primary_income_unit - This represents the unit of measure for the primary_income_quantity.

4.2.4 Cost Item

This entity represents an item that is a cost in regards to ranching or farming. It should be noted that this entity is related to a budget via a many to one relationship.

name - This represents the name of the cost. For instance if the cost is fertilizer the name of this cost item would be fertilizer.

notes - This is user input data that allows the user to make notes about a cost item.

parent_budget - This is a reference to the budget that this cost item is associated with.

parent_category - This a broad category that this item belongs to. For instance if this item were round up it would belong to the broader category of pesticides.

category - This is the category the item falls into. For instance if the cost is paying one of my farm workers this would be labor.

sub_category - This allows for even farther categorization of cost items.

cost_type - This represents what kind of cost this item is. Mainly weather its a fixed or variable type cost.

farm_unit_quantity - This stores how many farm units we will need to apply this cost too when making calculations.

unit - This represents the unit that this cost item we be measured in.

unit_quantity - This is the number of units of this cost item that we will need.

cost_total - This is how much the this cost item will cost based on the number units needed and the cost per unit.

cost_per_unit - This represents how much each unit of this cost item will cost.

cost_per_farm_unit this represents how much it will cost to per farm_unit for this cost item.

4.2.5 Income Item

This entity represents an item that provides income in to the budget such as a crop or livestock. This item is related to the budget entity by a many to one relationship.

Name - This represents the name of the Income item. For instance if the income item is corn it would be name corn.

enterprise - he enterprise type that this budget represents. If it is a conventional crop we would say conventional. If it were an Organic type crop this would be Organic.

descriptor1 descriptor6 - Generic descriptor for the Income Item. May be used might not be used. This item will hold descriptions relevant to the Income Item we are storing.

notes - This will store notes about the Income Item to help the user keep income items organized.

weight -

farm_unit - This stores the unit that we will measure how much space we have to grow this Income item.

farm_unit_quantity This represents the number of farm_units we have to produce this income item.

sale_unit This specifies how we plan to measure the quantity of this income item.

sale_unit_quantity - This specifies how many sale units of this cost item we have to sell.

return_total - This is the total amount we would get from selling this income item.

price_per_farm_unit - This is how much money we sell our income item per farm unit we have produced.

price_per_sale_unit - This is the price we can sell this income item for per sale unit.

4.3 Climate Data

In this section we will discuss and define the entities that are required to represent climate data.

4.3.1 ClimateScenario

This entity represents a Climate Scenario. This entity keeps track of important user information in regards to Climate Simulations. It is important to note that this entity is related to the Climate Budget entity via a one to many relationship. This Entity is also related to the user entity via a many to one relationship.

user - This is a reference to the user who created the climate scenario.

Title - This a user entered title that represents the name for the scenario. This allows the user to keep their scenarios organized.

notes - This is user entered data about this climate scenario. This allows user to better track what each scenario is for.

created_date - This keeps track of the date that the climate scenario was created.

modified_date - This keeps track of the date that the climate scenario was lasted modified.

4.3.2 ClimateBudget

This entity represents one Budget simulation in a climate scenario. This entity is related to the ClimateScenario entity by a many to one relationship. This entity is also related to the user entity via a many to one relationship. This entity is also related to the budget entity via a many to one relationship.

user - This is a reference to the user entity. This represents the user who created this ClimateBudget.

climate_scenario - This is reference to the ClimateScenario entity this represents the climate scenario that this climate budget belongs with.

budget - This is a reference to the budget entity. This budget the budget we are considering for this climate budget.

modeling_estimate - This represents the estimation that the model produces for how much the climate prediction will effect the climate.

focus_group_estimate - This is an estimation of how much a model will effect a budget based on a focus group.

user_estimate - This is the estimation that the user provides for how much the climate factors will effect the budget being considered in this climate budget.

is_original - This is a Boolean flag that indicates weather or not this is an original climate budget or a university budget.

position - This indicates the geographical location that this climate budget is trying to display.

created_date - This tracks the date that the user created this climate budget.

4.3.3 ClimateFactor

This entity represents one climate factor that may effect the climate budget. For example we may consider the number of freezing nights in one year. This item is related to the climate budget entity via a many to one relationship.

climate_budget - This is reference to the ClimateBudget entity. This represents the budget we are considering for this climate factor.

name - This is the name of the climate factor. For instance average temperature.

state - This is the state for which this climate factor is being considered.

region - This is the region for which this climate factor is being considered.

user_estimate - This represents the amount the user estimates that this climate factor will impact total crop yield.

created_date - This keeps track of when this climate factor was created by the user.

modified_date This keeps track of when this climate factor was last modified by the user.

type this keeps track of what type of climate factor this climate factor represents. Currently we have two types long term and short term.

4.3.4 ClimateData

This entity represents Climate data that will be used to determine how climate change may effect a budget. This entity is related to the ClimateFactor entity in a one to one relationship.

ID - this is a unique ID by which we can identify this set of climate data.

Lat - This stores the Latitude of the climate data represented by this climate data entity.

Long - This stores the Longitude of the climate data represented by this climate data entity.

Climate_Factor - This is a reference to the ClimateFactor entity.

4.3.5 DataFrame

This entity represents one single point in time of a climate data model run. A collection of these DataFrames can represent the data for the whole model run. This entity is related to the ClimateData entity in a many to one relationship.

ID - This is a unique identifier we can use to identify this data frame.

Climate_data - This is a reference to the ClimateData entity.

DateTime - This keeps track of the date and time that this data point occurs at in time.

The Data - This is simply a place holder for the data we will need to store for each data frame. We can't currently know how this section of this entity will need to be formatted as we don't know what the data we will get from the NWCTB will look like.

5 Component Design

5.1 Front End Controller

The front end controller is essential as it is what passes the user input to the climate data API and receives valuable data from the backend to display too the user. This section will discuss individual angular components and their functions. On every page there will a toolbar that provides a link to the AgBizClimate landing page via clicking on the *AgBiz Logic* and a drop down box that provides user actions on user clicks.

5.1.1 Angular Components Design

In this section we will look at each page of AgBizClimate and view the angular components and their function. Shown below in figure **??** is the overall design for the angular components for each page. It should be noted that each box represents one page in the UI. The first section above the second bar represents the various components in the UI. The second section represents the various function that will be needed to respond to user input on the UI. For each each page we will discuss and describe the function of each UI component. We will also discuss each function in each component along with its intended purpose.

Figure 4: Overall structure for Controller and View

5.1.2 Landing Page

The landing page has three angular components: create_new_short_term, create_new_long_term, and climate_scenario_viewer. The create_new_short_term and create_new_long_term are clickable action components that utuilize the createNewShortTerm() and createNewLongTerm() functions to make calls to the back-end and begin the climate creation scenario processes. The climate_scenario_viewer component is used to make a call to the back-end using getUserScenarios() which grabs all scenarios tied to that user and presents them in a table.

5.1.3 New Climate Scenario

The new climate scenario page has two angular components: add, and edit. The add component allows users to add a new budget to their scenario from a list of provided budgets and the edit component allows users to alter budgets they already have attached to their accounts. There are two primary function calls associated to this page. The first function is getBudgets() which makes a call to the back-end to grab all available budgets for the user to select from. The second function is searchBudgets() which allows users to search for a specific budget from the list of all budgets.

5.1.4 Region Selection

The region selection page has two angular components: back, and continue. These components are clickable and cause redirects to the previous new climate scenario page or the chart page. The two functions included in this page are getStates(), and getCounties() which make calls to the back-end to grab a list of all available states and counties that AgBizClimate has seasonal weather data for.

5.1.5 Chart Page

The chart page has three angular components: back, continue, and grab. Back, and continue clickable event buttons that are used to return to the previous page or move on to the next page. The graph component is a static component that displays the seasonal precipitation and temperature for the location they selected. This page also makes use of the function estimateYieldChange() which takes user input and makes a call to the back-end to use an AgBizLogic formula to make a more accurate budget review for the user.

5.1.6 Budget Review Page

The budget review page has the most angular components for viewing and manipulating data. There are also clickable navigation components back, and continue which allow users to go back to the previous chart page or continue on and save their current climate scenario. Next, there are a series of components that allows actions to be taken on the income and cash_cost data objects. New income objects can be added via the income_add component, edited via the income_edit component, or removed from the list via the income_remove component. cash_costs can be edited via the cash_costs_edit component, assigned additional variable costs via the cash_costs_add_var_cost, assigned additional fixed costs via the cash_costs_add_fixed_cost component, or removed via the cash_costs_remove component. Finally, There are a series of functions that these components use to make calls to the back-end to retrieve and manipulate data. The getIncome() and getCashCosts() functions retrieve the income and cash_costs associated with the budget to be displayed in the view. The addIncome(), addCashCostsFixed(), and addCashCostsVariable() functions are all used to add an aditional data object to the data objects or data object lists. The removeIncome() and removeCashCosts() functions are used to remove specific items from the list. Lastly, the editIncome(), and editCashCosts() functions are used to manipulate selected income and cast_costs data objects.

5.2 Controller Design

5.2.1 Overview

As an interface between Model and View, the Controller plays a very important role in the entire application architecture. First, it controls the way the application responds to the requests that result from the end-users' operations on UI. Secondly, it maintains the underlying relations between user interface (View) and data (Model). Thirdly, it accepts input from end-users, makes some necessary transformations, and then applies the final results on the data (Model). In this way, it controls the computation of the data (Models). At the same time, it could also reflect the variations on data (Model) back into UI (Views).

5.2.2 Overall Design

Overall, the controller tries to decompose the coupling between Model and View. However, as we would mainly employ the python Django framework for the whole AgBizClimate project. The Controller component acts as intermediary agent between front-end UI and back-end Climate APIs. However, it should be noted that the Django framework does not do well at separating the View and Controller clearly. As a result the view and controller are tightly interwoven. Therefore, it is better to deliberately define a middle layer between view and controllers. In this project, we could adapt some design pattern techniques called Mediator and Event dispatch mechanism. The figure

?? depicts the overall structure for Controller and View. We can see that the user action is sent to Controller from View through a Mediator to Controller. And the final effects of Controller on View are sent back to View through Mediator. And the user actions are transferred by event dispatching mechanism. Therefore, the complex dependency relations between Controller and View are largely broken.

Figure 5: Overall structure for Controller and View

According to the data design, there are four kinds of general data types: user data, climate scenarios, budget data and the climate data. Consequently, we will need four general controllers to handle each data item. However, it is not possible or reasonable to deal with all of the business logic within the four general controllers. We plan to define some specific controllers to deal with more concrete tasks and organize them hierarchically. That is, underneath the four high level general controllers there are more low level small detailed controllers for various businesses / events. And the events are transferred gradually from lower levels to higher levels. In figure **??** presents the top Controller design of the AgBizClimate project. Basically there are two general controller types: Frontend Controller and Backend Controller, which mainly handles the operations/events from UI and data models, respectively. Additionally, much of the work are actually finished by the four specific controller under Backend Controller: Climate Scenario Controller, Climate Data Controller, Climate Factor Controller and Climate Budget Controller. Also, it should be noted that the Backend Controller will call the existing AgBiz Logic Model for certain tasks.

Figure 6: Top controller design diagram

5.3 API Design

5.3.1 Overview

In this section we will discuss the overall design of the API that will interface with the NWCTB. This API needs to get the data from the NWCTB, format the data, and send it to the client. Currently, there is a lot of uncertainty around the design of this API because we do not know what sort of API access that we will be given from the NWCTB. We are trying to contact the NWCTB development team regarding our API access but the NWCTB hasn't been very responsive. Because we still don't have NWCTB API access yet and have no date when this might be accomplished, we will discuss several possible options that do not require NWCTB API access along with one design option that includes NWCTB API access.

5.3.2 Diagrams

Shown below are two diagrams. The first diagram shows a UML design of this module shown in Figure **??**. Including the route for the API, the functions public and private and the data models used as part of this module. Also shown below in Figure **??** is how a typical transaction will work with the Climate Data API.

Figure 7: Design of Climate Data API and associated models

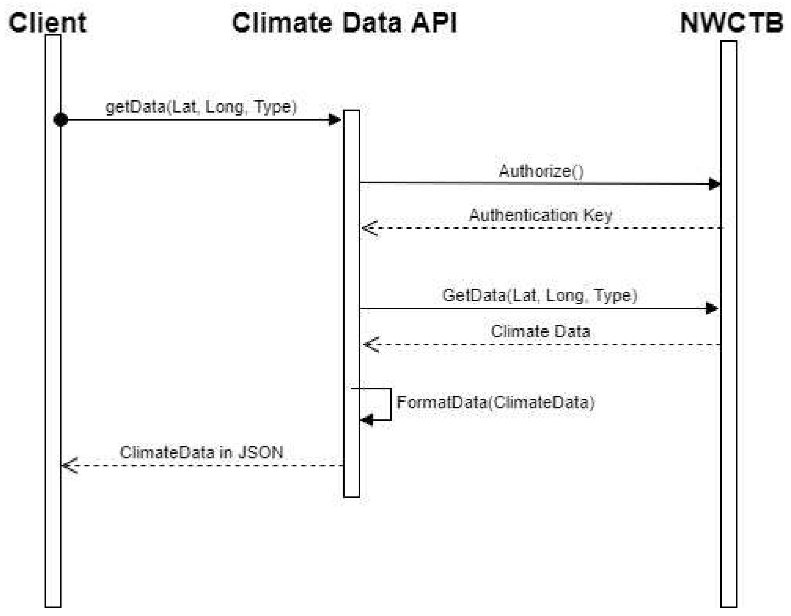


Figure 8: Typical Climate Data API Transaction

5.3.3 Overall Design

The climate data API can be divided into three different parts. The API Controller it self, the Data that the controller will be using to represent the climate data and the NWCTB interface. The following paragraphs will briefly describe each of these components.

5.3.4 Function Design

In this section we will define and describe each function in the Climate API design. We will also discuss what each function will take as input and what each function will return.

Public Methods

GetData(lat, long, parameterType) - This function call takes three paramaters as input. latitude, longitude and parameter type. The parameter type is the type of data that the client is requesting either precipitation or temperature. This call will then return the requested climate data as a JSON object to the client.

RouteCall: " /getdata/{lat}/{long}/{paramaterType}" - This route takes three parameters via the URL as input. These parameters are the latitude, longitude and the ParameterType. The parameter type is the type of data the user requested. In this case it's either precipitation or temperature. The data will then be formatted as a JSON object and will be returned to the calling client.

Private Methods

Authenticate() - This function will preform a hand shake with the NWCTB API to let the NWCTB API know who we are. This function will make a call to the Authorize function in the interface of the NWCTB API. This function will then get the authentication key for later use when we go to make requests to the NWCTB API.

GetDataFromNWCTB(ClimateData) - This function will set up and make the request to the NWCTB API. This function will take a ClimateData obejct as a parameter preloaded with the ID, Lat, and Long fields completed. This function will return the climateData object after initializing the DataFrame collection with the data received from the NWCTB.

FormatData(ClimateData) - This function will take a climate data object and will transform the object into a JSON object and return it.

5.3.5 Model Design

In this section we will define and discuss the models we plan to use in the Climate Data API. For each model we will define and describe each data member and function.

5.3.6 NWCTB Interface

We will not be talking about how the NWCTB Interface is designed as for the purposes of this project this is a black box that returns the climate data we requested. However we will discuss the method calls we plan to use for the NWCTB API.

It should also be noted that this sections is what we are guessing the API Access should look like. Currently we are unsure how we will be accessing the API and what the calls will look like. But for the purposes of the design of this system we have taken our best guess and what it might look like.

Methods

Authroize() - This function takes no parameters and preforms a handshake between our application and the NWCTB application. This function call will return an authentication key that we can use later to make function calls to the NWCTB API.

getData(lat, long, Type) - This call takes latitude, longitude and the data type we are requesting. The data type in this case will either be precipitation or temperature. This function will then return the data we requested or an error if the request we made was unauthorized or was a bad request.

5.3.7 Data Design

In this section we will discuss the design of the models that we will use for the Climate Data API. For each model we will define and describe the fields and methods.

ClimateData This model represents the climate data for one whole climate simulation.

Fields ID - This is an ID that uniquely identifies this ClimateData object. This will be used to find specific model runs. Lat - This represents that latitude at which this climate situation is located.
Long - This represents the longitude at which this climate situation is located.
DataFrame[] - This is a collection of data frames that make up the climate data for this climate simulation.

Methods toJSON() - This method takes the ClimateData object and turns it into the equivalent JSON representation.

DataFrame This model represents one frame of data for a climate simulation. A data frame represents one moment in time in a climate simulation. A collection of these data frames make up a climate simulation.

Fields ID - This is a Unique identifier that will allow us to identify the order that the data frames need to go in.
Date_time - This is the date time that this data frame occurs on in time.
The Data - This is a place holder for the data. We currently do not have API access to the NWCTB. As a result we are not sure how the data will be formatted. Once we have the data from the northwest climate tool box we will able replace this place holder with the actual data for each frame.

Methods ToJSON() - This produces the equivalent JSON representation of the data frame.

5.3.8 Possible Alternative to the NWCTB

In this section we will discuss other possible solutions for getting our climate data other than using the NWCTB. We are discussing this because it is possible that we may not get API access to the NWCTB or that we will get access after this project is completed. Because there is some uncertainty regarding the API access we have come up with a few different options if we are unable to get the necessary API Access.

Automate Downloading the Data
One possible solution to getting the data without downloading the data is to write a script to download the data from the website. The data is available for download on their website and It wouldn't be too difficult to create a scrip that goes to their website and download the data. There are some potential issues we may run it using this approach. Firstly, they may throttle our speed if we try to download to many data entries at once or to often. This may happen many times as a person goes through climate scenarios for different crops. One way to over come this problem would be to cache the data in a database locally. This would allow us to only ever grab a data set for a location once. However, with this approach we would need to find a way to ensure that this data is the most updated data. This is the preferred alternative if we do not end up getting the NWCTB API access.

Build a New Service From the Ground Up
Another possible solution would be to build a new service that went out to the NOAA got the different climate predictions and averaged together the result. The model data used by the NWCTB is public record available to the public for download. It would be possible to create a new service that did essentially the same thing as the NWCTB but without the user interface. There are a few nice things about this solution in that it would give us control over the data from start to finish and would allow us to make calls to an internal service rather than an external one. However, this would have serious impacts on our development time. This would also be a rather complex problem that would require a lot of research. Preferably we will not use this option.

Find A New Climate Data API
Finally, Another solution would be to find a different API for the climate data we need. Currently we are unaware if another Climate Data API exists that would suite the needs of our application. However, if we were to find one that provided the data we need changing the design of our application to accommodate the new API probably wouldn't cause big issues. Assuming we can find another data source this approach would be ideal.

5.4 Testing Design

5.4.1 Front End Testing

We plan to test the front end using a Angular testing framework. Using this testing frame work we will test that every possible user interaction produces some sort of UI action. Essentially the idea is that we ant to test that our UI is responsive for every possible UI action.

5.4.2 Controller testing

Controllers are the workhorse of MVC. And the tests of them would to some extent guarantee the correctness of the business logic of the application. Basically, most controllers either render a view or handle form submissions. Thus the test of controller should mainly focus on the concrete user operations or response but not the entire framework or functionalists of the application. And these tests would be done in the phase of unit test. Generally, each test for controller may consist of two steps.
Firstly, one launches a request to the controller method to be tested. This is mainly done by user inputs from the view. In practice, the mocking actions from testing frameworks would replace human operations to do this.
Secondly, one verifies that expected response is received or certain effect has taken place. This step could also be done easily by auto testing frameworks after one has define appropriate test cases.

5.4.3 API Testing

For the API we will provide testing for every possible API route. We will also test that erogenous resumes result in correct response. Beyond that we will also test each function in the API. For each function we will test a variety of different inputs for each function including valid and invalid inputs. We will then ensure that the correct result is returned for valid requests. For invalid request we will ensure that the correct error message is returned.

6 User Interface Design

6.1 Overview of User Interface

The user interface is an essential part of any web application as it is the layer between the user and what they want and need out of an application. In this section we will be discussing the User Interface. Specifically, we will be detailing the specific UI elements that make up each page.

6.2 Screen Images

Figure 9: Landing Page

Figure 10: Allows user to select budgets and make notes

Figure 11: Allows User to select location

Figure 12: Displays the Data in a plot

Figure 13: Allows user to view their budget

6.3 Screen Objects and Actions

6.3.1 Landing Page

The Landing page consists of two primary parts: the information jumbotron, new scenario creation, and a scenario listings. The information jumbotron is a static and gives the mission statement of the AgBizClimate product. The new scenario creation selection allows the user to click on one of two boxes specifying if they would like to create a new short or long term climate scenario. Finally, the scenario listings section is a dynamic table that provides information on the title, notes, date created, and date last modified for all entered climate scenarios.

6.3.2 Climate Scenarios

The new climate scenario page consists of three primary parts: the information section, and budget selection. The information section will state that the user is beginning a new AgBizClimate scenario and provide rules and data entry components for the scenario and notes for the new scenario. Specifically it will allow users to enter text specifying the scenario name and notes for the new scenario. Finally budget selection portion will allow users to add a budget or select a preexisting budget from the existing database.

6.3.3 Region Selection

The region selection page will consist of a jumbotron stating that the user is on the region selection page, a brief message explaining what the page is for, and two drop down boxes that the user can use to select the desired state and county for their new climate scenario. This page will also have a back, and continue button that allow users to either backtrack through the new climate creation process or move on to the new step.

6.3.4 Chart Page

The chart page will consist of a title prompting the user to consider how the following graph will affect their enterprise, a graph containing localized temperature, and precipitation data based on previous user input in the region selection page and a text box where the user can enter the percent they think the forecast will affect their yields or quality. This page will also contain the back, and continue buttons the other pages have which allow the user to more easily move around the tool.

6.3.5 Budget Review

The budget review page will have two major section. The first section is the income section which is a table containing the following information: income gross returns, unit sold as, quantity sold, price per unit, and total value. This table will also sum the total value from this section for the user. The second section is a general cash costs section and contains the the following information: name, unit, quantity, price per unit, and total cost. Finally for each table users will have access to buttons that will allow them to add new entries, remove entries, and edit entries.

7 Requirements Matrix

Shown below in figure **??** is our requirements matrix. This matrix shows how the different components will fufill the project requirements. For a full list of requirements please see the AgBizClimate requirements document.

Figure 14: Requirements Matrix

3.2 Changes

4 Tech Review

Oregon State University

CS 461

Fall 2017

Tech Review AgBizClimate©

Author:
Thomas Noelcke

Instructor:
D. Kevin McGrath
Kirsten Winters

Abstract

The purpose of this document is to research and consider different technical options for our application. In this document we research different options for data storage, HTTP request frame works, and testing frameworks. We will consider three possible choices for each section of the application. For each of these options we will weight the pros and cons of each. After comparing the different options we will select the option we would like to use for the *AgBizClimate* application. We have divided our application into 9 different section and each of us has preformed this analysis for each of the nine sections.

Contents

Oregon State University

CS 461

Fall 2017

Tech Review: Linking Seasonal Weather Data to AgBizClimate

Author:
Shane Barrantes, 29

Instructor:
D. Kevin McGrath
Kirsten Winters

Abstract

This document will provide an overarching analysis on front-end frameworks, graphing frameworks, and back-end design that we considered and selected for the AgBizClimate project. The topics included in this document will mirror my primary responsibilities for this project.

1 Front-End Frameworks

1.1 Overview

Front-end technologies are extremely important to web applications because they supply the foundation and interface for user interaction. This interaction is the first glimpse that users get into the application and will use it to judge the appearance, feel, and usability of the product. We want to use a front-end framework to give structure to our front-end interface, creating a groundwork for our application and making sure we can add any additional features with minimal time expenditure. We will be using the framework to build a user interface that is easily navigable and will provide dynamic graphing output based on user input.

1.2 Criteria

The front-end framework we pick needs to accomplish several things. First, it must enable our project to be highly customizable. Secondly, it needs to have a logical structure that allows us to create new features easily and produce a functional interface. Lastly, it needs to play nicely with graphing libraries so that we can cleanly display the graphing output we generate from the seasonal weather data to the user.

1.3 Potential Choices

1.3.1 Angular JavaScript

Angular JavaScript is a relatively new web framework built around HTML5, CSS3, and JavaScript that is developed and maintained by Google[1]. Angular JavaScript is not Googles first framework, but it is the most modern so it is highly dependable. Since the debut of this framework it has standardized the way web applications are structured and is used as a model for front-end design[1]. Angular supports easy REST actions, Model View Controller, and Model View View-Model.

1.3.2 React JavaScript

React is the latest and greatest front-end JavaScript library. It is only four years old and quickly becoming the most used front-end technology due to its simplicity and strength in building user interfaces[2]. This strength comes from the fact that you dont have to write separate HTML with react, but instead you describe what you want and React builds the HTML for the designer. Perhaps Reacts biggest strength is its reactive updating. When an input is changed React instantly updates the component without refreshing the page.

1.3.3 Raw JavaScript

JavaScript without additional frameworks is still fully functional for designing a front-end interface and structure and brings several strengths. The first strength is that raw JavaScript designers dont have to spend additional time learning and choosing which framework they want to use. Secondly, getting off the ground and creating the application can be easier due to not having to spend time setting up the application framework which can sometimes be overkill in comparison with the project goals. Lastly expanding the code base once its created will be easier since you wont have to pull in and learn additional frameworks, but this strength only exists if creator is also the maintainer of the wrote the code.

1.4 Discussion

Raw JavaScript's benefit is that we wont have to learn additional frameworks and can immediately write our own code base. However, for this project I believe it is a weakness since the rest of the AgBizTeam will have to maintain our code after we complete the module; so existing within a standardized framework is a good idea. The existing model for AgBizClimate is model view controller which Angular JavaScript was built to support therefore making the product is consistent and reliable. React is a phenomenal front-end technology with no real weaknesses except for not being a self contained framework.

1.5 Conclusion

For the purposes of this project raw JavaScript is not a viable solution for the previously mentioned reasons. React JavaScript and Angular JavaScript dont play well together so its important that we only use one of them for the front-end technology. Since our team is producing a submodule of an existing product we are required to use the technologies the development chose at the start of development; so we will be using Angular JavaScript for our front-end framework. As a side note we will also be using the front-end HTML framework Bootstrap with React.

2 Graphing Frameworks

2.1 Overview

One of the essential and most valuable components of the AgBizClimate submodule is its ability to generate useful and explicit graphs based on user input. These graphs need to be visually pleasing and easy to interpret so that users can quickly ascertain the relevant information they can use to assist their normal decision making process.

2.2 Criteria

The graphing framework that we choose needs to work closely with JavaScript and HTML5 since that is the primary base for our dynamic web application. It also needs to work quickly since we are trying to provide user with rapid feedback based on input and have high responsiveness and feedback to increase overall user satisfaction. Ideally the graph selection we chose will have minimal overhead so integration is quick and easy.

2.3 Potential Choices

2.3.1 Angular Chart.js

Angular chart.js is an open source graphing library for JavaScript. It is responsive and works well with JavaScript and HTML5. It can provide eight different types of charts and can interleave different chart types together [4]. This interleaving process creates singular graphs that can illustrate data differences more clearly. Chart.js is script-able and also supports animations which could assist users in understanding the data[4].

2.3.2 Plotly.js

Plotly.js is a high level JavaScript graphing library that is built on top of d3.js and stack.gl[5]. It provides the ability to chart data with 20 different chart types including 3D graphing, animations, sub-plotting, and mixed plotting[5]. One of the stand out aspects of Plotly.js is its ability to stream data in and dynamically produce graphs.

2.3.3 D3.js

D3.js for data driven documents is a JavaScript library that was created solely to manipulate documents based on data. It is one of the most widely used and popular JavaScript graphing libraries in existence. It works closely with front-end frameworks to produce high quality HTML5 tables and visualizations. D3 is sleek, fast, and effective; allowing high quality graphs to be generated with almost no overhead and assisting in high responsiveness and usability[6].

2.4 Discussion

Chart.JS is an effective open source JavaScript library that would work well for our project. Its primary drawback is the limited number of chart types. With the current AgBizClimate setup this is not an issue, but when the submodule expands it could create problems down the road due to limitations on visualizations. Plotly.js is an extremely powerful JavaScript tool built on top of the other option, D3.js and provides a wide variety of graphing options. However, due to the size and high level of abstraction Plotly.js is slower than the other two graphing libraries with more required overhead. D3.js is the nice middle ground between these three libraries. It's faster with more responsiveness than plotly.js and it provides a wider range of chart types than Charts.js.

2.5 Conclusion

All of the graphing frameworks this document has discussed have their varying strengths and weaknesses, however I believe D3.js is best suited for the job. Since our team is producing a submodule of an existing product we are required to use the technologies the development chose at the start of development; so we will be using Angular Chart.js for our graphing framework.

3 Back-end Designs

3.1 Overview

The Back-end design or software architectural pattern is an essential part of building a functioning web application. Deciding on which model to use will influence design decisions for all parts of the web application including the front-end, and the back-end The following choices for back-end design are the most popular and modern options that are being used today.

3.2 Criteria

The back-end technology has two key requirements in order for it to be successful with the AgBizClimate project. The first requirement is that there needs to be a distinct front-end and back-end. Secondly, it is essential that the front-end is stated and sessioned, so that the application remembers the user inputted steps to reach the decision assistance stage.

3.3 Potential Choices

3.3.1 REST API

REST APIs also known as a representational state transfer application passing interfaces have a clear separation between the client and server which is highly desirable for creating seamless user interfaces[7]. This separation makes products using REST extremely scalable with very little effort. Rest APIs are also useful due to the API itself being separate from the code-base. This means that you can have servers running different languages, but as long as the API is the same then the application will still function.

3.3.2 Model View Controller

The Model View Controller is one of the most basic and widely used back-end architectural patterns. The Model consists of the logic and collection of classes necessary for the web application. The View is what the user sees and where the user interface resides and the controller is what handles requests and passes information between the model and the view[9]. The Model View Controller Paradigms main strength is the separation between the visual components of a web application and the functional back-end. This allows the front-end interface to be altered with no real effect on back-end functionality.

3.3.3 Model view ViewModel

The Model View View-model design pattern is utilized to separate the front-end from the back-end with an integrated ViewModel component. The Model consists of the logic and collection of classes necessary for the web application. The View is what the user sees and where the user interface resides. The ViewModel is responsible for altering the state of the view and manipulating the model with the information that was gained from the altered view[8]. This paradigm allows events to trigger in the view itself.

3.4 Discussion

Each of the previous paradigms have their strengths and weaknesses. The REST API is fantastic for simple queries to the back-end, but it has a harder time tracking progression through multiple steps which is what we need for AgBizClimate. The Model View ViewModel method is great for projects that have long forums and require more dynamic views, but its a bit overboard for our needs on this project. The Model View Controller method allows the designer to have a distinct front-end and back-end while processing data between the components. Maintaining this structures allows the development team to spend very little time working on the front end after it's initial design and completion.

3.5 Conclusion

Since our team is producing a submodule of an existing product we are required to use the technologies the development chose at the start of development; so we will be using the Model View Controller paradigm for our back-end design.

References

[1] Angular JavaScript
`https://www.linkedin.com/pulse/20140613173601-45832080-why-to-choose-angularjs-javascript-framewo`
Pankaj Kumar Jha, June 13th, 2014.

[2] React JavaScript,
`https://medium.freecodecamp.org/yes-react-is-taking-over-front-end-development-the-question-is-wh`
Samer Buna March 30th, 2017.

[3] Raw JavaScript,
`https://www.sitepoint.com/frameworkless-javascript/`. Pawel Zagrobelny.

[4] Chart.js
`http://www.chartjs.org/`.

[5] Plotly.js,
`https://plot.ly/javascript/`.

[6] d3js,
`https://d3js.org/`.

[7] The Representational State Transfer,
`https://www.service-architecture.com/articles` Douglas K Barry.

[8] The MVVM Pattern,
`https://msdn.microsoft.com/en-us/library/hh848246.aspx`

[9] The MVC Pattern,
`https://msdn.microsoft.com/en-us/library/windows/hardware/ff550694(v=vs.85).aspx`

Oregon State University

CS 461

Fall 2017

Tech Review AgBizClimate©

Author:
Thomas Noelcke

Instructor:
D. Kevin McGrath
Kirsten Winters

Abstract

The purpose of this document is to research and consider different technical options for our application. In this document we research different options for data storage, HTTP request frame works, and testing frameworks. I will consider three possible choices for each section of the application. For each of these options I will weight the pros and cons of each. After comparing the different options I will select the option I would like to use for the *AgBizClimate* application.

Contents

1 Data Storage

1.1 Overview

For the *AgBizClimate* application we will need a way to store data so it can be easily retrieved later. For this application we will store a variety of data including budget data, weather data, and user information. This information will need to be quickly recalled so it can be used in our application. Generally, we will want to select a data storage option that will be easy to set up and allow us a lot of flexibility with what kind of data that can be stored. We will also want a data storage option that will quickly recall stored data so it can be used by the application.

1.2 Criteria

To determine the best choice for our application i will analyze the performance of the data storage based on, Ease of Development, and Ease of Set Up and Flexibility. I will analyze run time speed however, this will not be one of the criteria as it is not critical that our data is retrieved as quickly as possible. Generally, its much more important to consider Ease of development, Set up and flexibility because the primary concern is being able to get the application up and running quickly. Ease of development and ease of set up will be a subjective measures for each option. In those sections I will use the opinion of other software developers along with my own experience to compare each option.For these criteria I will rate each option on a scale from very easy to very hard. Flexibility will be a measure of how easy it is to store different kinds of data using each option. For Flexibility I will measure each option from flexible to rigid.

1.3 Potential Choices

1.3.1 PostGreSQL

PostGreSQL is an open-source relational database management system. PostGreSQL uses the Sever Querying Language (SQL). PostGreSQL uses the relational database model. This model sets up tables that represent a certain type of data. We can then run SQL quires on this data base to get the data we need for our application [1].
Though SQL is quick, In my experience it is not as developer friendly as the other options in this analysis. Generally, writing raw SQL queries is difficult and time consuming. This is especially true when your data models are very complicated. Using Raw SQL also requires the developer to manually figure out how to make the mapping between the database and the models used at the application level. With raw SQL it is also more difficult to change the structure of the database once you have created the database. These changes will require potentially complicated scripts along with scripts that convert the data to fit into the new structure [2].
Another problem with SQL in my experience is that it is more difficult to set up. The configuration process can be complicated. Additionally, if you choose to use raw SQL you must also set up your data base as third party application separately from your actual application.
Another problem with SQL is that it is rather rigid in the way that you must store data. For example if you want to store a list object in an SQL data base you must create a new table where one row represents one item in the list. This item must have its own unique id. Additionally, if you want to have a list of lists it gets even more complicated. Now you need to create another table to represent a name and ID for each list you want to store and you must relate every item you want to put in that list back to the parent item in another table. This can get very complicated in a hurry. Another problem is if you don't know what the structure of the data is going to look like before run time it is impossible to store this data in an SQL database. This makes PostGreSQL rather rigid in

terms of flexibility.

1.3.2 Python ORM

Python ORM or Object Relational Mapper does not substitute for an SQL database. There will still need to be an SQL database running on the back end. However, the ORM framework allows developers to create objects in python that then map to the data base. Often times this type of frame work allows the developer to create the objects first and let the framework deal with creating the SQL database on the backend. Given that this is not a replacement for an SQL it does make working with an SQL database much easier.
This type of frame work has several advantages, it allows for easy development and set up. The ORM frame work allows the developer to be completely insulated from the SQL data base on the back end. This means that instead of writing SQL queries to get data from the database the developer is able to use objects in python to access data that is stored in a database. This makes development and set up Easier on the developer. This is because the developer doesn't have to worry about writing complicated SQL statements or setting up complicated relationships between tables. This frame work also allows the developer to develop the code first and let the framework worry about creating the database the data will ultimately be stored in [3]
The ORM framework also allows for great flexibility in what you can store in a database. This type of frame work allows for mappings between python objects and the data base. So nearly anything you can store in an object in python, you can also store in a database. However, it should be noted that you must know the structure the object you are trying to store before run time.
Though this approach is very easy for the developer it isn't with out cost. The ORM approach does take a hit in terms of run time performance. The ORM framework will be slower than raw SQL. Another problem with this type of framework is that it doesnt leave the developer very much control over the database. This means that you are stuck with what you get. If the frame work structures something in the database in a way you don't like you don't have a ton of choice about that. Additionally, making database chances can cause you to loose data in the database if you are not careful about how you handle the migrations [3].

1.3.3 MongoDB

MongoDB is an open source database NoSQL database. This means that instead of using tables and rows like an SQL relational database, MongoDB uses collections and documents. Documents are individual data members where there is a key value associated with each document. Collections contain multiple documents. Collections allow the developer to store many items in a database. This structure allows for the structure of the data being stored to be determined after run time. This allows for greater flexibility because you can store dynamically generated objects[4].
MongoDB is also fairly simple to set up. This is because of the way that objects are stored we don't need to set up and complicated tables. We can simply set up our data base and insert the data. This also means that if we want to change the structure of the data down the road we don't need to make massive changes to the data base. This makes MongoDB much more developer friendly and also very flexible[2].
However, It should be noted that the usability and flexibility of MongoDB is not with out cost. The biggest cost of the flexibility of MongoDB is that it is harder to relate two items in a database. For instance if want to have one entity that represents a user and another that represents a user role, this becomes a challenge in MongoDB. There are work around and ways to solve this problem however it should be mentioned that this is a problem[5]. Another trade off in using MongoDB read and write operations are generally asynchronous. This means that you can tell the database to do something and then move on other tasks and the data base will return the result of your query later.

On the surface this sounds nice but if you are doing a lot of reading and writing operations in the same block of code this can cause problems. For instance if you write several items to the data base and then need to read items out that depend on those items you just added you may get errors because the first write hasn't finished yet[6]. Finally, MongoDB may also be much slower for some operations than traditional SQL. One example where this becomes apparent are aggregate functions where you want to preform some sort of manipulation of the order of the data[7].

1.4 Discussion

As mentioned earlier we can divide the types of data we need to store into two distinct groups, weather data and user data. The user data will generally have relationships between different parts of the data. For instance a user will have an address and a phone number. The user will also have various different budgets. These relationships make the data easier to store in an SQL database such as PostGreSQL. To farther simplify storing the data we could also use Python ORM. This would allow us to create relationships in the data, easily develop the database and store the data. The weather data on the other hand is much more like a large list. This sort of data can be sotred in an SQL data base such as PostGreSQL however, this would make the development of the project more difficult. A more suitable choice for storing the weather data would be MongoDB. This is because MongoDB allows for easy storage of lists with little complexity than PostGreSQL. MongoDB provides the easiest and most flexible way to store the weather data.

1.5 Conclusion

For this project we will use PostGreSQL and MongoDB. We decided to use these frame works because our client requested it. This is because we are adding to our clients system that has already been created. However, the client did put in some good though into these choices. They chose PostGreSQL because its a powerful relation database management system that will work well for the user data. They also chose MongoDB for the weather data because it provides a simple to set up and easy to use storage tool for storing large lists of data.

2 HTTP Request FrameWork

2.1 Overview

The *AgBizClimate* project will need a way for the client on the front end of the application to communicate with the API at the back end of the application. We have already determined that we will use HTTP request to facilitate this communication. However, we have not decided which framework we should use to make the HTTP requests with. Generally, the frame work we use to make HTTP requests should be easy to use, easy to read and allow us to quickly right requests.

2.2 Criteria

For the purposes of this analysis we will consider ease of development, readability and compatibility with AngularJS. We are considering compatibility with AngularJS because our client has required us to use angular. This will be measured on a scale from compatible to uncompilable. Ease of development will be a subjective measure of how hard it will be to develop software with a framework. This will be measured on a scale from easy to very hard. Readability will be a subjective measure of how easy it is to decipher the meaning of the request from the code. This shall be measured on a scale from readable to incomprehensible.

2.3 Potential Choices

2.3.1 jQuery Ajax

When talking about HTTP requests in java script Ajax is probably what immediately comes to most software developers minds. Ajax is one of the original frameworks for making HTTP requests with out updating the page. Ajax allows for asynchronous requests to the backend. Generally, Ajax is not very human readable. Ajax can be read by humans but those humans are going to need to have had experience with Ajax before. It is important to note that Ajax requests will require more work that most of the other HTTP request frameworks we are discussing. Additionally, Ajax can be use with Angular however we will be required to take extra steps to ensure that the requests are handled correctly. If you choose to use this framework with angular you will also need to do more work to be able to test your code.

2.3.2 axios

Axios is a promise based HTTP client for web browsers. Axios will also allow us to make asynchronous calls to the backend of our application. It is also important to note that axios HTTP requests are much more concise making them human readable[8]. Axios can also be integrated into Angular however, this will require some set up. If you choose to use axios with angular you will need to set up your own custom tests because this frame work will not integrate into angular testing framework. Additionally, you may run into headaches down the road when angular updates are made because axios is not maintained by angular.

2.3.3 AngularJS $http

$http is a module in AngularJS that enables communication with remote HTTP servers by using the browsers built in tools. This frame work provides a layer over top of an ajax request that makes the request easier to read and write. These requests are much easier to read and writing the request is much easier that raw Ajax. This HTTP frame work also integrates seamlessly with AngularJS because its a core service. This means that testing these requests with the Angular testing frame work is fairly easy and updates to Angular wont break your HTTP requests [8][9].

2.4 Discussion

In comparing these three frame works I noticed that $http really stood out from the other two frame works. This is because $http is a core service in the AngularJS framework. This makes it much more compatible with Angular than Ajax or axios. Comparing $http and axios out side of the context of Angular they are pretty similar. Both produce much more readable http requests than Ajax and both are are fairly easy to use.

2.5 Conclusion

For this project we will use $http to make http requests between the front end of our application and the back end of our application. We chose to use this framework because it integrates seamlessly with Angular while also making HTTP request readable and easy to develop.

3 Testing Frame Work

3.1 Overview

For the AgBizCliate system we will be required to write unit tests for our code. Unit tests will help us ensure that our application meats the functional requirements. Unit tests will also be helpful for feature projects to ensure that parts of the application have not been broken by a change. In general good unit tests will improve the reliability and extend the life of a project. For this project we want a testing frame work that will allow us to quickly create unit tests in Python for Django Applications. We want to be able to do this because our client has already specified that we will be using Python and Django.

3.2 Criteria

For the purposes of this analysis we will consider ease of development, compatibility with Django, and simplicity of setup. Ease of development is a subjective measure of how easily tests can be developed with each framework. Ease of development will be rated on a scale of easy to hard. This frame work will also need to be compatible with Django as this is the frame work our application will be written in. Finally, we will also consider ease of set up. This is a measure of how easy a framework is to set up so we can begin writing unit tests.

3.3 Potential Choices

3.3.1 Python unittest

Python unittest is a testing frame work built into Pythons standard library. The unittest framework provides the necessary scaffolding to setup, shutdown and run unit tests. This testing framework will feel familiar is it is part of the greater Junit project. This makes writing unit tests feel familiar and fairly easy. However, for this application using this testing frame work would require a lot of set up. This is because this is a web application and unittest does not include tools that make it easier to set up and run tests for web applications. More generally, unittest wills take longer to develop because it provides less scaffolding[10]. This makes unittest less developer friendly. Unit test can easily be used with Django as it is part of the python library. For this reason unittest is also very simple to set up.

3.3.2 py.test

py.test is a popular python testing framework that helps reduce the repletion required in unittest. py.test also contains powerful utilities that make testing many kinds of applications easier. For our project py.test would make testing our web app much simpler than unittest. This would make test development quick and easy. Though this framework makes tests easier to develop that does come at a cost. This framework requires extra setup to integrate with Django. This will make setting up this framework more difficult that either Django.test or unittest. Though the set up isn't complicated it would make testing our application more difficult and more complicated [11].

3.3.3 Django.test

Django.test is a testing frame work that is baked into Django. This frame work provides powerful tools for testing Django applications. This frame work is part of the greater Django framework. Because Django.test is built into the Django framework, there is very little set up. This also means that there will be no compatibility issues with Django either now or in the future. This testing suite also provides powerful utilities that make it easier to develop tests. This frame work also gives us specific tools designed to test Django applications[13].

3.4 Discussion

While considering the options there is one option that fits our application the best which is Django.test. Django.test will allow us to test our Django application with minimal effort as it requires no special set up. Additionally, Django.test also contains tools like py.test that will enable quick and easy development of tests. Though py.test may also be a good option, it doesn't integrate into Django as well as Django.test. Unittest is also another good testing option as it is apart of the python language. However for the purposes of this project unittest would make the development of tests more difficult.

3.5 Conclusion

For this project we will be using Django.test to write unit tests for our application. We chose to do this in part because our client asked us to use this frame work. However, our client asked us to use this frame work with the goals for our application in mind. For this application we want to be able to quickly and easily generate unit tests and Django.test is one of the best tools available for testing Django applications.

4 References

[1] Home, PostgreSQL Tutorial. [Online]. Available: http://www.postgresqltutorial.com/what-is-postgresql/. [Accessed: 22-Nov-2017].
[2] MongoDB vs SQL: Day 1-2, MongoDB. [Online]. Available: https://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2. [Accessed: 22-Nov-2017].
[3] M. Makai, Object-relational mappers (ORMs), Object-relational Mappers (ORMs) - Full Stack Python. [Online]. Available: https://www.fullstackpython.com/object-relational-mappers-orms.html. [Accessed: 22-Nov-2017].
[4] What is MongoDB? - Definition from WhatIs.com, SearchDataManagement. [Online]. Available: http://searchdatamanagement.techtarget.com/definition/MongoDB. [Accessed: 22-Nov-2017].
[5] J. Headley, The Problem with MongoDB Hacker Noon, Hacker Noon, 12-Feb-2017. [Online]. Available: https://hackernoon.com/the-problem-with-mongodb-d255e897b4b. [Accessed: 22-Nov-2017].
[6] T. S. Chief, Potential problems and issues with using MongoDB, Stackchief. [Online]. Available: https://www.stackchief.com/blog/Problems%20with%20MongoDB. [Accessed: 22-Nov-2017].
[7] A. C. Weinberger, Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB, ArangoDB, 13-Oct-2017. [Online]. Available: https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/. [Accessed: 22-Nov-2017].
[8] axios, npm. [Online]. Available: https://www.npmjs.com/package/axios. [Accessed: 22-Nov-2017].
[9] $http, AngularJS. [Online]. Available: https://docs.angularjs.org/api/ng/service/$http. [Accessed: 22-Nov-2017].
[10] unittest vs py.test, Bytes IT Community. [Online]. Available: https://bytes.com/topic/python/answers/43330-unittest-vs-py-test. [Accessed: 22-Nov-2017].
[11] 26.4. unittest - Unit testing framework 26.4. unittest - Unit testing framework Python 3.6.3 documentation. [Online]. Available: https://docs.python.org/3/library/unittest.html. [Accessed: 22-Nov-2017].
[12] C. Maske, The Engine Room, Using pytest with Django - The Engine Room - TrackMaven. [Online]. Available: http://engineroom.trackmaven.com/blog/using-pytest-with-django/. [Accessed:

22-Nov-2017].
[13] Django Tutorial Part 10: Testing a Django web application, Mozilla Developer Network. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing. [Accessed: 22-Nov-2017].

Oregon State University

CS 461

Fall 2017

Tech Review

Author:
Shengpei Yuan

Instructor:
D. Kevin McGrath
Kirsten Winters

Abstract

The purpose of this document is to research different technical options and consider possible choices for our application. In this document, I have researched different options for containers, back end design, and styling framework, and I will analyze three choices for each section of the application. For each option, I will introduce the principle, advantages and disadvantages as it relates to this project. After comparing all three choices, I will choose the appropriate option for the AgBizClimate application.

1 Container

1.1 Overview

Containers are a powerful virtualization technology that is helpful for software development with the cloud environment. Generally speaking, they greatly help environment construction and deployment work for developers and maintainers. Containers also improve efficiency and reduce cost of developing complicated software systems.

1.2 Criteria

Containers are one important technology in cloud computing. It provides independent running environment for various software applications. It is believed that containers may replace virtual machines in the near future as the dominant technology for constructing cloud-computing environments. Basically, containers work on the OS layer and provide runtime environments for programs. The OS distributes computing, memory and peripheral resources for containers, and it owns isolated namespaces for programs.
The performance of containers should be considered carefully as there are substantial differences between computing architectures and their native hosts. Ideally, we would expect little or no reduction on computing or memory access efficiency when using containers relative to the native host. To test whether containers meets this requirement, one feasible method is to examine and compare the specifics of the same application deploying both on containers and native host.

1.3 Potential Choices

1.3.1 LXC

LXC is one of the earliest containers to provide virtual Linux runtime environment for software applications. The core characteristics of LXC are built on the basis of resource management and isolations techniques of the Linux kernel. For instance, the cgroups feature of Linux kernel that divides processes into groups for management is the essential framework for managing resources for processes in LXC.

1.3.2 Docker

Docker is a mature container tool has almost replaced the conventional Linux Containers (LXC) and helped solve the availability problems of many software systems. It is claimed that Docker is the prevailing virtualization technology in todays software building environment. However, with the rapid development of the Docker ecosystem as a tool for cloud-computing it has made constructing other businesses based on it harder. Docker is one of the best components for constructing a software system in virtual computing environments. One perspective is that Docker is or will become an ecosystem for software construction. This will make it vulnerable in many ways, whereas Rocket makes a fresh start and only focus on working as a component to help construct complex software systems. Another perspective is that the core value of Rocket is exactly the start point of Docker, integrating complicated software systems into one independent platform. The founder of Docker, Hykes agrees that Rocket could be one important tool for customizing configurations of containers, and will define the future of containers technology.

1.3.3 Rocket

Rocket is a quite new open-source container technology for software developing created by CoreOS in 2014. It is implemented in Go. As a command-line tool, Rocket works quite like Docker to package software applications and their dependency libraries or systems to a plantable container.

It is known that Rocket would have no compatibility issues with Docker. Also, rocket will bring new ideas for software container technology and relevant markets. It never tries to provide broad friendly functionalities like cloud acceleration tools, and integrated systems. Rocket will become more purified as a standard container tool for the software industry. It has a start point that multiple heterogeneous container platforms could be integrated into. This will help solve the availability problem. Moreover, it would try to alleviate the security issues of Docker. It should be noted that Docker and Rocket both have their own strength and weakness. Generally speaking, Docker would be more competent at complicated and huge systems, while Rocket is a better fit for lightweight and small software applications.

1.4 Conclusion

The vast adoption of Rocket among great software systems proves that it satisfies the requirements of the software world. Rocket is designed to boost the success of software systems, including specific measures like guaranteed security, flexible components and open standards. Different companies may choose different container tools between Docker and Rocket according to the specific contexts of their systems. As a result, for this project, we will adopt Rocket as an independent container library, so that it could be easily integrated into the existing project AgBizClimate and the corresponding environment. In other words, we need a container tool to construct components for our system rather than to create a new system from scratch.

2 Back end Design

2.1 Overview

There are several powerful and popular models for building the back end of a web applications. RESTAPI and MVC are two important technologies among them. Besides, we would also describe a little about MMVC, an extended version of MVC. We will introduce them in the order of when they were proposed in order to help readers to understand them more easily.

2.2 Criteria

The backend architecture technologies mainly solve the problem of decomposing the functionality of the application reasonably and efficiently for software developers. Ideally, all modules have clear bounds between each other and tight connections within themselves. Besides this, it is also important to present the structure and functionality of the application clearly and easily to both developers and end-users.
The performance of the back end architecture is important as once determined they dominate the entire development process of the application. The back end architecture has great impact on the extendability and robustness of the program. It can be hard to test whether a back-end architecture can fully meet our design requirements as it is very abstract high layer design. However, it is commonly believed that a good back-end architecture must make improvements on the efficiency of entire process of system development, function extensibility, code readability, and maintainability.

2.3 Potential Choices

2.3.1 MVC

The MVC (Model View Controller) model was proposed as early as 1970s and used by Smalltalk, and it is still popular and used to develop large software applications in various fields today. Many software languages and frameworks like the prevailing Java Struts and Spring MVC use the MVC architecture. The core concepts behind the MVC model is that no matter how simple or complex a software system is, it could always be stratified into three layers from the perspective of structure. First, the View layer, which lies on the top, is the one directly seen by the end user. It usually works as an interface between users and the program, and is also called the shell of the program. Secondly, the Model layer, which lies in the bottom, usually represents and stores the data or information of the program. For many programs, this layer is the core layer that largely dominates the structure of the other two layers. Thirdly, the Controller layer, which lies in the middle, is mainly responsible for the interactions between the Model and View layers. More specifically, it accepts user instructions from View layer, retrieves data, manipulates the model, and sends back the resulting data to view for to be displayed. The above three layers are tightly connected together as well as being independent. The internal operations within each layer never affect other two layers, and each layer provides appropriate interfaces for the other layers. The essential point of this design is that the entire system could be divided into independent modules so that neither changing appearances nor changing data would alter other two layers. Hence, it greatly reduces the cost of maintaining, upgrading and testing the system.

2.3.2 RESTAPI

RESTAPI (Representational State Transfer) was proposed by Roy T. Fileding in 2000 as a simple and extensive standard for developing web applications. In fact the HTTP protocol is one typical application of the RESTAPI architecture. With the vast applications of cloud computing technology, RESTAPI is very popular used by many software architectures and developers to build large web applications. The most essential features of RESTAPI are resources, unified interface, URI and statelessness. The core concept of RESTAPI is state transfer. More specifically, the network resources and actions are clearly isolated from the state of the application. The transferring of states is described by the URI so that it is very clear for both developers and end-users. By Passing around states from back end to front end, a uniform interface is created by mapping HTTP methods to CRUD operations. The RESTAPIs, front end and back end, are stateless, so that the APIs are very efficient. Consequently, all information, including modifications, are saved and represented by the requesters, and servers contain no state information.

2.3.3 MMVC

The MMVC is an extension of MVC that optimized the Model or data layer. It is essentially the same as MVC except that it adds a view Model between Model layer and View layer. The new layer acts as an interface between Model and View layer since the data models in applications are becoming more and more complicated. We need to know three essential points about MMVC. Firstly, it is compatible with the existing MVC architecture. Secondly, it makes the software applications more testable. Thirdly, it works best with a binding mechanism. It should be mentioned that MMVC is a relatively new back-end architecture technology and is currently not used broadly in the software industry.

2.4 Conclusion

MVC and RESTAPI try to build a powerful general architecture for building various software systems. MVC tries to model the system based on actions, whereas RESTAPI mainly focuses on the data, the sate and transition to different states. MVC emphasizes dividing the internal implementations into three layers Model, View and Controller, and RESAPI decompose the web applications in terms of outside appearance, transitions of states of system. Basically, this project would try to follow RESTAPI standards for back-end design so that the programs are well structured and flexible for extension.

3 Styling framework

3.1 Overview

The styling frameworks are higher level programming languages based on CSS (Cascading Style Sheets) to make CSS (Cascading Style Sheets) development flexible and efficient. There are many popular styling frameworks for building beautiful and consistent looking web applications. We would talk about three popular ones LESS, SASS and Bootstrap.

3.2 Criteria

The core function of styling framework is to allow easier and efficient web UI design using CSS. A good styling framework will have following three features. First, it provides programming ability like logic examination, loops and functions for basic CSS code. Second, it provides rich and practical predefined templates, components, and a suite of themes for quick design of web UIs. Thirdly, it integrates other front-end languages like HTML and Javascript for flexible and easy developing.
The performance of front end styling framework is also important as it largely influences the UI design and system interaction. For instance, a good styling framework would provide rich dynamic UI components with well defined interfaces. To test whether the framework fully meets the requirements, one could compare the specs for the styling of HTML components before and after using relative frameworks. In fact, it is quite easy for testers or end-users to measure the differences between different UIs after using them.

3.3 Potential Choices

3.3.1 LESS

LESS extends CSS and introduces module conceptions into it. It provides much functionality for front-end developers beyond the basic CSS. It has similar grammar rules as basic CSS. The LESS grammar rules are similar to SASS grammar rules. Like SASS, LESS is also an extension of CSS3 that introduces rules, variables, selector and inheritance. It tries to generate well-formatted CSS codes that are easy to organize and maintain.

3.3.2 SASS

For front-end programmers of web applications, SASS provides more powerful functionality than LESS, which works more like a real programming language than LESS. However, for UI designers, LESS seems to be clearer.

3.3.3 Bootstrap

Unlike LESS and SASS, Bootstrap is one comprehensive and powerful front-end framework based on HTML, CSS and JavaScript. Generally speaking, Bootstrap tries to integrate tools like Compass, Blueprint and h5bp. Bootstrap is a comprehensive framework for web front-end development. It should be mentioned that Bootstrap use Normalize.css to reset CSS, which has becoming the practical standard (used more broadly than Eric meyer 2.0 implementation of Compass). It is also compatible with h5bp, meaning you can use h5bp and Bootstrap concurrently in one project. Generally, Bootstrap has three key features. Firstly, it includes the complete basic modules of CSS, although not as powerful as Compass. This make it possible for programmers to use basic CSS attributes for simple customization of HTML elements. Secondly, it provides some suites of predefined CSS, including one grid layout system like blueprint but with a different style. This helps programmers to quickly define the overall look of their web applications. Thirdly, it provides a group of UI components based on jquery like dialogs, navigation menus, and edit boxes. They are all both powerful and ascetically pleasing. This may be the most powerful strength of Bootstrap. In fact, it is becoming a practical standard of most jquery-based web projects.

3.4 Conclusion

Since we are not professional UI designers and work as programmers for the AgbizClimate application We plan to use Bootstrap. Bootstrap will allow us to employ both the powerful programming ability and the rich UI components of Bootstrap without needing much graphic design.

5 Weekly Blog Posts

5.1 Shane Barrantes

5.2 Thomas Noelcke

5.3 Shengpei Yuan

6 Final Poster

7 Project Documentation

7.1 Essential Documents

7.2 Recommended Technical Resources

8 Conclusions and Reflections

8.1 Shane Barrantes

8.2 Thomas Noelcke

8.3 Shengpei Yuan

9 Appendix

9.1 Appendix 1: Essential Code Listings

9.2 Appendix 2: Catch all