PURDUE UNIVERSITY FORT WAYNE
Department of Computer Science
COLLEGE OF ENGINEERING, TECHNOLOGY, AND COMPUTER SCIENCE

The 2018-2019 Do It Best Corp. Dashboard team in a moment of triumph - winning Techapalooza 2019

Integrated Application Development Walkthrough

HOW TO DEVELOP AN INTEGRATED WEB APP – *THE DIGITAL HARVEST WEBSITE*

Max Fowler, Haemin Ryu, Beomjin Kim

Content

1. An Integrated App – *Digital Harvest*

The modern software developer has a need to develop and maintain integrated applications, utilizing a myriad of different technologies and moving parts. As a follow up to the *Integrated Application Development Guidebook* by Bolinger et. al, this document and related source code is a living example a fully integrated application prototype. The document will briefly discuss requirements and design, show off all the steps for the first prototype of *Digital Harvest*, and wrap up with some considerations for testing and lessons learned. By the end of the document, student teams preparing for large scale project implementation will have a comfortable grasp on project organization and integrated programming.

Do It Best Corp. Project Dashboard and Viridian Trails Fort Wayne – two integrated applications developed at PFW

The document distills the experience of three team members highly skilled and familiar with the integrated application development process. Maxwell Fowler served as an advisor to four senior project teams during his time at Purdue Fort Wayne, as well as a number of independent studies, with a high focus towards integrated applications. Haemin Ryu recently completed her senior design project and produced much of the front end code. Finally, since his time as department chair, Beomjin Kim has served as an advisor to several projects each semester as well as an overall organizer of senior design at PFW's Computer Science department.

1.1 What is *Digital Harvest*?

A realistic integrated application development tutorial requires a realistic problem statement. It would serve no good for the integrated application tutorial to serve no practical purpose. Common integrated applications involve API use, at least at the level of data analytics, if not also payment taking APIs. A mobile responsive storefront application makes a perfect, simple sample of the development needs and techniques needed for modern applications. At the same time, a novel storefront serves as a better example as it delivers a more believable capstone project experience. As much fun as the author would have with a storefront for alpacas, such a project would remain contrived. Further, merely creating a knock-off Amazon Prime would not serve as a realistic project experience.

A storefront serving as a digital farmer's market was chosen as the prototype. It is a realistic project for the Midwest given the region's prominence for farming. It is a novel storefront, as it offers a specific kind of goods (produce) with enough variety to make for an interesting database development experience while not having so broad a scope as to be infeasible as a prototype project.

To inform the future sections of the paper, a project proposal was created for the *Web Storefront for Local Food Growers* proposal. The name *Digital Harvest* will be used from this point forward – we will assume it was established early on in a meeting with the sponsor by a clever senior student. The full proposal is given for consideration on page 4.

With the proposal in mind, sections 2 and 3 of the paper will discuss the early meetings your team will have. The team will need to plan well for their first meetings to get strong requirements and design ideas, even if flexible to change, to proceed smoothly during the development process.

Tutorial's Sample Project Proposal

Title	Web Storefront for Local Food Growers
Sponsor	Contact person: Timothy Ackerman Company name: Local Produce Services
	Contact info: tsample@gmail.com
Type	☑ Application development ☐ Information systems ☐ Research-focused
Description	Local Produce Services is a small business aiming to support and digitize farmers' markets and other small, local producers. To that end, LPS wants a modern online storefront which growers may use to sell produce and track purchases. LPS wants the storefront to support the following major uses: 1. Local growers should be able to make a storefront account to advertise their produce and receive orders from customers 2. Local growers should be able to see analytics on how much of their produce is purchased so that they can shift their production based on this information 3. Non-growers should be able to make an account in order to save payment details and address
Team size	☐ 2 ☑ 3 ☑ 4 ☐ > 4
Required backgrounds	Web application development, database management
Required resources (HW/SW)	A server for the database and web application,
Other notes	LPS would like the application developed to be mobile responsive, so that people may use their phone to access the website if they do not want to use a computer

2. Project Requirements

2.1 What to plan for the first sponsor meeting

Your team has chosen to work on the online farmer's market application. The sponsor is new to the department, so not much is known about project expectations from the start. Setting up an early meeting is crucial to getting development off on the right foot and keeping the team happy and productive.

Broadly speaking, the first meeting with a sponsor should focus on the following areas of interest:

- Future meeting flow. Will the sponsor be hands on or hands off? Do they want to be available to the team frequently or only meet a few times? Try to establish this flow early.
- Proposal agreement. While most teams use some form of Agile development and some flexibility in the project scope, requirements, and design is expected, agreeing on the core proposal prevents the worst of unanticipated change or scope creep.
- Project technology. This is a **significant** decision most teams overlook. Often times neither students nor sponsors know precisely the best technology or approach for an application, especially sponsors who are non-technical. Prepare to ask questions to find out what the best approach is. Some projects are best handled with a native mobile app that connects to a server, while others may be cross-platform or a mobile responsive website. Further, try to meet the sponsors' requests if possible. A sponsor who wants a cross-platform mobile solution without needing an iOS developer's account wants a mobile responsive site: don't decide to code it in Swift just because you prefer Mac OS!
- Broad requirements. As most teams use Agile, it is unlikely the team will have fully fleshed out requirements on day 1. Further, sponsors cannot be guaranteed to be as available for requirement grooming as some Agile methodologies require. Try to get the big ticket requirement items down such that the team can later refine them into some form of user story or requirements document. Offer to keep this document live and updated, sharing it with the sponsor and team so everyone knows how the project is progressing.

2.2 Moving from a description to requirements

The first meeting, or perhaps the first few, should see the team with some sort of direction on the project. The proposal is agreed upon and a contract is signed between the team and sponsor. However, the project may still not have formally defined, implementation level requirements. Considering the description from the proposal on page 4, a team might produce the following high level requirements with the sponsor:

1. Local growers should be able to manage their own products
2. Local growers should be able to access and ship their sales
3. Local growers should have analytics on their sales
4. User accounts should be able to buy produce
5. User accounts should be able to save payment details
6. User accounts should be able to keep a shopping cart to buy multiple items

While these requirements are a bit lower level than the three points our sponsor Timothy asked for, these do not constitute requirements that can be directly implemented as a single task. Consider requirement (1). This requirement alone requires that growers be able to make an account, identify as a grower rather than just a purchaser, upload (and assumedly remove) listings for their produce, and be able to set information like prices and quantity available at bare minimum. Those points could be further broken down; making an account will require a registration page, setting up a user table in a database, user authentication security, confirmation email support, and other related development items. The question becomes what level of granularity requirements should be broken down to so the team can track their progress.

Providing an *exhaustive* list of requirements is beyond the document's scope, as is providing a breakdown of different kinds of requirement formats. At a high level, most requirements are written at a level that can be implemented and are given some sort of acceptance criteria. We will use an informal user story format to demonstrate the breakdown of a couple of requirements into tasks for the sake of example, but this process will need to be adapted by teams based on their software development process and individual needs[1]. Many user stories are written in a card format, so a few sample cards are shown on the following page. All of the stories shown in this document will be two sided cards. The front will contain a story in the form: *As a <user role>, I want <function> such that <result/goal>*. The back of the card will list some acceptance criteria.

[1] Max Fowler recommends Scrum using teams skim *Essential Scrum* by Rubin. The book is a great Scrum reference for any level of developer or project manager.

User story breakdown for making an account (and some associated considerations)

Front	Back
As a non-grower user, I want to be able to make a user account on Digital Harvest such that I may buy locally grown goods.	Users can select a non-growing account type at sign up. Users can create an account on an account creation page.
As a grower user, I want to be able to make a grower account on Digital Harvest such that I may sell my locally grown goods.	Growers can select a growing account type at sign up. Growers can create an account on an account creation page.
As any type of user, I want to be able to receive a confirmation email when I make an account such that I confirm my account exists and known I have a recovery email.	Users should be sent a confirmation email upon successful registration to the website. User accounts should store the email as a recovery method.
As any type of user, I want to be able to receive a password reset email such that I may change my password and access my account should I forget.	The system should support password reset emails when users request them. Password reset emails should be secure and passwords should never be provided in plaintext.

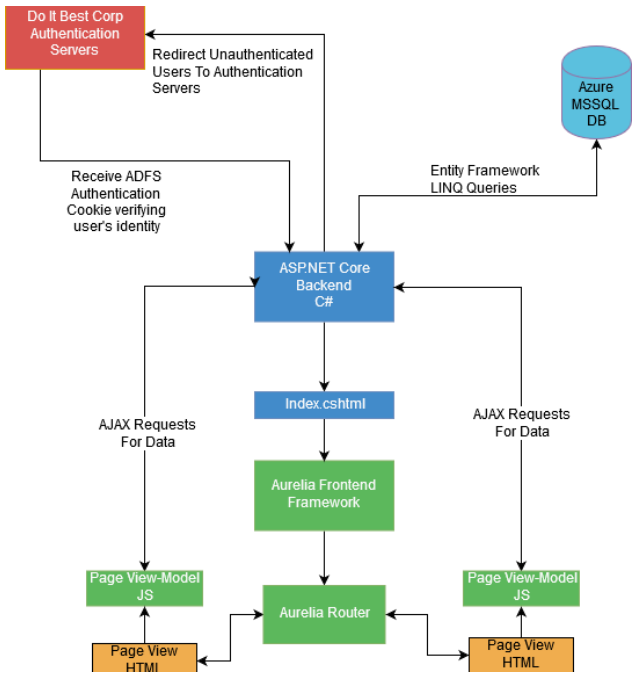
Notice that, even broken down to the user roles and email behavior, these requirements are not exhaustive. Many user stories are broken down further into development tasks to be performed to support the story. These tasks are not always considered stories in themselves and different management frameworks present them differently, from task checklists attached to user stories to task items managed in a scheduling system. Below is one sample task breakdown for creating user accounts.

Story	Tasks
As a non-grower user, I want to be able to make a user account on Digital Harvest such that I may buy locally grown goods.	○ New user account page is created ○ New user form can be posted to database upon completion ○ User database table exists ○ User creation supports role selection – non-grower and grower ○ New user page has input validation

These samples should help your team get off the ground. In the next section, we will discuss the high-level design the prototype will use.

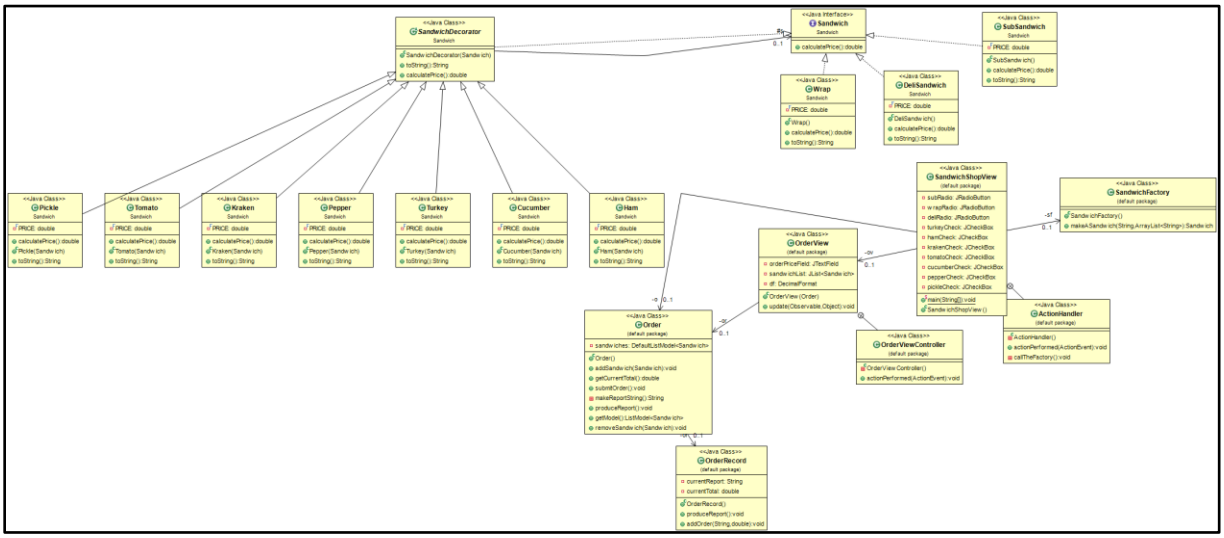
3. High-Level Application Architecture Design

Based on your development model, you will eventually hit the need to consider your application's design. Many Agile using student teams tend to skip considering design in favor of a more "cowboy coding," oriented approach. Nothing could be more disastrous to a team's success than failing to consider how the moving parts of a large-scale integrated application will function together[2]. Most students working on large-scale development will have at least been exposed to the Unified Modeling Language (UML) and have familiarity with some of the document types, such as class diagrams and sequence diagrams. However, it is also true that a number of tools exist to generate these at a reasonable level from existing code, which can make upfront development of fine-tuned supporting documents feel like an excessive chore. The solution to this feeling is to develop high level architecture and module diagrams early on. As always, these are living documents. No one iteration of the diagram should take a significant time, but incremental changes and updates during each development cycle will make both development and final documentation validation a smoother process.



2018-2019 Do It Best Corp. team's architecture diagram. Good enough for government work and most advisors

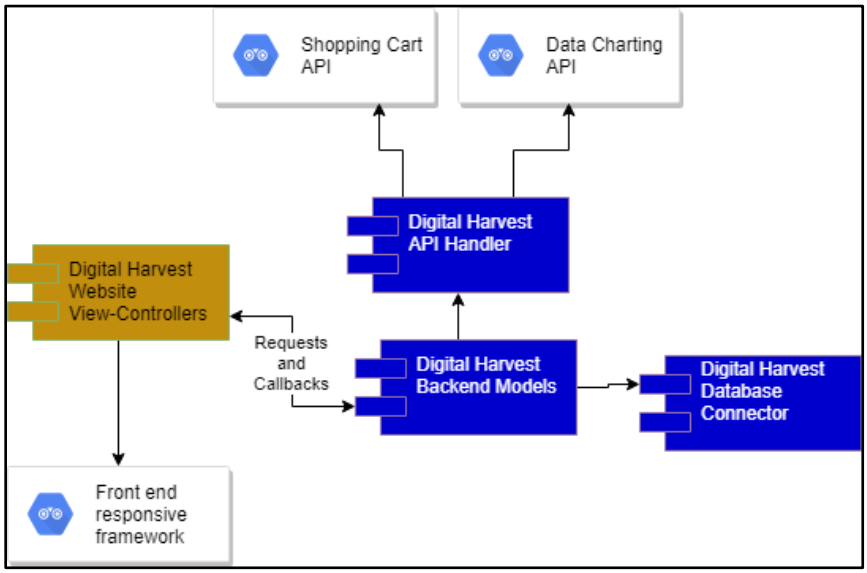
[2] Author Max Fowler knows this painfully well – his first web app hosted from a Raspberry Pi with its own internal Wi-Fi took around 70 hours due to a total disregard for planning. Even those of us with degrees make these mistakes!



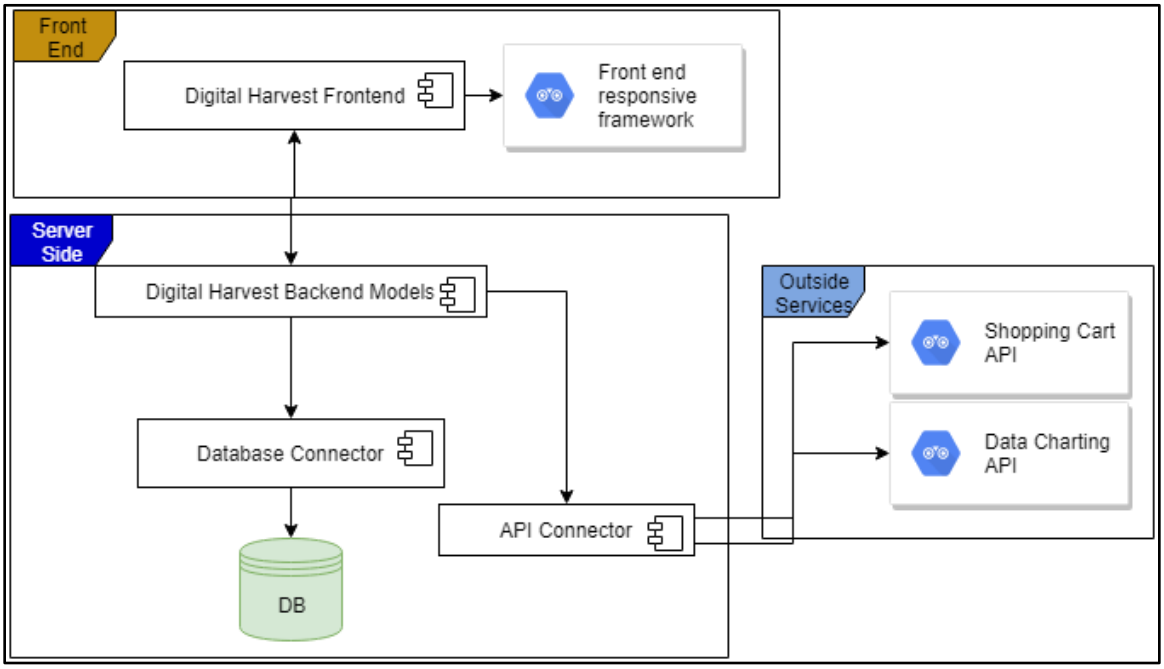
A UML diagram for Fowler's CS161 project "Sandwich Shop Point of Sale System" generated using the ObjectAid UML Explorer for the Eclipse IDE

3.1 First basic documents – module/component and architecture diagrams

Two of the most basic forms of diagrams for a software product are component diagrams and architecture diagrams. Component diagrams are generally defined as diagrams to show component level interactions. In this case, component refers to a module of classes forming a system or portion of an application [1]. This is often why the terms module and component diagram are conflated. Architecture diagrams are less clearly defined, as the style and use of the diagram can vary based on team needs. System architecture diagrams serve system architects as a planning tool, website architectures focus on the hierarchy of a website's flow and development stack, and application diagrams can change heavily based on the template used and level of detail desired [2]. Full discussion of these topics is best left for a course on project management or software engineering. Instead, the walkthrough provides two high level examples of what would pass for planning design documentation for *Digital Harvest*. To keep the burden on teams interested in replicating the style low, draw.io, a free online drawing tool, was used. At this level of fidelity, the diagrams are notably similar. The module diagram removes some of the hardware and non-code related components.



A simple module diagram for Digital Harvest



A simple architecture diagram for Digital Harvest

Notice the above are at a high level of simplicity and abstraction. These are not production level design documents, but will serve as a good planning tool that only took a small amount of time to organize. It would be useful to further break down the module diagram into component parts, to see which view-controllers on the front end speak to what parts of the backend, but that level of fidelity is too low granularity for this document.

3.2 Decisions Made (and Deferred) During Design

Notice that the above diagrams feature a major decision that was made: the use of APIs and frameworks. Not all components in a large-scale integrated application need to be coded from scratch. On the contrary, large-scale development largely relies on knowing what design patterns to apply and when to use APIs to avoid reinventing well tested wheels. Specifically, the above documents identify three potential APIs or frameworks. Two are on the backend for shopping cart and data visualization. On the front end, some form of responsive framework will be used to help with responsiveness on multiple types of screens, including mobile. Simply knowing where these APIs fit in early will help drive development and save time.

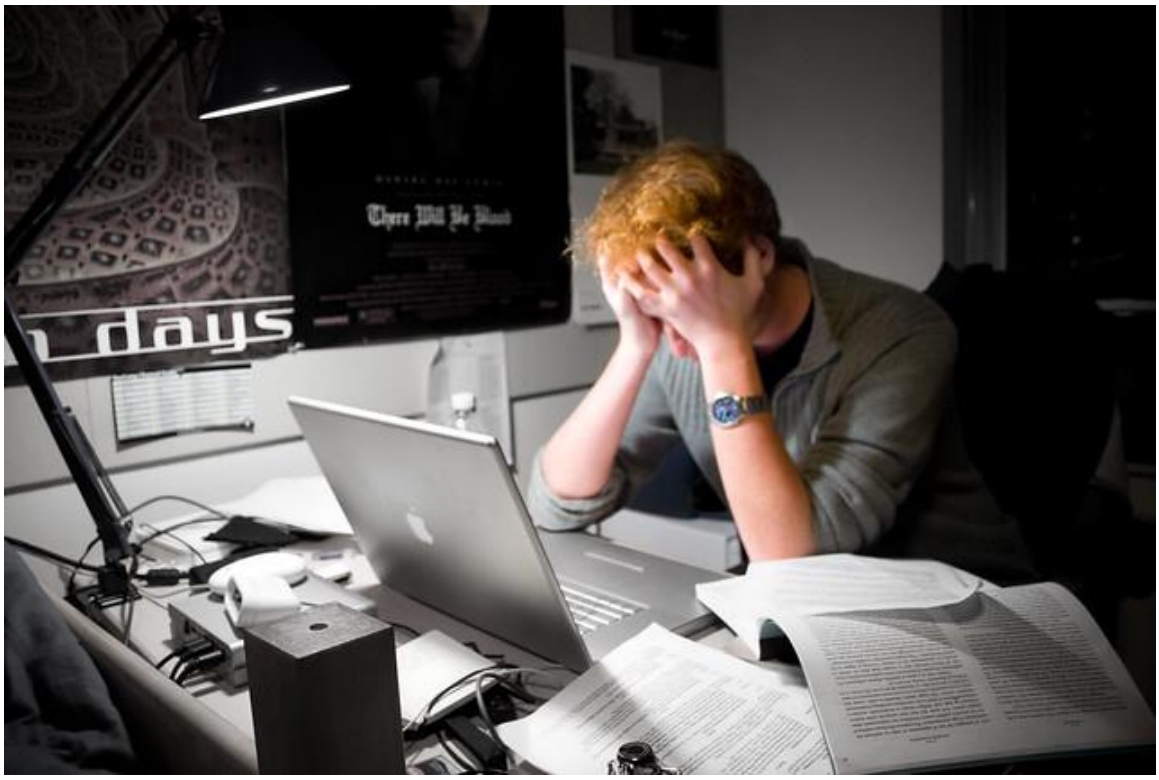
It is also important to note what decisions were not specifically made here, to instead be deferred. For one, a breakdown of the modules into smaller components was avoided – likely, the backend module will be a small family of modules, with at least one working with the charting API and some working with general database interaction as pages are made and use cases fleshed out. Further, the APIs have not yet been specified. An important part of development is researching potential API options, which at this stage of the prototype's development had not been fully completed.

A third note to remember, especially in Agile, is that this document is not sent in stone. Over time, the diagrams may be fleshed out to show more fine-tuned modules. More diagrams may be added. More APIs could be added as well, such as Google's Firebase to handle secure user authentication for us. The important part of the early design steps is having a document down on paper so the group has general consensus and a direction. As the application matures, refining diagrams and adding new diagrams to represent the state of the project is natural.

With the project requirements and design discussed at a high level, we can move on to an integral part of the process that most groups do not consider; developer operations. Specifically, the next section will deal with setting up developer environments, servers, basic databases, and a project source repository in order to ease the development burden.

4. The Developer Environment

With the requirements elucidated and the design sketched out, your team may be tempted to jump right into coding. At last, you can put fingers to keyboards and blaze a trail to success!



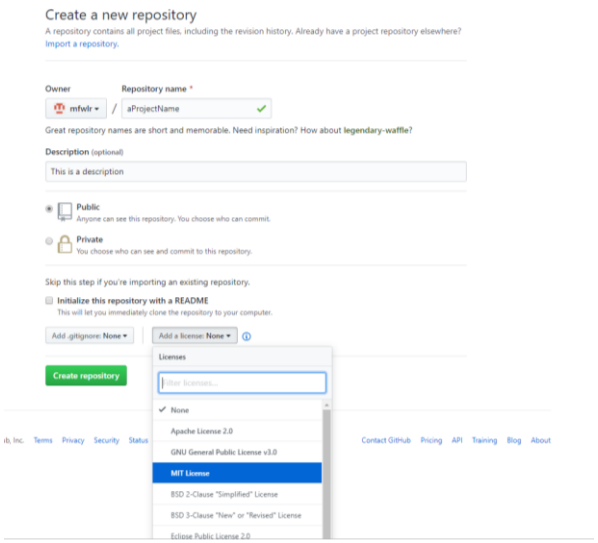
Or fall into the pit of missing semicolons, broken dependencies, and despair. Your choice. [3]

However, before you begin programming, it is important to have a consistent environment set up for the team. Specifically, there are two major items the team should decide upon. The first is the Integrated Developer Environment (IDE) the team wants to use. Based on your project, you could be spoiled for choice or be limited. Java based work has myriad IDE options, from Eclipse to IntelliJ, while Unity programming is going to mostly be confined to Unity's own software and Android Studio remains the typical choice for Android development. Secondly, the team must select a method of version control. Ideally, this will be a tool developed specifically for software versioning and distributed programming, such as GitHub, Bitbucket, or Subversion. Try to avoid the DropBox/Google Drive approach to version control[3]. In this section, we will discuss our team's choice of source control, GitHub, and IDE, Visual Studio Code.

[3] Author Max Fowler had a team do this. Keeping on the same software version was an absolute nightmare.

6.1 GitHub Set Up

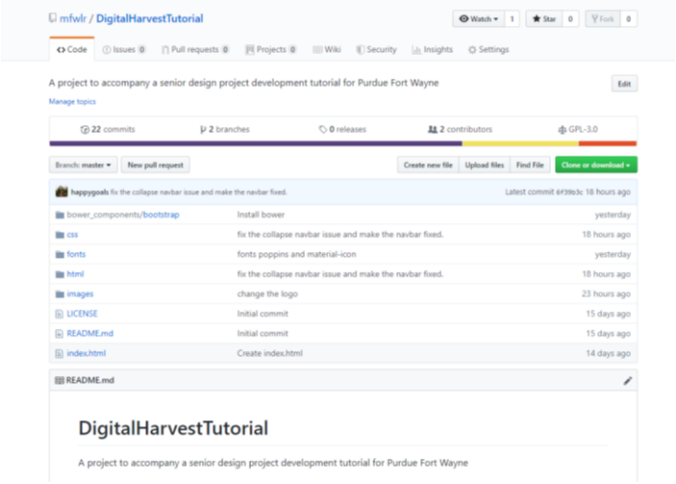
GitHub was selected as our source control method for a few reasons. Firstly, GitHub and other GitHub like tools (e.g. Atlassian's Bitbucket) are a firm industry standard. Second, GitHub also serves as a convenient digital resume of projects, making it a great place to host and show off the *Digital Harvest* prototype. Making an account is simple, so we will jump to the next step of setting up a project repository. On the repository tab of your GitHub account, hitting the green "+New" button will take you to the repository setup page. In general, it is advised to pick a name that is representative of the project and to initialize the repo with a beginning README document. Your team may also want to consider the license the software is under. *Digital Harvest* is licensed under GNU GPL v 3.0 as the code itself is not particularly in need of copyright protection. The license your team uses will most likely depend on your client, any NDAs you have signed, and other project details[4].



The view when creating a new repository.

Once the GitHub has been created, you will have a view of the project as shown in the following screenshot. You will want to invite your whole team by clicking on the contributors tab and adding their GitHub accounts as members of the project. Notice that *Digital Harvest* currently has two contributors - Max Fowler, as mfwlr, and Haemin Ryu, as happygoals.

[4] BitBucket allows for small teams to have private repositories

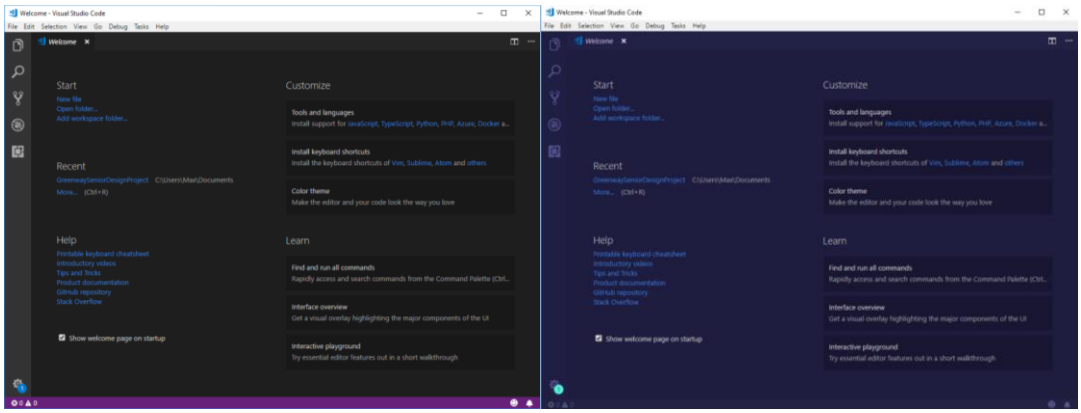


The Digital Harvest GitHub on July 24, 2019.

It is assumed that the teams reading this document have heard of Git before, although may not have used Git on a project before. To ease the usage of GitHub, we will use Git integration in our IDE of choice, Visual Studio Code. A general overview of Git based source control can be followed over at TutorialsPoint's Git tutorial [4]. As specific GitHub commands are needed in this guidebook, they will be introduced, but keep a tutorial or command reference on hand during your development process as a safety net.

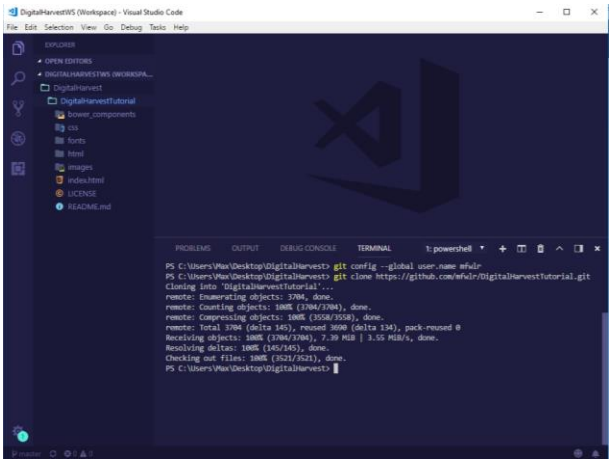
6.2 Visual Studio Code Set Up

With our GitHub ready, downloading our IDE is the next step. Visual Studio Code was chosen for the large number of plugins available, the easy GitHub integration, and wide cross-language support. Another good choice for cross-language, integrated development projects is IntelliJ, if you prefer that IDE. When Visual Studio Code is first installed and launched, your team will be greeted by a view like this. We will be using 'Add Workspace' in a moment to set up a working folder for our application, but first I suggest playing around with the IDE settings. At bare minimum, find a color profile that works for you and your monitor. Eye comfort is an important part of successful development! You may also want to pre-install plugins for the languages you know you will be writing in. For Digital Harvest, we preinstalled PHP, JavaScript, and other web stack plugins.



The default VS Code dark theme (left) vs Night Rider (right)

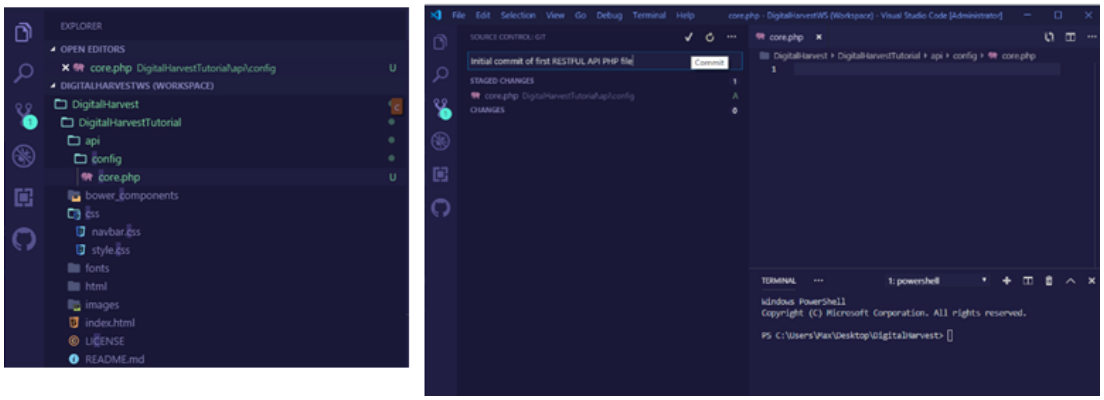
Once you are happy with your environment, use 'Add a workspace folder' to set up a new VS Code project. Once you have a workspace for your project, navigate to the plugin installer to download the official *GitHub Pull Requests* plugin. We will use this plugin to run our GitHub commands. In order to use it, though, we need to set up our project as a GitHub project. This is an easy process: select the Terminal in the View menu and then enter the two commands shown in the screenshot below. The first sets your GitHub username for the project, while the second clones your project repo. Be sure to use the https link for your repo if you want to avoid ssh key usage.



All of the code you saw earlier, now in VS Code

As a final step, to show off *GitHub Pull Requests* usage, we are going to push a small update - adding a "core.php" file for the Restful API we will write later on. We placed it in an api/config directory, giving us a project layout as shown. Further, our source control tab is now updated. We can easily add a commit message and push our change up to our GitHub account. Make sure to stage the change with the "+" button, add a message, and commit. Once all your changes are commited, type "git push" into the terminal and your changes will appear on your GitHub repository.

As we wrap this section up, a word of caution about git pull and git push. Be sure to run git pull before *any* new code is written and before *any* push attempt. This is the best way to avoid painful merge conflicts that you may run into when multiple team members are working on the application.



Project structure and commit window

5. Database Deployment

At this stage, your team is finally ready to implement a part of the project. We will start off with the beating heart of your project, and any integrated web application – the database. How your team chooses to store data and what data the application needs will inform many of the design decisions moving forward with the project. Databases can also be implemented in a number of different ways, from using a more traditional MySql or SQL Server implementation to relying on APIs such as Google's Firebase. In this section, we first discuss the high-level needs for *Digital Harvest's* database, then present both a SQL style database design, and finally discuss deployment of that database to a server.

5.1 Data Architecting – what does the database need?

Any integrated application will require certain typical data be stored. Practically all integrated applications have some sort user account. Further, integrated applications usually have some kind of data linked to that account. In games, this may be account unlocks. In calendar and meeting applications, this could include data such as address books and meetings. As *Digital Harvest* is a prototype for an online farmer's market, there are data needs common to storefronts as well as domain specific data needs. When first planning out a database, a helpful starting place is a list of high-level data requirements. For example, see the following list:

- User account data
- User card/payment data
- Products on the storefront
- User order data

The list naturally needs to be broken down. For example, user account data includes a number of attributes. There are basic requirements that are obvious: a user name, email, and password. Most databases also feature some id field to use as a *primary key* – the attribute which uniquely identifies the entity in a database. This may lead us to use a relation such as the following for our accounts:

ID	User Name	Email	Password

However, as this is a storefront, the user account will likely also require address information for shipping. Further, as the storefront supports both *suppliers* of product and *purchasers* or *consumers* of product, the database needs to consider user roles. One way to structure such a record is as follows:

ID	User Name	Email	Password	Address	State	Zip code	User Role

At a high level, let us consider the requirements for the other three bullet points above. Note that these might end up factoring into multiple parts of a future database.

THE USER CARD AND PAYMENT DATA

This particular relation is one that we can mostly skip. As we have already decided to use a shopping cart API, we will rely on that API to handle the payment details. The biggest benefit of this is allowing a third-party service to handle compliance with credit card storage laws. All merchants are required to follow the Payment Card Industry Data Security Standard (PCI DSS) when it comes to storing and handling credit cards [5]. An example commercial solution for storing data is the PaySimple Solution used by PaySimple, which promises the following [6]:

- Data is encrypted at all times other than transaction transmission
- CVV2 data is never stored
- Swipe data is never stored

We will keep PCI DSS compliance in mind when it comes to selecting our shopping cart API. For now, though, we will take for granted that merchants and the services they use must be PCI DSS compliant and leave that particular piece of data architecting to the API.

PRODUCT ON THE STOREFRONT

Product storage is a relatively straight forward item, especially if we put a (reasonable) burden on the seller to list product details. Products should have an image, a type (e.g. fruit or vegetable), a name, a description, and a cost. Naturally, they must also be attached to the user who is selling them. We can structure a product record as follows:

Product ID	Seller ID	Product Name	Type	Image	Description	Cost

This will be sufficient for a prototype application. However, it is important to note that *sufficiency* does not mean *perfect* nor *acceptable to a client*. It is quite possible that fields would be needed to add units of measurement, e.g. bushels or pounds, as well as the magnitude of the measure. We will skip this for the prototype and assume that the seller will include such information in the product description.

One field we should not skip is another of prime convenience for a seller – quantity. A seller likely will have multiple items that are the same in terms of all of their attributes. For example, an orchard may sell multiple bushels of green apples and granny smith apples a season. As such, we can add a quantity field that will reduce by one each time an order is placed:

Product ID	Seller ID	Product Name	Product Image	Description	Cost	Quantity

USER ORDER DATA

The storage for user orders is perhaps one of the most complicated pieces of data storage. Naturally, an order requires two users: a customer and a supplier. The order also requires at least one product to be sold, if not multiple products. It follows that there should be a shipping address involved in the order as well as a status for that order. We can use a record such as:

Order ID	Buyer ID	Seller ID	[Product IDs]	Shipping Address	Status

Notice that this record is lacking a field that would be nice to have: the total cost of the order. We specifically avoid having this in the record as that data can easily be calculated on the fly. Rather than store redundant data, we can fetch the cost from all products in the order using their product ID and then calculate the cost. In general, it is preferred to calculate fields that are created using other field's data to avoid *stale data*, which are fields that are out of date when the fields from which they are derived are updated. In practice, it is unlikely that the total cost would become stale once an order is placed, as the prices for the items will be unlikely to be updated once an order is placed.

However, with this record we run into the same problem we did with our first design for products, as orders may have multiple orders for the same product. We may instead wish to style orders as two records. One record is the order itself, while the other record is a specific item in that order and the quantity being ordered. We will all this an, "order item," as shown below:

Order ID	Buyer ID	Seller ID	Shipping Address	Status

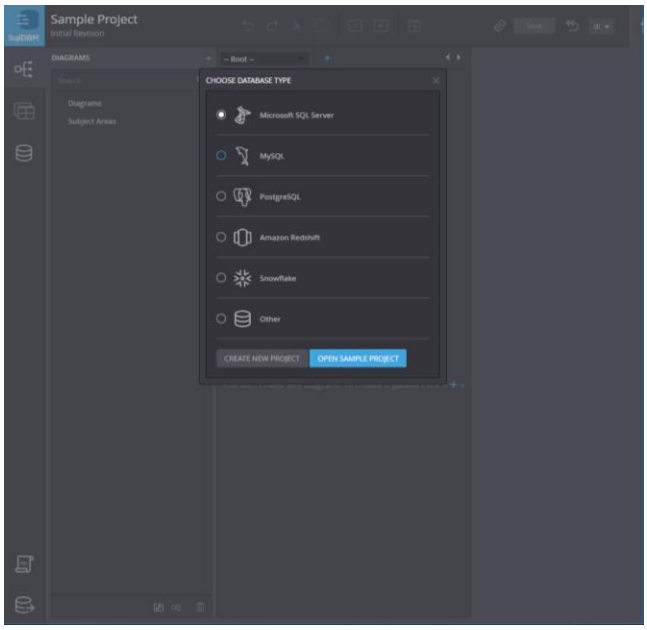
Order Item ID	Order ID	Product ID	Quantity

The robustness of this design is preferable, as it eases the burden of keeping track of a list of products by linking products to an order using the Order ID and allows us to manage quantities of items in an order in a simple way.

With this brief discussion on record format finished, we will move on to how we would represent these records as a SQL database.

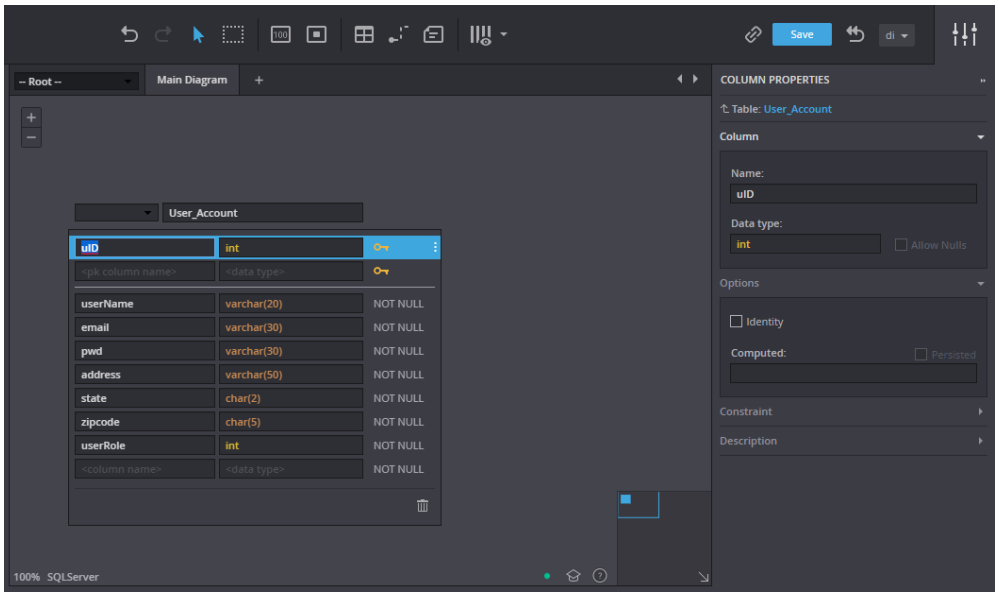
5.2 A SQL database for *Digital Harvest*

When starting to design a SQL database, one needs to choose a tool to draw up the database schema. Common commercial tools include Microsoft Visio and Vertabelo. We could also use Draw.io or LucidChart, with the former being free and the latter supporting both free and paid options. Finally, we could always use a built-in table editor in a SQL tool, including MySql Workbench for MySql or DB Browser for SQLite. However, one tool with a free tier that showed promise as SqlDBM, a database schema generator that allows for SQL generation, collaboration, and importing of databases from a number of SQL styles [7]. The free tier does limit us to a single active project, with only three versions in our revision history and limited SQL generation. If your project is heavily database driven, you may find it useful to pay for the $15 tier, which allows unlimited projects, revision access, and SQL generation! After creating an account and selecting a free tier, we set up a MySQL database to use as our sample for this prototype.



The view on SqlDBM when creating a new project

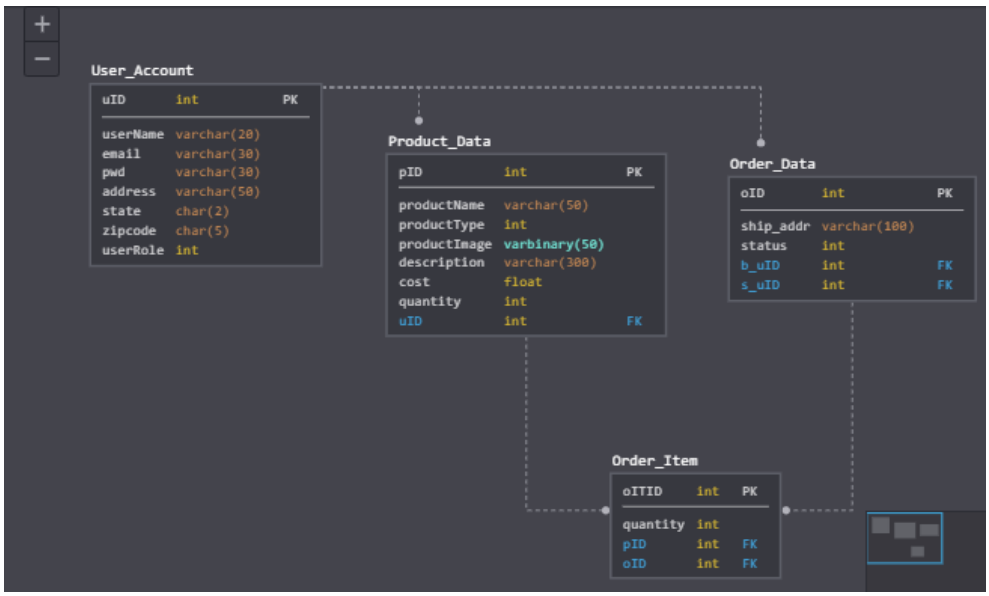
We will start off by creating a table for user accounts. Based on our earlier discussion of what a user record would contain, making the table is quite easy. SqlDBM allows us to edit both the name of fields and the data type, as well as easily set the primary key. For the most part, our fields for this table can be varchars. Integers are used for small fields, such as user role and the user ID.



The User_Account table

The rest of the database design mostly follows from our discussion of the record format from the first section.

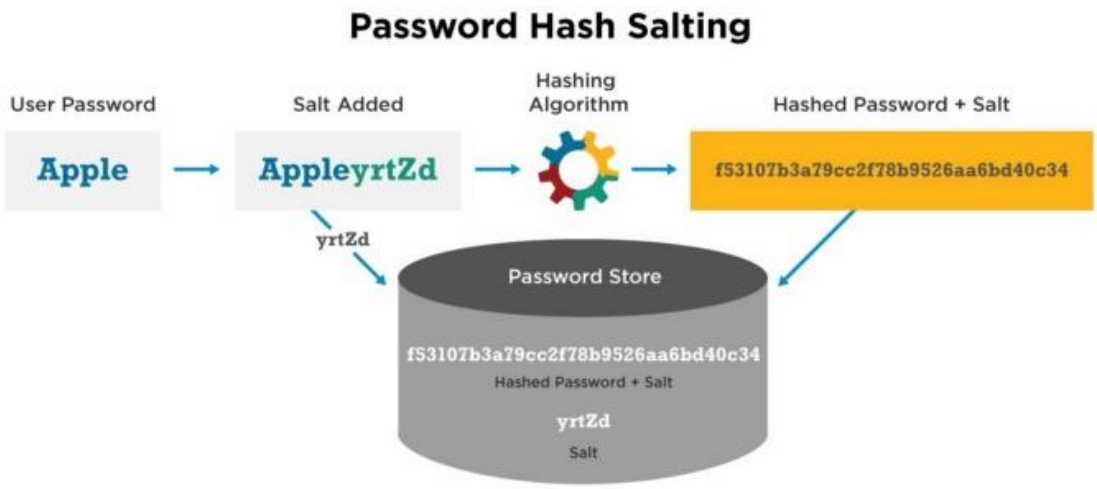
It is important to note a few details in the design. The first detail to note is the use of foreign keys. In the full schema shown below, foreign keys are identified in blue. For the most part, these match the primary keys in name. In Order_Data, as two users are involved in an order, b_uID is used to indicate the buyer's ID and s_uID is the seller.



The full schema for Digital Harvest

The second detail is the data type for the product image. In the schema above, a varbinary is used to store the image as a blob. This may not be the best design, choice, as the best practice depends upon the file size of the image. Gray's Microsoft research paper, *To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem*, provides a few general guidelines for choosing to blob data versus saving the data on a server and accessing it via a URL. In general, data at 256 KB or below can be easily stored as a blob, while data over 1 MB should be stored in a filesystem and accessed via a URL. Between that, storage age of the data and read:write ratios are key deciding factors [8]. A full discussion of the paper's results is out of scope for this tutorial and the exact decision made will depend on your server availability, file size, and project needs. We used a varbinary here for simplicity.

The third note on the database's design has more to do with the future implementation of user login. Passwords should never be stored as plaintext. A plaintext password is a disaster for a potential data breach, as malicious actors can use plaintext passwords to easily steal peoples' accounts. A common way to handle this is hashing a password with a *salt*, which obfuscates the original password's text. Chathuranga's article on password hashing for in .NET Core provides a nice diagram of the basics of password salting.



The diagram used by Chaturanga for password salting [9]

In a general sense, if your team chooses to implement your own database and own login procedure, please follow security best practices! If you are writing your database access code in PHP, a common language to use for SQL database access, it is trivial to add salt support. PHP password_hash() function automatically salts passwords for you when the hashing function is applied [10].

```
$hash = password_hash($password, PASSWORD_DEFAULT);
```

The SqlDBM tool allows us to generate the SQL code to create our database, saving us some more time. However, the free tier only allows us to generate two tables at a time. This is not a huge restriction, but it does mean we will need to generate the SQL in a couple of pieces. Also note that the SQL statements used here are for MySQL. Different SQL using DBMS may use slightly different syntax for their SQL! The next page will have the SQL code for our table creation. Section 5.3 will then show the deployment process of the database on a Debian Linux server.

MYSQL TABLE CREATION STATEMENTS

```
-- ****************** SqlDBM: MySQL ******************;
-- ***************************************************;


-- ************************************ `User_Account`

CREATE TABLE `User_Account`
(
 `uID`      int NOT NULL ,
 `userName` varchar(20) NOT NULL ,
 `email`    varchar(30) NOT NULL ,
 `pwd`      varchar(30) NOT NULL ,
 `address`  varchar(50) NOT NULL ,
 `state`    char(2) NOT NULL ,
 `zipcode`  char(5) NOT NULL ,
 `userRole` int NOT NULL ,

PRIMARY KEY (`uID`)
);

-- ************************************ `Product_Data`

CREATE TABLE `Product_Data`
(
 `pID`          int NOT NULL ,
 `productName`  varchar(50) NOT NULL ,
 `productImage` varbinary(50) NOT NULL ,
 `description`  varchar(300) NOT NULL ,
 `cost`         float NOT NULL ,
 `quantity`     int NOT NULL ,
 `uID`          int NOT NULL ,

PRIMARY KEY (`pID`),
KEY `fkIdx_23` (`uID`),
CONSTRAINT `FK_23` FOREIGN KEY `fkIdx_23` (`uID`) REFERENCES
`User_Account` (`uID`)
);

-- ************************************ `Order_Data`

CREATE TABLE `Order_Data`
(
 `oID`       int NOT NULL ,
 `ship_addr` varchar(100) NOT NULL ,
 `status`    int NOT NULL ,
 `b_uID`     int NOT NULL ,
 `s_uID`     int NOT NULL ,

PRIMARY KEY (`oID`),
KEY `fkIdx_32` (`b_uID`),
```

```
CONSTRAINT `FK_32` FOREIGN KEY `fkIdx_32` (`b_uID`) REFERENCES
`User_Account` (`uID`),
KEY `fkIdx_35` (`s_uID`),
CONSTRAINT `FK_35` FOREIGN KEY `fkIdx_35` (`s_uID`) REFERENCES
`User_Account` (`uID`)
);

-- ********************************** `Order_Item`

CREATE TABLE `Order_Item`
(
 `oITID`    int NOT NULL ,
 `quantity` int NOT NULL ,
 `pID`      int NOT NULL ,
 `oID`      int NOT NULL ,

PRIMARY KEY (`oITID`),
KEY `fkIdx_41` (`pID`),
CONSTRAINT `FK_41` FOREIGN KEY `fkIdx_41` (`pID`) REFERENCES
`Product_Data` (`pID`),
KEY `fkIdx_44` (`oID`),
CONSTRAINT `FK_44` FOREIGN KEY `fkIdx_44` (`oID`) REFERENCES
`Order_Data` (`oID`)
);
```

5.3 Deploying a database to a server

Deploying a database to a server can vary based on the kind of server your team uses. For this tutorial, our team used a Debian GNU/Linux 9 box. As we are using a MySql database, that naturally means we will be using phpMyAdmin to set up out database. phpMyAdmin requires that a LAMP (Linux, Apache, MySql, PHP) stack be set up on the server. In order to do this, assuming you have an account with super user privileges[5], the following terminal commands will suffice:

- sudo apt-get update
- sudo apt-get install apache2
- sudo apt-get install mysql-server
- mysql_secure_installation
- sudo apt-get install php

Once you are finished running these commands, you can begin the phpMyAdmin install process with sudo apt-get install -y phpmyadmin. When the configuration process begins, please use the apache2 server. The phpMyAdmin configuration process will carry your team through a number of steps, including setting up the user account for phpMyAdmin. Be sure to keep track of these passwords and steps, such that you do not lock yourself out of your database. Finally, you will need the php *mcrypt* module and will need to restart the apache2 server. The following commands will handle this process:

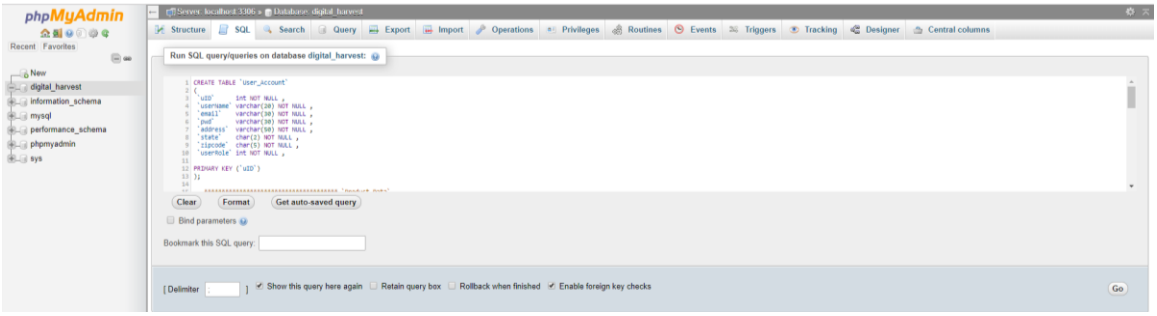
- php -m | grep mcrypt
- If needed:
    - sudo apt install php-dev libmcrypt-dev php-pear
    - sudo pecl channel-update pecl.php.net
    - sudo apt-get install php-pear pkg-config libbson-1.0 libmongoc-1.0-0 php-xml php7.0-xml php-dev
    - sudo pecl install mcrypt-1.0.2[6]
    - Add extension=mcrypt.so to the php.ini
- sudo service apache2 restart

With that, you will arrive at a successful install of phpMyAdmin on your server! You can find it at localhost/phpmyadmin or using your server name (Myserver/phpmyadmin).

[5] "sudo" stands for "superuser do"
[6] You may also need to update your PHP version – this process is best Googled for your particular server setup

phpMyAdmin's login screen in all it's glory

Once you are in the phpMyAdmin console, you will want to first create a new database for your project and then add the tables we generated using SqlDBM. We can copy and paste the generated SQL in without any fuss, assuming we generated the SQL as MySql compliant. With this step complete, we can now program a connector to our backend database to run on the server.



Using the 'Go' button will run the SQL and create our database tables.

6: The Backend Data API

When working with a MySql database on a server, one of the most common languages to use as a connector is PHP: Hypertext Preprocessor (PHP is a recursive acronym). Older data-driven websites would often embed their PHP and web page HTML into the same files. The PHP code would run when a page was first loaded which allowed pages to perform processing and load in data from a server. However, embedding all of the PHP needed for a modern web application into the web pages directly is bad form for a number of reasons:

- Embedding business logic inside of files predominately for display creates a highly coupled, messy architecture
- Future fixes and modification becomes more difficult, as the PHP code is spread out in a number of files across the system
- Embedding your PHP is not a good way to modularize code. A fair number of web applications follow the Model-View-Controller (MVC) design pattern. While *Digital Harvest* is not *strictly* MVC, we still strive to follow good practices.

Luckily, PHP can be used to write REST APIs. We will use PHP to write a RESTful API for *Digital Harvest*, which will allow our front end to access the database in a more modern and asynchronous fashion.

6.1 What is a REST API?

You should already be familiar with application programming interfaces (APIs). APIs are the bundles of protocols, functions, and tools used to build software. Some examples of APIs include OpenGL for graphics support and reqres.in, a REST API used for testing AJAX[7] requests. Many web APIs are RESTful in their design.

REST stands for Representational State Transfer. Developed by Roy Fielding in 2000, it was designed for distributed web and media systems [11]. The restfulapi.net website provides a good overview of the guiding principles of REST. In general, REST is made to support client-server applications while reducing the need for server states and providing uniformity for API design. Information is abstracted into resources, removing the need for front end pages to deal directly with the table structure on a server.

The full breadth of REST APIs, and proper REST coding, is out of scope of this document. However, to provide a simple REST API for *Digital Harvest*, we adapted two helpful

[7] More on AJAX later on

tutorials, *How To Create A Simple REST API in PHP? Step By Step Guide!* and *REST API Authentication Example in PHP – JWT Tutorial* from Mike Dalisay [12], [13].

6.2 API Basics – Connection

The tutorial recommends a basic series of directories to divide up the API code. There is a directory for the API itself which holds a config directory, a directory for the object types, a directory for any needed library support, a directory for shared resources, and directories for each of the endpoint CRUD (create, read, update, delete) functions that we need to implement. Note that this guidebook will not exhaustively discuss every file created, although all the files will be located publicly on the project GitHub.

The first file needed is the database connection file itself. This file will go into the config directory as *database.php*. The code from the file is shown below. This is fairly basic object-oriented PHP[8]. With the connection code written, we will now work on the data.

```php
<?php
class Database{

    //The digital harvest credentials
    private $host = "localhost";
    private $db_name = "digitalharvestdb";
    private $username = "root";
    private $password = "";
    public $conn;
    // get the database connection
    public function getConnection(){
        $this->conn = null;
        try{
            $this->conn = new PDO("mysql:host=" . $this->host . ";
           dbname="    . $this->db_name, $this->username,
           $this->password);
            $this->conn->exec("set names utf8");
        }catch(PDOException $exception){
            echo "Connection error: " . $exception->getMessage();
        }
        return $this->conn;
    }
}
?>
```

[8] At this point, we switched over to an XAMPP install for development. This way, we keep any passwords off of the GitHub.

6.3 Following the Tutorial – ProductData Object

The tutorial linked above goes through the process of creating the file structure to support products. As products are accessible resources that represent some kind of item, we can use the same general setup for our product_data, order_item, and order_data relations from the *Digital Harvest* database. In this section, we will first present the product_data.php file. Note that each object file has a connection variable and table name, as well as fields for all of the records stored in the table in the database. Thus, each instance of a ProductData object is itself a record in the product_data table. Further, each object will have functions related to the queries the object needs. At this stage, for example, ProductData has a read function.

```php
<?php
class ProductData{

    // database connection and table name
    private $conn;
    private $table_name = "product_data";

    // object properties
    public $pID;
    public $productName;
    public $productImage;
    public $description;
    public $cost;
    public $quantity;
    public $uID;

    // constructor with $db as database connection
    public function __construct($db){
        $this->conn = $db;
    }

    function read(){

        // select all query
        $query = "SELECT * FROM " . $this->table_name;

        // prepare query statement
        $stmt = $this->conn->prepare($query);

        // execute query
        $stmt->execute();
```

```
        return $stmt;
    }
}
?>
```

Following this, we need to create an endpoint for product related tasks. This will be in its own product_data folder in the code base. Each endpoint can have up to four files, one for each of the CRUD operations. We will start with the reading code, which will allow *Digital Harvest* to pull and display all of the uploaded products. This is helpful for the main page of the *Digital Harvest* website.

```
<?php
// required headers
header("Access-Control-Allow-Origin: *");
header("Content-Type: application/json; charset=UTF-8");

//includes for other files
include_once '../config/database.php';
include_once '../objects/product.php';

//Set up database connection
$database = new Database();
$db = $database->getConnection();

// Create a ProductData to hold the information
$theProduct = new ProductData($db);

// Get all the products available
$stmt = $theProduct->read();
$num = $stmt->rowCount();

// If we have products
if($num>0){

    // products array
    $allProducts=array();
    $allProducts["records"]=array();

    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)){
        // extract row
        // this will make $row['name'] to
        // just $name only
        extract($row);
```

```
        $aProduct=array(
            "pID" => $pID,
            "productName" => $productName,
            "productImage" => $productImage,
            "description" => html_entity_decode($description),
            "cost" => $cost,
            "quantity" => $quantity,
            "uID" => $uID
        );

        array_push($allProducts["records"], $aProduct);
    }

    // set response code - 200 OK
    http_response_code(200);

    // Use  json format so that REST API use is easy
    echo json_encode($products_arr);
}

else{

    // set response code - 404 Not found
    http_response_code(404);

    // tell the user no products found
    echo json_encode(
        array("message" => "The Harvest is empty - come back soon.")
    );
}
```

References

[1] *Component Diagram Tutorial*. [Online]. Available: https://www.lucidchart.com/pages/uml-component-diagram

[2] *Architecture Diagram Overview*. [Online]. Available: https://www.edrawsoft.com/architecture-diagram.php. [Accessed: 09-Jul-2019]

[3] P. A. Hess, *Frustration*. 2008 [Online]. Available: https://www.flickr.com/photos/peterhess/2976755407/. [Accessed: 24-Jul-2019]

[4] "Git Tutorial," *www.tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/git/. [Accessed: 24-Jul-2019]

[5] "Official PCI Security Standards Council Site - Verify PCI Compliance, Download Data Security and Credit Card Security Standards." [Online]. Available: https://www.pcisecuritystandards.org/pci_security/maintaining_payment_security. [Accessed: 22-Jul-2019]

[6] "Security," *PaySimple*. [Online]. Available: https://paysimple.com/security. [Accessed: 22-Jul-2019]

[7] "SqlDBM - Online Database Modeler." [Online]. Available: https://sqldbm.com/Home/. [Accessed: 22-Jul-2019]

[8] J. Gray, "To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem," Apr. 2006 [Online]. Available: https://www.microsoft.com/en-us/research/publication/to-blob-or-not-to-blob-large-object-storage-in-a-database-or-a-filesystem/. [Accessed: 23-Jul-2019]

[9] N. Chathuranga, ."NET Core 3.0 (Preview 4) Web API Authentication from Scratch (Part 2): Password Hashing.," *Medium*, 15-May-2019. [Online]. Available: https://medium.com/developer-diary/net-core-3-0-preview-4-web-api-authentication-from-scratch-part-2-password-hashing-7e43b64cbe25. [Accessed: 23-Jul-2019]

[10] "Hashing Passwords with the PHP 5.5 Password Hashing API," *SitePoint*. 16-Sep-2013 [Online]. Available: https://www.sitepoint.com/hashing-passwords-php-5-5-password-hashing-api/. [Accessed: 23-Jul-2019]

[11] *What is REST – Learn to create timeless REST APIs*. [Online]. Available: https://restfulapi.net/. [Accessed: 26-Aug-2019]

[12] "How To Create A Simple REST API in PHP - Step By Step Guide!" [Online]. Available: https://www.codeofaninja.com/2017/02/create-simple-rest-api-in-php.html. [Accessed: 25-Aug-2019]

[13] "REST API Authentication Example in PHP - JWT Tutorial -." [Online]. Available: https://www.codeofaninja.com/2018/09/rest-api-authentication-example-php-jwt-tutorial.html. [Accessed: 25-Aug-2019]