ArtistTree Technical Report

The Code Artists:

- Stefan Grasu
- Kyle Wiley
- Jesse Wright
- Walter Sagehorn
- James Graham
- Carson Moore

Table Of Contents:

Introduction:

Problem:

Art is a hard subject to get a grasp of. There are a huge variety of artists, pieces, periods and styles to get an idea about if a person wants to say that know a bit about art. Presently, the best way to do this is to either take an art history class, (a fate worse than death for some STEM students,) or visit a museum, (which is fun but inconvenient). Our website makes art more accessible for the layperson by organizing it into intuitive groupings that help a user get an understanding of periods, places, people, and works of art that they might be interested in.

Use Cases:

Use Case:	View an artist, work, style, or century
Primary Actor:	User
Preconditions:	There are enough artists, works, styles, and centuries in our web pages so that they can be accessed
Postconditions:	n/a
Basic Flow:	1. The user begins on the splash page 2. The user navigates to the model page for one of the four models

	3. The user employs a grid layout on that model page to select one instance of that model that they're interested in 4. The user is able to see information about a specific model (the attributes) 5. The user can then follow a connection to another model from the instance page

Use Case:	A user can sort/filter the instances on a model's page
Primary Actor:	User
Preconditions:	There are enough artists, works, styles, and centuries in our web pages so that they can be accessed, and they have attributes they can be sorted by
Postconditions:	The underlying data is not affected
Basic Flow:	1. A user arrives on the splash page 2. The user navigates to the model page 3. The user has a list of sorting/filtering options they can select 4. Any filtering or sorting options can be applied through a simple UI interaction

Design:

The Rijksmuseum API:

The API we're drawing from (http://rijksmuseum.github.io/), gives us a lot of ways we can look at the artworks in the Rijksmuseum collection. All of their calls give back fairly large responses packed with data about the art and the people who made them. In fact, we added the colors attribute to our "work" model because the API provides us that data, and we thought that would be interesting information to incorporate into the website. The API is actually a little more extensive than we need for our website, as it provides a number of hooks that allow uploading art to their user pages and allow the retrieval of the museum's events calendar. For our purposes, we stick to the calls that have to do with the art and artists in their collection, **not** on their user pages.

The ArtistTree API:

Phase 1:

Our API as of this phase, is pretty basic. We went ahead and decided that we would want to give the option to retrieve *all* models of a specific type, because we think our packaging of the data is quite useful. In looking at the documentation for the API we're drawing from, we determined that we needed to really play with the data to make it fit nicely into our models — models that we chose because they provide an intuitive arrangement of the data. Because of that, when we provide endpoints for our

data we wanted to lean on the strength of our models to provide the entirety of our data to our users, should they choose to retrieve all of it.

Phase 2:

When we spent some time looking at our API, we determined that for the most part, it was pretty alright as it stood at the end of last phase. Our data is really — for the most part anyways — a polite and logical repacking of the Rijksmuseum data, and our API makes it available in a much more digestible format than we found it in.

Searching:

Additionally, there is the option to search for specific artists and artworks, with a number of parameters that you can search by. For this phase, we decided to omit any discussion of searching centuries or styles, as those models rely heavily on data that will be computed from our database, which hasn't been implemented yet. When we do implement the database, we'll update the API to include more search options for centuries and styles, as well as adding more granular search options for artists and artworks. (I imagine our endpoints will change as well — those are just a guess for now.)

As for the actual how of searching: for any search request, you MUST provide one of the parameters that you can search by, but any additional fields are optional. The search will attempt to narrow down the results using all the fields you provide, so if one of the fields is invalid or nothing comes back for that field, we'll search using the other fields. We combine search results across all fields given, so if you include a specific name and a set of colors, you'll get the artwork matching that name and all artworks that have similar colors.

Tools:

Front End:

Phase 1:

For our front end we're just using Bootstrap for now, along with basic HTML and CSS. In the next phase, we'll be adding components from either Angular or React, but for now we're not deciding which one to use over the other yet. Because Angular 2 is in Typescript, and not Javascript, we weren't sure we wanted to commit to using it. Even though several members of our group have used Angular 1.X before, we aren't sure how much of our knowledge will transfer to Angular 2. React, on the other hand, uses Javascript, which we as a group are more familiar with. We'll decide as we move forward whether we want to learn a new language, or a new framework.

In the future we'll use Angular/React to serve the common elements on pages, as well as turn the static model pages into served templates based on the models. Angular/React will make it easy to take data from our database and pipe it through to the instance pages on demand. Additionally we might lean on Angular/React to improve carousels, layouts, and the multimedia elements of our instance pages.

The only elements of our front end that are shared across the site for now are the navbar and the colors. These were standardized in our CSS and HTML, and used wherever they were needed. In the future, there will be more shared elements, but for now these are all we needed.

Home:

Bootstrap made it easy to make the carousel home page, and our subject being art made our website really pop from the start. Having the paintings cleanly fill up the whole page does require that they be treated as background images in CSS within the carousel, but once they are the final result looks excellent. The navbar was aligned absolutely, because otherwise it pushed the carousel off of the page. Customising the navbar was lengthier than we expected, as it required the creation of a custom class specifically to hold the CSS attributes for our navbar, (of which there were many.) In the future, we'll update the home page to scroll through more paintings, or maybe a random selection of paintings out of our database.

About:

The about page was fairly straightforward, and makes extensive use of Bootstrap classes to keep the layout clean. Specifically, the img-circle class made the professional picture arrangement extremely simple to implement. Everything is laid out sequentially using a basic HTML arrangement.

Models Pages:

For now, the model pages are pretty straightforward grid views. The pages for each of the models display the information that we have for each model in a simple grid of information. The pages that display all the instances of a model type are basic grids of the available instances. In the future, we're going to add some customization to these pages, as well as make them serve from templates. This will reduce the amount of files we need, as well as make the pages more dynamic and play more nicely with our database.

Phase 2:

Changes for phase 2:

For phase two, we moved to a dynamic model for loading pages. Each page that we want to load now gets a template built using Jinja and the Django web framework, and is served via a route using Flask. Flask is the part that handles the links, (so if a user were to navigate to artistree.me/artists for example, Flask resolves that link into a page,) and Jinja and Django make it so that we can use object-oriented principles and basic looping in our page templates.

Flask makes it easy to do things like implement our API and serve pages with logical and easy to remember URLs. Parameter passing is easy too: any parameters that represented data from our backend would normally have to be requested via javascript or ES6, retrieved and returned from the backend, and then inserted into the page. With Flask and Jinja, all we have to do is request the data we know we're going to need inside the route for a particular page — think the artist objects for the /artists/ route — and pass it as a parameter to the page. Then, in our HTML template file for that particular page, we can assume there will be a parameter matching the name of the one sent via flask, and we can draw any and all data out of that object that we want.

Page templating makes things a lot simpler in other ways as well. Now we only have one page per model that renders all the data for each model as it's requested, dramatically reducing the number and complexity of files. Another useful thing that templating allows us to do in this phase is inherit from other HTML pages/templates. For example, we only wrote our nav bar *once*, and we import it across every page that needs a navbar with only a simple extends statement. That way, when we update the

navbar, it changes on every page automatically — representing a huge step up in modularity from before.

Project structure also changed for our front-end code. In phase one, we had every page being served out of the “static” folder, as all of it was unchanging and developed separately. For phase two, we moved all the template files to their respective folder inside our main folder (idb), and moved all styling related code to static CSS files in our static folder. Now, our project directory more directly reflects the modularity of our code.

Lessons Learned on Front End:

One major lesson we learned and wanted to note was in our usage of React. React is a useful library, and has a lot of very useful features. For this project, we probably could have gotten by with Jinja and Flask, but the requirement was to use React. Our assumption was that this would be an easy integration, even if we thought it was superfluous — after all, React’s documentation says that we can integrate it with a small portion of our application to test whether or not React works for our use case. This assumption was false.

React, like Angular, is designed to facilitate and make easy web design focused around single-page applications. When we went to integrate React, we made a couple of mistakes related to that:

1. We didn’t begin reading the React documentation early enough, and made assumptions about our use of React.
2. Because we didn’t leave ourselves enough time to understand how we *should* use React, we weren’t able to take advantage of its full potential.

If we had looked at the React documentation ahead of time, we would have seen that React was designed to work as a single-page application using React for the majority of its the front-end code, using Flask only to retrieve data it could not produce. We would have seen that Jinja, although *possible* to use as our front-end templating system, doesn’t work *with* React, it works *in place of* many of React’s

features. Jinja actively makes our job harder, and integrating React into what is essentially a legacy codebase is a use case that *exists* but is neither recommended by Facebook themselves, or the authors of this report.

Given extra time and the benefit of hindsight, we would have begun phase 2 as early as possible, recreating our application as a single-page app that uses React for its front-end, and Flask for its backend. Our code would be cleaner, (we didn't use ES6,) and it would work better than it does right now. Our entire application would function more appropriately.

All of this is not to say that our application isn't good — we're proud of our work — but programmatically speaking, we faced many unnecessary challenges that could have been avoided by starting early and reading documentation in advance.

Back End:

Phase 1:

The back end currently consists of basic routing via Flask. Flask routing simply serves static resources at the moment, namely HTML pages, but we plan for it to use Jinja2 to serve more dynamic and powerful templated pages in the future. The database side of things consists of a simple SQLite3 database (generated from database.py). The models and table schema for the database are defined in accordance with our UML diagram below and are represented through SQLAlchemy (specifically Flask-SQLAlchemy) models using SQLAlchemy's declarative method. At the moment, no model class has a function; this is intentional (at the moment), as all cross-model references are handled through association tables.

Phase 2:

Our backend didn't change much for phase two, besides the addition of more routes and the serving of new templates dynamically. At least during the testing of our front-end, we had to massage the data from the test database a bit, but that wasn't any great challenge — it amounted to making sure all the strings/numbers the front end retrieves are in the proper format.

Database:

Testing:

For writing our SQLAlchemy code and testing our front end, the fabulous and talented Jesse Wright created a test database in SQLite that held auto-generated test data so that we could check for corner and degenerate cases.

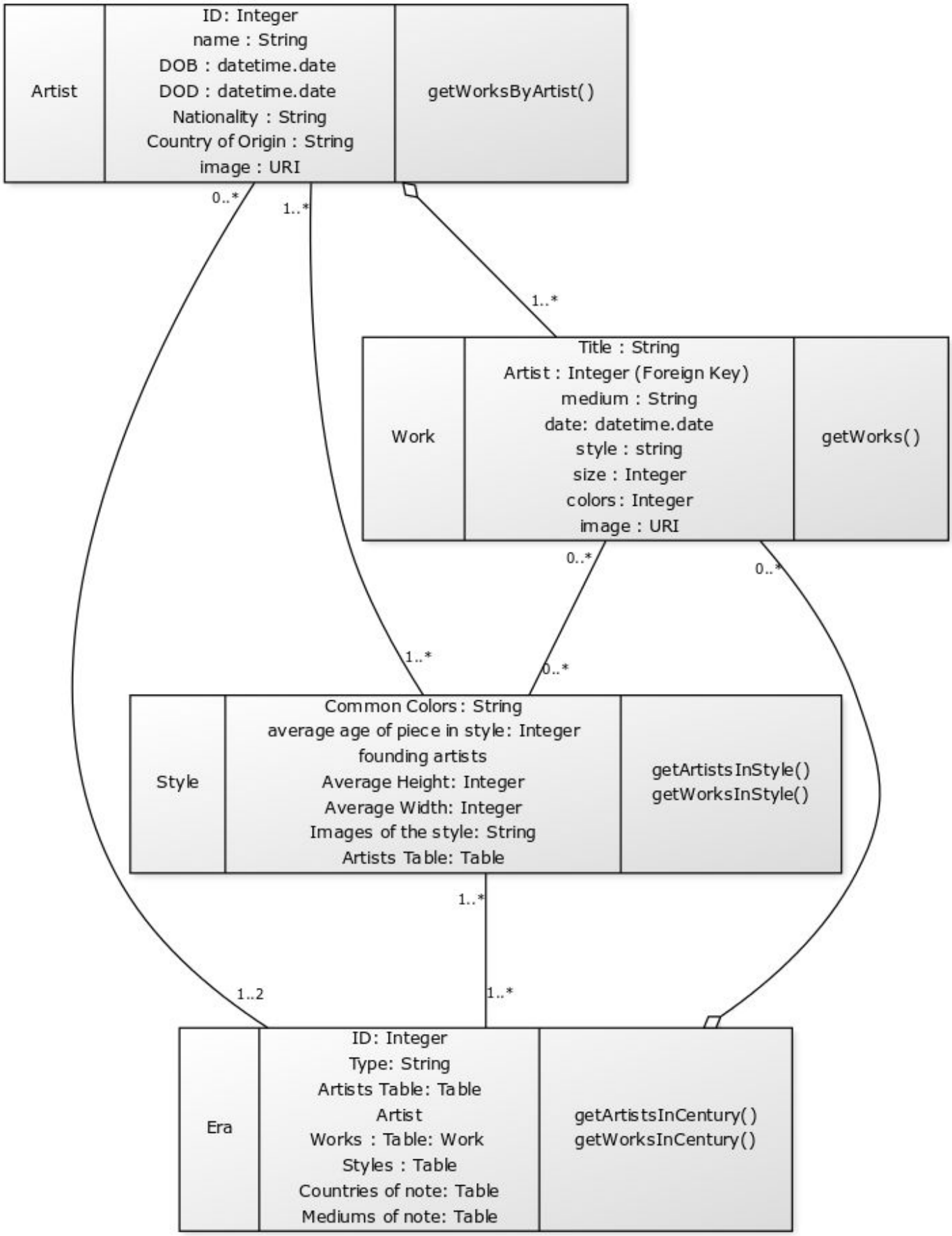
As far as the connections between the database and our front-end code go (both for testing and for production,) things were actually pretty easy. Because of SQLAlchemy making the *actual* source of the data opaque to our front end/routing code, there was no need to write different code for our different databases throughout development; instead, we simply wrote queries, and when it was time to switch to the production database, we changed the parameter that SQLAlchemy uses to find our database. Instead of querying a local test DB, it just redirected the same SQL Queries to our Google Cloud database.

Production:

After going back and forth for awhile about how we would actually implement the database, we settled on using GCloud's internal database system for serving it. GCloud makes the creation of a database straightforward, but when you want to actually use that database, there are a few difficulties involved:

Firstly, we needed to switch to the GCloud Python "flexible" environment instead of the "static" environment, which meant some changes in files like requirements.txt to get GCloud to configure our application properly. (Flex environment, as best as we can determine, manages the number of VM's running our app and other behind the scenes things like that.)

Secondly, we had to create a local PostgreSQL database before we were able to create a cloud version. This is mainly because of the limitations of GCloud's database import, which only wants to accept database files from a storage bucket inside of Google Cloud Storage. It wasn't a technical challenge — we just made a local database with the data we scraped and then dropped it on GCloud — but it was a logistical hurdle we had to clear.


Artist
ID: Integer
name : String
DOB : datetime.date
DOD : datetime.date
Nationality : String
Country of Origin : String
image : URI
getWorksByArtist()
0..*
1..*
1..*
Work
Title : String
Artist : Integer (Foreign Key)
medium : String
date: datetime.date
style : string
size : Integer
colors: Integer
image : URI
getWorks()
0..*
0..*
1..*
0..*
Style
Common Colors: String
average age of piece in style: Integer
founding artists
Average Height: Integer
Average Width: Integer
Images of the style: String
Artists Table: Table
getArtistsInStyle()
getWorksInStyle()
1..*
1..2
1..*
Era
ID: Integer
Type: String
Artists Table: Table
Artist
Works : Table: Work
Styles : Table
Countries of note: Table
Mediums of note: Table
getArtistsInCentury()
getWorksInCentury()

Hosting:

For our hosting we chose to go with Google Cloud Platform, mainly due to the familiarity of our group members with Google technologies and the free credits given to us by the Google cloud talk. Thankfully, Google Cloud Platform makes the hosting and running of our website fairly straightforward. One thing to note in particular about Google Cloud Platform is that it has a *lot* of components. For the purposes of this project, you're only going to care about an extreme subset of those options and unfortunately Google hasn't seen fit to organize the web development tools in a way that helps cs373 students. We've written the guide below to help explain how one hosts a website, based on our experience.

Hosting a project on GCP for cs373:

Preparing your website (and if necessary, your wallet):

The first step in preparing your website for deployment, is to actually develop your website. The code and files that you will need can be developed as though you are hosting the website locally. You'll treat everything in your code like it all lives in the same directory, (because eventually, it will, once Google has a hold of it.) This makes

development a little bit easier, although with flask and whatever other libraries you have this may still be a lot of work!

Once your website is developed, give Google access to your code by creating a repository in the “Development” menu on the left side menu on cloud.google.com. Make sure this repository is a mirror of your github repo with all your project code, and once it shows up under the “repositories” menu, you’re ready to proceed.

As a brief aside, the reason the previous step was necessary is that Google will only be able to clone into a VM repos it knows about, and this is how you inform Google that a repo exists.

The next thing we’ll need to talk about is app.yaml.

Inside your code repo, make sure you have an app.yaml. That file specifies the environment that you want your app to run in, (essentially setting parameters that the virtual machine running your website will need.) If you should have any dependencies or things you need installed on the box that will run your code, this is the place to specify them. Additionally, this is where you’ll specify the resource limits that you’ll require. So if, for some awful reason, your website uses 64gb of memory, you better:

a.) have your wallet ready, and

b.) specify that requirement in app.yaml.

There are also nifty scaling settings in here so that if you need a minimum number of VM’s running your code or whatever you can make that happen here. There’s also a file named requirements.txt, which you can employ to specify the python dependencies that your project has. It isn’t a requirement, (ironically,) but if you use the command “pip install -r requirements.txt -t lib/” you can install all of the requirements for your application inside the “lib” folder, where your app will eventually want them. We’re unsure whether or not you need to pre-install all your dependencies into the “lib” folder or if you’re able to have Google install them via app.yaml, but for this phase we chose to run the aforementioned command and specify our requirements in app.yaml. In future phases we’ll need to clear up what exactly the pipeline is for dependencies, in case we end up using any external libraries.

Deploying your app:

Deploying your app is shockingly easy, but you'll need to make a fairly large decision when you do so. Once all your code is ready to go, and debugged on your local machine, you open the GCloud console and clone the repo that you previously told Google about into a folder inside your GCloud console instance. Navigate inside that folder, to the directory containing your app.yaml, and give the command "gcloud app create." This instructs Google Cloud to create an app for your project, which it can then deploy to web users. When you run that command, it'll prompt you to choose a region, and you can type in the number of the region that's right for your website. (I assume, for this class, that you'll probably select US-Central.) **Note:** whatever region you pick is going to be the region this app lives in between now and the end of time — which I have been assured is going to be a while. Whatever region you pick, make sure it's the right one; your choice is irreversible.

Now your application is ready to be deployed. The command for that, as you might imagine, is "gcloud app deploy app.yaml --project projectid", where "projectid" is the actual ID of your project. (Look for that in the project selection menu, which is the three circles next to your project name next to the Google Cloud Platform logo.) Google App Engine will now host your website on the URL you have set up in the settings for your project. To access it, you can just type in the URL like any other website, and it will route you. By default, your project is hosted on "projectid.appspot.com" where "projectid" is your project ID as previously mentioned.

Routing to your custom domain:

It's nice of Google to host your project for you on appspot, but you probably don't want to make your business cards with an appspot domain printed on them. For the purposes of cs373 in fact, having a custom domain is a requirement! Luckily, routing from your Namecheap domain to your website is pretty easy.

Open Google Cloud Platform, and navigate to the app-engine page. Go to settings, and then to custom domain settings. Click "add a custom domain" and follow the steps as prompted. When it asks you to add a TXT record to your DNS settings, head over to your Namecheap administration page. Select the custom domain that you own, and then navigate to the "advanced DNS" tab. There, you can add the record Google wants for verification purposes. Once you've done that, sit back and wait while the DNS settings update on Namecheap's end. (For us, this took about 25-30 minutes.) Namecheap says in their documentation that these changes could take up to 30 minutes, so plan accordingly.

Once you've got your domain verified, Google Cloud will ask you to add a number of records to your advanced DNS settings to actually perform the routings that you want. They give you a list, and adding records is easy. The only thing you'll need to watch out for is that one of the records needs to be "aliased" as "www". What that means is that the "host" field for that record needs to be "www", rather than "@" for all the others. Once you've added all of Google's routing information, delete any other records that may be there for your github pages page, and within 30 minutes your Namecheap domain should route to your application.