5-2018

An Embarrassment of Riches: Data Integration in VR Pompeii

Adam Schoelz

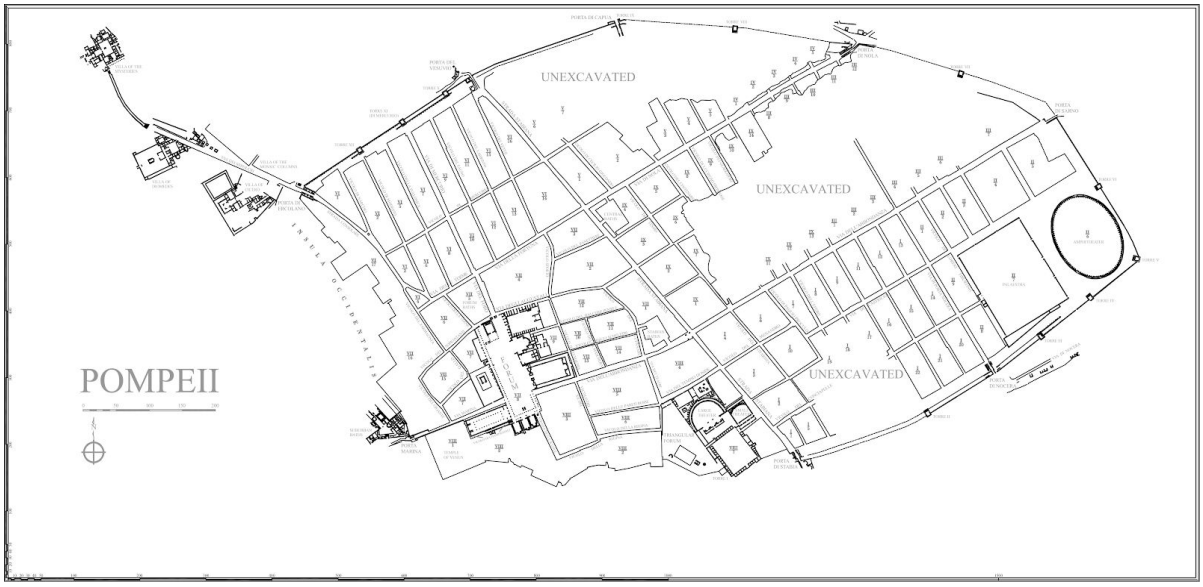
An Embarrassment of Riches: Data Integration in VR Pompeii

By Adam Schoelz

April 18, 2018

Abstract:

It is fair to say that Pompeii is the most studied archaeological site in the world. Beyond the extensive remains of the city itself, the timing of its rediscovery and excavation place it in a unique historiographical position. The city has been continuously studied since the 18th century, with historians and archaeologists constantly reevaluating older sources as our knowledge of the ancient world expands. While several studies have approached the city from a data driven perspective, no studies of the city have taken a quantitative holistic approach on the scale of the VR Pompeii project. Hyper-specificity has been the order of the day, leaving our knowledge of the city structure incomplete. The VR Pompeii project at the University of Arkansas aims to address this by performing, in concert, topographical network analysis of houses and neighborhoods, convolutional neural network identification and categorization of wall images, and analysis of space usage through subject tracking and electroencephalogram (EEG) data. Coordination of this data to maintain search-ability for non-technical scholars is a major challenge. To this end, the purpose of this research has been to design and then implement a database that allows for all of the VR Pompeii project data to be accessed together, with room for expansion, while maintaining a simple user interface to empower non technical users to ask questions that no researcher has been able to ask about the ancient city of Pompeii.



The ancient city of Pompeii

Introduction:

Project Summary

Classics is a field that lends itself to incredible specificity. The same body of text -- and even some archaeological sites -- has been studied for well over two thousand years, so novelty is a novelty. Like other academics, classical scholars are taught to have an incredibly high level of expertise in an incredibly specific part of the field. This has the advantage of generating new and interesting scholarship that reveals more and more to us about the nature of the ancient world. However, it does mean that sometimes, the forest is missed for the trees.

Consider the site of Pompeii, a Roman city in the bay of Naples that in 79 C.E. was destroyed by the eruption of nearby Mount Vesuvius. In a matter of hours the city was covered in burning ash, its inhabitants annihilated by pyroclastic flows. For over a thousand years the remains of the city lay undisturbed, until it was first seen again by the eyes of classical scholars in the 18th century. Since that time, the site has been continuously studied, and now is one of the

most popular archaeological sites in the world. It offers a tantalizing picture of everyday Roman life -- bread and chickpeas flash-fried but preserved, graffiti exhorting sexual conquests and insulting enemies, and fast food shops, still almost ready to serve a hot lunch. Most tantalizingly, there are many Roman houses preserved in Pompeii, allowing us a view of Roman living across the socioeconomic spectrum. These houses have been endlessly talked about since their unveiling, but in the tradition of Classics, holistic approaches have been dropped in favor of examining particular aspects of these houses.

Enter the VR Pompeii project. This project uses video game technology and design principles to let users explore Pompeian houses in real time 3D space, with the ultimate goal of exploring these houses in VR. By combining spatial and proxemic analysis with tracking of players in the houses, EEG data of said players as they move through the space, and neural-net based analysis of Pompeian wall paintings, this project will be the first of its kind -- a holistic and quantitative analysis of the houses of the city of Pompeii, allowing researchers to ask questions that have never before been possible.

A project of this scale has a significant amount of data associated with it, from a variety of sources. In addition the data's variation, it must be stored in an easily accessible way. Often when discussing databases with non-technical personnel, it is found that they work around the database, as opposed to working with it. Unpleasant UI experiences and difficulty of creating encompassing searches across multiple data sets means that non technical researchers often try to extract data from the database and search through it on their own. This is less than ideal. For a project of this type to succeed, data needs to be easily searchable across lines of collection -- the entire point of the project is comparison across these lines. The ability to search what is

necessary without necessarily exposing that complexity to the user is highly important for the success of this project.

In addition, the database must support platform-independent insertion and updating. Data will be streaming in from many different sources and languages, and needs to be transformed into a unified structure. Data from all applications is treated equally, and therefore data requirements must rigid and prescriptive.

Approach:

Wrapping a database with a **RE**presentational **S**tate **T**ransfer API, or REST API, was the best way to serve all these needs. A stateless, independent station for all applications to interact with data helps secure and unify data and can enable a sophisticated search that is simple to use. However, several issues arise when composing such an application that must be considered. These include the style and schema of the internal database, as well as the structured nature of querying and creating data in that database.

Pompeian house, photo from 2017

Issue: relational versus non-relational database

Relational databases are by far the most common databases used in computer applications. Data is stored in tables, which have rows and columns. Columns define the parameters of data, and rows define the data itself. Each row has a primary key, which acts as an unique identifier within that table, and can have a foreign key, which is a reference to a row in another table. Tables relate to one another through these foreign keys, thus the name -- relational database.

The emergence of NoSQL databases -- that is to say, databases that are not relational in the way they store data -- in the last five years as 'the database for websites' has muddied the waters of choosing a database. Where relational databases were once a given, database designers must now weigh the advantages of a NoSQL approach, which often are friendlier to web frameworks. NoSQL databases, also called schemaless databases, eschew traditional relational models in favor of a document based approach. The content of a document is not fixed, and so elements within a NoSQL database do not necessarily resemble each other. Documents are also stored in a JSON-like key-value pair fashion, making interoperability with web languages very simple.

MongoDB, the most prominent NoSQL document-based database, was strongly considered for this project, and it appeared at first to be a highly viable option. In particular, the ease of integrating it with the Express framework on which the REST API was being constructed was highly appealing, as the plugin for Mongo is more fully featured then its MySQL equivalent. In addition, the lack of rigidity to data structure in Mongo might have been useful for one

particular -- and unusual -- aspect of VR Pompeii project data: wall art elements. These elements are highly variable in terms of content and location, and seem perfect for the flexibility of a NoSQL database.

However, after some research into the realities of its usage, the shortcomings of Mongo became readily apparent. The main value of the VR Pompeii data is not in the data itself, but in the ability to relate the data to itself and reveal non obvious, quantitative relationships between different elements found in Roman domestic environments. Relational databases make this conceptually simple -- they're based on relations, after all. NoSQL databases, despite the ease of setting them up, quickly turn into a denormalized nightmare when trying to extract related data. Extracting data from one document in MongoDB is quite fast. However, storing VR Pompeii data in a single document would require massive reduplication of data, and storing in multiple documents would require implementing a relational-style model which reduces the NoSQL advantage significantly.

In addition, rigidity of form has its advantages. Most of VR Pompeii's data is well defined in structure. An SQL database is fail-fast and will indicate if inputted data is improperly formatted, which is an asset to the project. I also have greater familiarity with SQL, so issues that arise while developing in that ecosystem are more likely to be solvable than issues with MongoDB. In the end, using MySQL came out as by far the best choice.

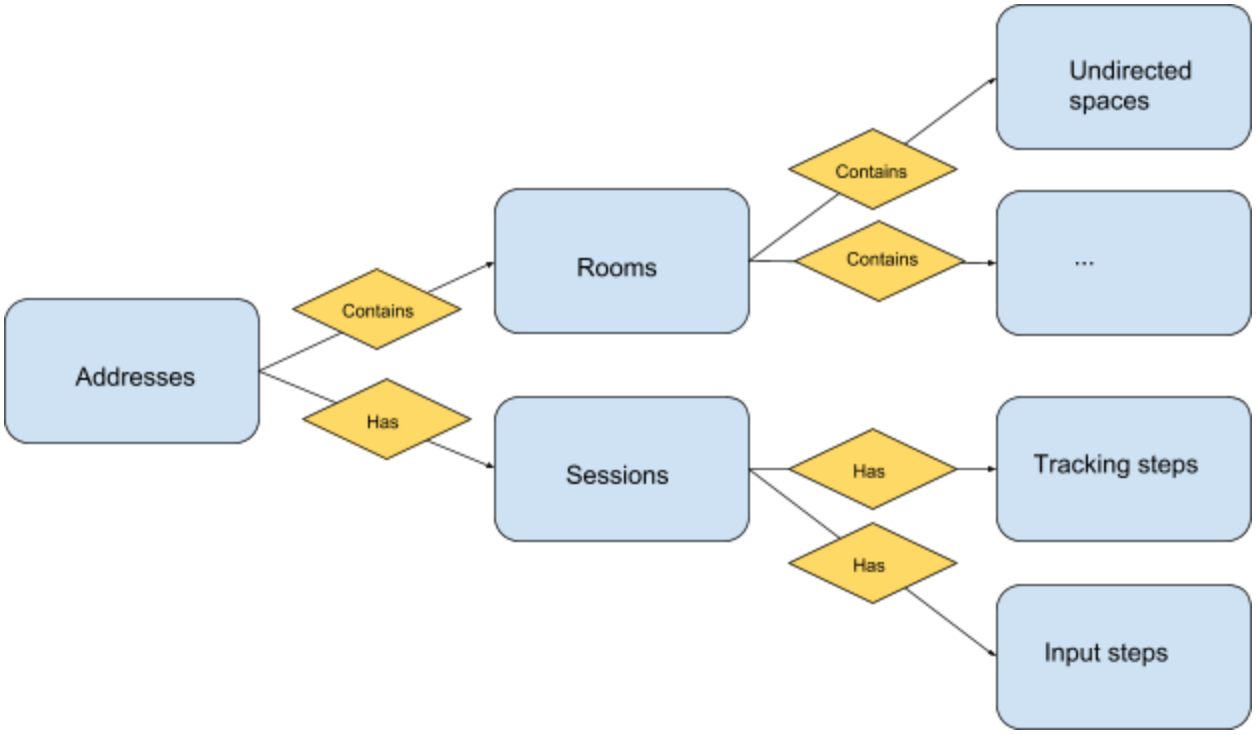
Issue: schema

As stated above, the value of VR Pompeii's data lies in its relationships between data objects. These relationships must be carefully mapped in a schema to achieve two objectives:

ease of data retrieval, and ease of adding new data that relates to previous data. An important aspect of any new data generated is that it will be more specific than old data: having house-level data engenders room-level data creation, and room-level engenders space- and wall- level creation, and so on.

With this in mind, in the VR Pompeii database, tables can be more or less granular. These relationships are not absolute, but are defined by a table's relationship to either super-tables or sub-tables. A less granular table is a sub-table that a child references, and a more granular is a super-table that is referenced by other tables. In general, these relationships map to the scale of the data they handle: the least granular table in the database is the addresses table, which defines entire neighborhoods, and the most granular table currently is the undirected spaces table, which defines a portion of a single room within a house. Information to be added to the database will almost certainly be equally or more granular, like wall or wall area data.

The essential structure is as follows: more granular tables have a **single reference to an owner** in an immediately less granular table. In this way, **wall areas** are owned by **walls**, **walls** by **rooms**, **rooms** by **houses**. A owner-owned relationship is multidimensionally one to many: just as there are many **rooms** in one **house**, **rooms** and **sessions** can both link back to a single **house**.



This design allows us a great deal of flexibility in the more granular direction -- that is, the direction of more specific information. This will require rigorous initial planning at the highest level. It also allows us parallel granularity, where, if we wanted to track information about doorways, we could add a Doorway table in which every entry has a reference to an owning Room, without updating the Room, Wall, or more granular Wall Area table. This allows us the advantage of easily and dynamically updating the database even years down the line as we generate larger bodies of relevant data.

An important aspect to VR Pompeii data is that it is fairly static. Deletion and updating are fairly rare. Once an analysis of a space has been completed to the details of the negotiated data contract, that data will probably never change. With this in mind, when a record is removed, that deletion should cascade to all children of that object.

Summary of Tables

Currently in the database, there are six defined tables: addresses, input_steps, rooms, sessions, tracking_steps, and undirected_spaces. The purpose and model of these databases is outlined below. It is important to note that several tables will be added to the database as the relevant data is generated. These tables include wall, wall area and directed spaces. However, the exact nature of the data in these tables has not yet been defined.

Addresses

Purpose: The address table is the top level table in the database. It contains house-level location data that allows different systems to attach themselves to the proper location. Address data is not geospatial, however, but follows the standard Pompeii addressing system used by archaeologists. Using established standards for addressing increases the ability of nontechnical users to use the database.

Fields:

ID	A not-null incrementing integer internal to the database for quick comparisons. Primary key.
Regio	A regio is a Roman equivalent of a neighborhood. In Pompeii, regiones are numerically numbered. Stored as an integer.
Insula	An insula is roughly equivalent to a modern city block. Like regiones, these are numbered, which makes storing them an an integer simple.

Doorways	The final piece of the established convention for Pompeian addresses, Doorways describes the range of (numbered) doorways in an insula a given house covers. Due to discrepancies in formatting, this is best stored as a string.
Name	Pompeian houses are practically all named. This is not quite as useful as formal identification feature due to the brittleness and time requirements of string comparison, but can be useful as a search tool. Stored as a lengthy string.

Purpose: A normalization of a possible field in the address table, Room contains the room code from the standard addressing system and a links back to the overarching house. Room is useful as a concept separate from house for several reasons. There is of course a one-to-many relationship between rooms and houses. In addition, several sub-tables contain linked room-specific data. Spatial analysis on a room as designated by the standard system is not necessarily useful -- it is more productive to break up a room into multiple spaces, each with its own proxemic values. However, this makes linking wall analysis with proxemic analysis difficult. Hence, it is better to link both into the superstructure of Room, where there is no analysis performed, but both Walls and Spaces can easily be traced. Using the standard addressing system means that archaeologists with no knowledge of our database will be able to navigate this structure as well.

Fields:

ID	A not-null incrementing integer internal to the database for quick comparisons. Primary key.
Code	Pompeian rooms have been assigned a code by excavators. The assignment itself does not follow any convention -- they are not numbered clockwise or from the top down or according to any real method. However, they are essential as a standard of identification in searches.
Address	Foreign key reference to an address ID.

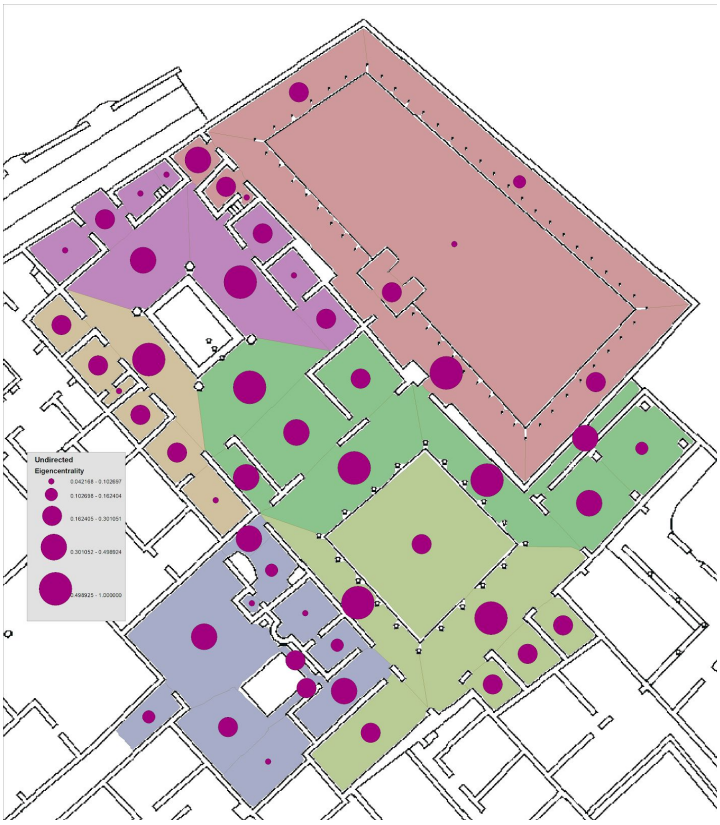
Undirected-Spaces

Purpose: The spaces table contains network topographical analysis of Pompeian houses. Houses are split into spaces using a modified watershed analysis in ARCgis, and then a number of social network analysis algorithms are employed to determine the properties of that space. Rooms and Spaces have a one-to-many relationship. While it might seem intuitive to have a one-to-one relationship between Rooms and Spaces, this does not reflect the realities of the data, or of traversing through the space in reality. While the atrium -- a large room in the front of the house with a water feature in the center and a skylight above -- might be considered one room on maps, the central water feature ends up splitting the space significantly, making network topology of the entire room less than useful. Analysis is better performed on multiple subspaces -- thus the splitting of Spaces from Rooms in the database.

Fields:

ID	A not-null incrementing integer internal to the database for quick comparisons. Primary key.
X	The GIS longitudinal location of the room. Stored as a double for maximum accuracy.
Y	The GIS latitudinal location of the room. Stored as a double for maximum accuracy.
Area	Value indicating the area of the room in square meters.
Squareness	Value indicating how close the room is to a square. Low values indicate long, narrow rooms.
Eccentricity	Value indicating lack of connection to other spaces in network. Stored as a double.
Closeness Centrality	A measure of closeness to other spaces in the network. Stored as a double.
Harmonic Closeness Centrality	Another measure of closeness in the network. Stored as a double.
Betweenness Centrality	A measure of betweenness in the network based on shortest paths. Stored as a double.
Eigen Centrality	A measure of the influence of a node in a network. Stored as a double.

Modularity	A measure of the strength of division of a network into modules. Stored as a double.
Room	Foreign key reference to room that contains this space. Stored as integer.



Pompeian house broken up into Spaces, with Eigen centrality overlaid

Sessions

The type of environment that players navigate during VR Pompeii testing

Purpose: The sessions table is storage and container for demographic data relating to user testing. In VR Pompeii tests, users walk through a house either without a goal or in search of an object. A demographic survey is performed beforehand, and then the player's input and location are tracked as they move through the building. This information is stored in two sub-tables that link back to an owning Session. It relates the larger bodies of tracking data, and enables analysis to occur on whole sessions. While it is possible to grab a subset of tracking data from one of the sub tables, the design intention is to consider sessions holistically and retrieve an entire session and its sub-data at once.

Fields:

ID	A not-null incrementing integer internal to the database for quick comparisons. Primary key.
Resolution	The number of times the user's position and rotation is polled per second in this session.
Age	The age of the test subject.
Race	The race of the test subject.
Gender	The gender of the test subject.
Game Experience	The amount of experience the test subject has navigating 3D space in games and interactive visualizations, previous to VR Pompeii testing.
Address	Foreign key reference to the house the test subject is exploring.

Tracking_steps

Purpose: The tracking_steps table is designed to store individual tracking markers of a player's testing session. These markers are taken at a tick rate of the session's resolution, and represent a player's movement through the space. This is a powerful analysis tool that allows researchers to see the interaction between calculated values of spaces and real world usage. For example, by aggregating player positions, a heatmap can be generated of a house showing frequency of space occupation. This heatmap can be compared with network analysis to determine how parts of the house are processed by visitors, and how that changes as familiarity with the space grows.

Fields:

Session	Foreign key reference to a session. Part of composite primary key with Stamp.
Pos_x	The x position of the test subject in the house in Unity units. Unity units are analogous to meters for the purpose of scaling houses and player movement virtually. Unity units can be transformed to GIS coordinates (and vice versa) based on a known constant. Stored as a float.
Pos_y	The y position of the test subject in the house in Unity units. Stored as a float.

Pos_z	The z position of the test subject in the house in Unity units. Stored as a float.
Head_rot_x	The x rotation of the test subject's head, in Euler degrees. Stored as a float.
Head_rot_y	The y rotation of the test subject's head, in Euler degrees. Stored as a float.
Head_rot_z	The z rotation of the test subject's head, in Euler degrees. Stored as a float.
Body_rot_x	The x rotation of the test subject's body, in Euler degrees. Stored as a float.
Body_rot_y	The y rotation of the test subject's body, in Euler degrees. Stored as a float.
Body_rot_z	The z rotation of the test subject's body, in Euler degrees. Stored as a float.
Stamp	Timestamp representing the time the player's location and rotation were polled, starting from the beginning of the session. Stored in the DateTime format. Part of composite primary key along with Session.

Input_steps

Purpose: The input_steps table tracks all input from a player during a play session with a timestamp. This information can be used to recreate the actions a user takes while navigating VR Pompeii. It can also be used to simulate player movement through domestic spaces. This is stored separately from the tracking_steps info because the two are not necessarily aligned. While tracking_steps recur on a fixed time scale, input is user generated. Input could be aligned with that fixed timescale, but that would reduce the accuracy of data especially of the resolution of tracking_steps is low. Creating a new table to preserve input data at an exact resolution is a pretty trivial way to ensure that agents are able to most accurately learn from test data.

Fields:

Session	Foreign key reference to the session that owns this input data. Part of composite primary key along with Stamp.
Input_code	String representing the input key pressed.
Stamp	Timestamp representing the time at which the input key was pressed, starting from the beginning of the session. Part of composite primary key along with session.

Issue: REST API framework

A REST API, standing for REpresentational State Transfer, is the most common architecture for web APIs on the internet today. The architecture guarantees a number of conditions that make using a RESTful web service simple for client side developers. Requests

are stateless; they contain all the information they need to be executed. In addition, RESTful services present a uniform interface to developers, which makes them easy for any application to use. REST APIs use standard HTTP operations -- GET, POST, UPDATE and DELETE -- hence there is no additional overhead in making a request to a RESTful web service.

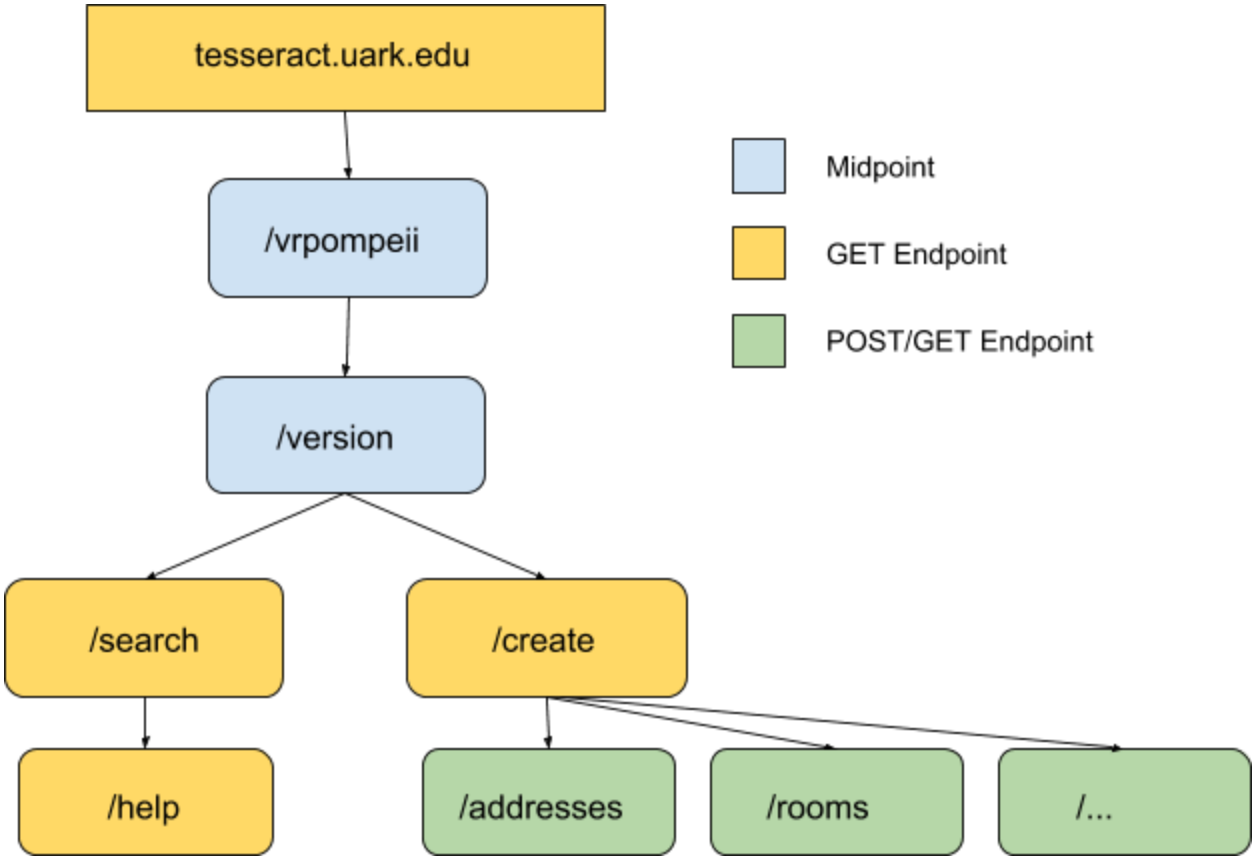
In practice, a REST API consists of a cluster of endpoints on a website, usually of the form www.example.com/api/create/something. Formally, an endpoint is a full url that, when hit, will do an operation. In this example, an application would POST a data object to that link, and then, given properly formatted data, the server would create new rows in the relevant table in the database with that data. Beyond the general architectural parameters outlined above, however, the endpoints of a REST API are not fixed.

While REST architectural principles are fixed, web development in general is a land in constant flux. Languages and frameworks are fractured and new technologies crop up daily. Selecting even a language can be difficult, let alone a framework. For this project, we considered PHP, Javascript and Rust for the backend language, and Sinatra, Rails, Express, Restify, Lumen and others for the backend framework. With this in mind, I went about selecting a language by assessing both the simplicity of learning and the longevity of a given server-side web language.

PHP was a prominent choice as it is still stubbornly the weapon of choice for many web developers, but in the past I have found it an onerous language for development. Javascript, while contentious in some circles, has proven as long lived as PHP, without many of the development complexities that make PHP development less than desirable. Javascript is also a proven web language, but without much of the baggage that makes PHP difficult to develop. While Javascript is a relative newcomer to backend scripting in the form of NodeJS, it has

quickly matured into one of the most popular languages for backend development. Rust was also considered, but I found multiple recommendations against using it for a rookie backend developer.

I chose Express for my framework because it also straddles the line between power and simplicity, and it exists within a mature ecosystem of tools. Express is oriented around URL endpoints in the same way REST APIs are. URLs are passed between handler functions called 'middleware,' which are called based on the URL path. What makes Express exceptionally powerful is that it uses regular expressions to match URLs, so any given URL can be parsed through multiple handlers until it reaches an endpoint, where a response is sent to the client. For a REST API, this is very powerful, as it allows creation of DRY -- **D**on't **R**epeat **Y**ourself -- code that is not possible using a different framework. Using Express, each 'branch' of an API can have common processing occur first and then that data can be passed to the next middleware function. These middleware chains allow significant data processing to occur ahead of the actual endpoint -- for instance, authentication can be handled at a shared 'root,' and then the HTTP request can be passed on to the next middleware for processing.



Map of the current VR Pompeii REST API

Issue: REST API data creation

Creation endpoints for the VR Pompeii database were the top priority for initial development, so that testing data could be stored even before it could be easily queried.

In the VR Pompeii project, applications POST a JSON-formatted data object to a 'create/objects' endpoint. In general, this endpoint parses the object and inserts the data contained therein into the appropriate table. If the data is inserted successfully, then a 200 all-clear is sent to the client. If the inserts fail, a 500 internal server error is sent. Inserts will often fail because of improperly formatted data, a GET to the same endpoint will send an example data object as a response.

A particular challenge when designing creation endpoints for VR Pompeii was the nested nature of the data. While each table has an ID field, exposing this so that sub-table inserts can access it is not an ideal solution. ID fields are non intuitive and applications can hardly be expected to guess what the ID of a given parent object is. While insertion could require a search request to get the ID for the relevant object out of the table, this again is non intuitive, and more importantly breaks the atomic nature of an insert operation. Developers should not have to guess that they need to perform a separate search hit before POSTing a creation object.

What can be expected of application developers is having the plain-language identifier for a data object, like the address of a house or room. This means that the search can be folded into the POST request itself, taking the burden of searching off of the developer. This introduces two complexities: handling the asynchronous search and communicating with application developers the requirements for creating objects.

The latter is simple, the former more complex. Developers can GET the same endpoints they POST data objects to in order to receive a generic sample object that they can use to format their data objects. It isn't the most dynamic solution, but it is an easy and intuitive way to communicate data requirements.

The asynchronous search prevents one major speed optimization: insert statements cannot be grouped into a single statement. Large inserts do not necessarily have one parent object. These objects must be queried before the insert statements are assembled. That query is asynchronous, so insert queries are assembled in the callback and cannot be grouped into a single statement. Instead, insert statements must be assembled and run one-by-one in the callback. This

has a speed drawback, but a security advantage, as multiple statements are a common vector for an SQL injection attack.

Issue: REST API generic search

A flexible and simple search was one of the cornerstones of VR Pompeii's REST API design. Giving applications the ability to perform dynamic and complex queries has proven to be a daunting task, but one rendered wholly necessary by the goals of the project. Making this search RESTful is also important, but imposes additional design challenges.

There are four options when designing a RESTful search. The first is a noninteractive search, which is the most secure but the least flexible. In this model, applications GET an endpoint that represents a specific SQL query which is not exposed to the user. This has the advantage and disadvantage of complete control over what a user can access from the database. In particular for this project, anticipating user requirements of the database is difficult, as the entire point is to render visible nonobvious data relationships. A hundred pre-generated queries could completely miss what users actually need. For this reason, this approach was considered too brittle.

The second approach is the exact opposite -- generating SQL queries client-side, which are then parsed and run by the server. This has the advantage of allowing users complete control over search, and makes handling search requests a breeze. It has the major, major disadvantage of complete insecurity. Especially in a public-facing database, trusting users is a foolish choice. However, running client-generated SQL queries remains a shockingly common practice in web development, probably because of ease of implementation. For the VR Pompeii application, this approach was considered far too insecure.

The third approach is POSTing a JSON search object to the server. This allows for cleanly formatted search data for easy parsing by the server. In addition, a complex search object could make processing server-side simpler, as it requires client preprocessing.

POSTing a search object in JSON has two disadvantages: it is not RESTful, and it is not a black box. Search is fundamentally not a POST operation, as there should be no change to the data set -- it is merely being accessed. POSTing a data object, while powerful, is non intuitive because of that fundamental disconnect.

In addition, the format of the data object necessarily reveals the nature of the generated search query. Given a sufficiently powerful search tool, the data object could easily reveal not just *what* is being searched, but *how*. This is ultimately an insecure approach, but it also makes the tool on the whole harder to use, as clients must understand more than the 'what' because of formatting issues. This approach was not selected because it is not RESTful and it requires too much of the client.

The fourth approach is embedding search parameters in the URL and generating a parameterized query based on those user-provided values. This is a very loosely formatted case -- the client only provides what it is looking for, and the parameters in which it wants to look.

A major advantage of this approach is that the search remains a black box. All the client has to do is provide a set of values, and it gets back a JSON data object. There is no knowledge required of how the search is structured beyond the most basic level. It is also intuitive. Once the client knows what it needs, there is no complex formatting to be done client-side. It is also RESTful, as search remains a GET operation.

Implementing this option is complex because information must be combined from multiple tables with database join operations. Luckily, the necessary joins can be isolated from the rest of the query and generated using a dependency-based table selection algorithm.

First, the required tables must be determined from the input parameters. This can be done by comparing input parameters to a given configuration object that knows what parameters belong to which table. Once the necessary tables have been determined, the joins between the tables must be generated. Because of the structure of the data, many tables cannot be directly joined -- if a user wants to search for the eigencentrality of a space as well as any sessions that cover that space, then the Addresses and Rooms table must also be joined into the search to create common ground between those two tables.

While attempting to do this through generating explicit joins has proven very complex, implicit joins make this far simpler. Generating explicit join statements requires walking the dependency tree and generating context-sensitive join statements that are modified depending on whether they are being joined up the tree or down the tree. By contrast, one the intermediate dependencies have been found, implicit joins -- where the SQL query handler orders the joins -- can be easily assembled and run.

Issue: REST API security

As with any public-facing database, security is an important and complex consideration. There are multiple attack vectors on any REST API facing a database, but by far the most common is an SQL injection. Given the ability to query a database in an abstract way, a malicious user could ‘inject’ an SQL statement into another SQL statement, creating unintended and negative functionality.

There are clear ways to defend against SQL injection attacks, all based around limiting direct user input into the database. One could simply only use pre-generated queries, but this is difficult, as anticipating user needs is nigh-impossible. Pre generated queries that are too broad also run the risk of serving up a ton of data that the user does not need and generate weight on the server.

A more realistic approach is through the use of parameterized queries. These queries are mostly static, with room for user input to be inserted at runtime. Obviously this creates room for an injection attack, but if these user-generated parameters are sanitized -- that is, neutralized of malicious ability -- then they are safe to use in the database. The NodeJS MySQL module helpfully includes the ability to sanitize user input, which makes implementing parameterized queries simple. The main requirement on a developer when using parameterized queries is discipline; escaping every user-input value is necessary. There is no room to leave a hole for unsanitized user input.

Conclusion:

The goal of this project was to provide the data backend for the VR Pompeii project in the form of a expandable database schema and a flexible REST API for accessing that data in a uniform and robust way. The main work of both of these tasks has been accomplished. The VR Pompeii schema is reactive enough to handle new tables while enforcing data requirements on tables that already exists, and the VR Pompeii REST API has its essential functions -- creation and search -- completed.

There is still a non-trivial amount of work to be done on this project during the summer. As data needs expand, new tables will need to be added to the database and functionality for

accessing them must be added to the REST API. While the workflow for doing this is well established, it will still require some design work. The large part of the labor for adding new tables will be the negotiation of data contracts, especially as more granular data pulls from multiple tables to generate new information.

In addition, work will continue on robustifying the search function of the REST API. A better authentication system will be a good first step. In addition, a user friendly website for querying the database should be constructed and made publicly visible. Finally, more complex operators will be added to search parameters, allowing for range searches.

References:

arcgis.com. [Online]. Available: https://www.arcgis.com/features/index.html. [Accessed: 18-Apr-2018].

"CHAPTER 5," *Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)*. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Accessed: 18-Apr-2018].

J. J. Dobbins, *The world of Pompeii*. London: Routledge, 2010.

N. Foundation, *Node.js*. [Online]. Available: https://nodejs.org/en/. [Accessed: 18-Apr-2018].

"MySQL Documentation," *MySQL*. [Online]. Available: https://dev.mysql.com/doc/. [Accessed: 18-Apr-2018].

"Using middleware," *Using Express middleware*. [Online]. Available: https://expressjs.com/en/guide/using-middleware.html. [Accessed: 18-Apr-2018].

"What Is MongoDB?," *MongoDB*. [Online]. Available: https://www.mongodb.com/what-is-mongodb. [Accessed: 18-Apr-2018].

"Why You Should Never Use MongoDB," *Sarah Mei*, 12-Nov-2013. [Online]. Available: http://www.sarahmei.com/blog/2013/11/11/why-you-should-never-use-mongodb/. [Accessed: 18-Apr-2018].

J. J. Dobbins, *The world of Pompeii*. London: Routledge, 2010.

G. P. Carratelli and I. Baldassarre, *Pompei: pitture e mosaici*. Roma: Istituto della enciclopedia italiana, 1990.