1 Introduction

Babylonian Seal Collection is a digital library for recording and displaying seals from the Yale Babylonian Collection. It is built on Django REST Framework (backend) and React (frontend). The Yale Babylonian Collection is one of the leading collections of Babylonian artifacts in the world, comprising over 45,000 items, including cuneiform tablets, cylinder seals, and other artifacts. However, the size and heterogeneity of the collection makes organizing it difficult. Efforts to digitize such collections have be stymied by disagreements over spelling, challenges in scanning artifacts, and more.

This project aims to design a database schema that enables more complex representations of the collection while improving usability and organization. Furthermore, it aims to build a robust API backend based on the latest web development standards and patterns. Finally, the client-side application utilizes graphical user interface best practices and user-oriented design considerations. I collaborated with the Yale Babylonian Collection researchers Agnete Lassen and Klaus Wagensoner.

The project is built with reusability and portability in mind, with proper documentation, commenting, and a logical project structure. This allows for future developers to easily add and modify features. Finally, the project is integrated with Heroku so that deploying to the web is as easy as possible. In sum, Babylonian Seal Collection is a full-stack web solution for the Babylonian Seal Collection.

While designing the application, I had to consider multiple issues, including the structure of existing databases, ease-of-use, and the needs of the Babylonian Collection researchers. The open-endedness of many fields meant that the final schema included several deep, nested relationships, creating challenges on the frontend. The lessons gained will be useful for similar ongoing projects.
2 Designing the Database

2.1 Tools

The first decision to make with regards to the database was whether to select a “NoSQL” database like MongoDB or Couchbase or a standard SQL database like MySQL or PostgreSQL. Secondly, I had to decide on which database to use. Finally, I had to decide whether to use an object-relational mapping layer or direct SQL queries.

After consultation with the Babylonian Collection researchers, I realized that given the large number of relationships among tables in the database, a standard SQL database was the most suitable. Next, database selection was largely a matter of preference, given the standard field types. However, the deciding factor was PostgreSQL’s free integration with Heroku. For local development, SQLite was already integrated with Django. Using Django and Django REST Framework on the backend (discussed later) also meant that I had to use Django’s object-relational mapping layer, which had the additional advantage of integrating models with the API serializers.

2.2 Process

Designing the database require careful communication with the Babylonian Collection researchers because while they had already been using a Filemaker database, certain concepts like table relations, enums, and so on had to be clarified. Other than consulting with the researchers, I contacted researchers working on the Cuneiform Digital Library Initiative at UCLA. I found out that they were also working on a database redesign and was given access to the existing schema for reference.

As discussed in my project proposal, I was told by Dr. Émilie Pagé-Perron, a co-Principal Investigator of the CDLI, that

The current CDLI schema is basically composed of three flat tables linked using the artifact id (no foreign key) and a couple of other tables containing textual analysis results (eg lists of words and the like). This simple design was put in place in order to provide a SQL copy of the FileMaker Database which is at the heart of
the catalogue data, so it could be accessed through a web interface. We are
currently in the midst of the full redesign of the database.1

Flat tables with no foreign keys makes searching and organizing the data especially
difficult. Search on the CDLI website usually takes more than 10 seconds. Additionally,
there is no straightforward way to discover relationships between objects on the frontend.
For example, the “Period” attribute does not link to a page with other artifacts that
originate from the same period. The database also suffered from sparseness. Some
collections only included data for specific fields. The data also needed to be cleaned. Some
columns were ambiguous, like “subgenre_remarks”. The “subgenre” column included rows
like “witness Archaic Vocabulary” and “Archaic Lu2 A (witness)”.

As such, I determined that as far as possible, the table columns should be kept to fixed
options or relationships to preserve data integrity over time. At the same time, it was
difficult to settle on fixed options for certain fields like “languages” or “periods” with the
researchers. The evolving and heterogenous nature of the field meant that researchers
preferred to keep these options open-ended and customizable. As a result, I had to create
additional tables to allow researchers to create new options.

The resulting schema (see Figure 1) included many tables to accommodate for this open-
endedness. In all, there were three types of tables:

1. “Core” tables: The Seal and Impression table formed the core of the schema. They
   were related to each other via the seal-impression relationship. Additionally, the
   Seal table had a many-to-many relationship with itself via the related_seals field.
   The rest of the tables were related to the core tables in some way, usually via a
   many-to-many relationship.

2. “Tag” tables: These were the tables that covered different field options for seals
   and impressions, such as languages, periods, and styles. They were characterized by
   a very simple structure – just an ID and a “name” or “title” CharField. This
   allowed for simpler serialization and visual representation, as will be discussed
   later.

---

1 Email correspondence.
3. “Nested” tables: These also contained different field options, except that instead of a single CharField, they included several fields. The most complex table of this type was the Text table, which included a many-to-many relationship field to a “Tag” table.

2.3 Challenges
While the Django ORM simplified the database setup, the lack of fine-grained control over the database meant that I had to deal with multiple migration files and keep track of the migration status. This required greater discipline over model changes. It was especially difficult dealing with nested-type tables as changing the type of relationship would cause cascading changes in the database.

Furthermore, the open-endedness of the actual collection often forced me to fall back on the many-to-many relationship, even when it might not initially seem like the most obvious choice. For example, the seal-impression relationship intuitively appeared to match a one-to-many relationship, in that one seal made multiple impressions, while every impression could only come from one seal. However, I also realized that it was possible for certain artefacts to feature impressions from several seals. Additionally, it might also be unclear which iteration of a seal was used to create an impression. As such, I had to change the relationship to a many-to-many relationship.

Another characteristic of the collection I had to deal with was sparseness. Other than the name, all other fields in the Seal model had to be nullable or able to be set to blank. For example, for the is_recarved field, rather than simply using a BooleanField, I had to use a NullBoolean field as researchers could simply not know for certain whether it was recurved or not. This had important implications for the frontend design and display. It also meant that I had to ensure that I picked nullable field types.
Figure 1. Visualization of database schema grouped into Django applications
3 Building the Backend

3.1 Tools

I first decided to separate the backend and frontend with a REST interface in between so that the API would be portable and flexible enough to use for data visualizations or alternative research. I also wanted to use Python on the backend because of my familiarity with it and performance concerns (a Node/JavaScript server could only be single-threaded and had concurrency issues to deal with). As such, the decision went down to either the Flask or Django web frameworks. Ultimately, Django won out because it had more “batteries included,” including the built-in ORM. Moreover, a popular add-on, Django REST Framework, allowed me to build an API with serializers and model integration relatively quickly. Features or functions that would have taken many lines of code to build manually in Flask were encapsulated in default classes in Django REST Framework.

3.2 Process

Implementing the models was a relatively simple process of translating the database schema into Django’s ORM specifications. However, given the difficulty in serializing complex nested tables, I took an incremental approach, beginning with the non-nested fields of the Seal table, then adding the “Tag” tables, followed by the “Nested” tables, and finally the Impression table. Along the way, I added user authentication via JSON Web Tokens, which would allow for greater backend-frontend separation.

Taking into consideration the frontend aspects of the application, I decided that only the seal and impression API endpoints would allow for create, update, and delete operations. This would allow one or two forms on the frontend to update the database, and keep the API focused on the “Core” tables. However, this also meant that the serializers for seals and impressions would have to be written manually to handle nested creation and updates of other tables. I had to override the default create and update functions in the serializers to handle this.

The incremental approach was especially useful because once I had figured out how to implement the nested serialization of a “Tag” table, for example, it was a simple process of replicating it for the rest of the “Tag” tables. Nevertheless, there were some differences
among the “Nested” tables. For example, while the Text table included a ManyToMany field, the Publication table only had CharFields and one ISBNField (which was a variation of a CharField). As such, I had to customize the serializers used for “Nested” tables and could not apply the exact same functions to them.

### 3.3 Challenges

While the “batteries included” nature of Django and Django REST Framework sped up my development considerably, it also caused problems when dealing with non-standard or heterogenous features like nested serializers. The default serializers did not support nested serialization, so I had to manually override the create and update functions to do so. This had implications further down the line as it affected the way Django REST Framework handled data validation. I had to approach the serialization code very carefully to ensure the flow of data was correct.

Another challenge was dealing with special fields like ISBNs or measurements. In some cases, I was able to include special libraries to add specialized validators and handling of data, such as the ISBNField. In others, like measurements, the libraries only created more issues due to incompatibility between the backend and frontend’s representation of the data. I had to examine the source code of these libraries to ensure that they would do what I expected.

Finally, the “many applications, one model” approach I took also created complications when trying to establish relationships between tables. The two “core” Seal and Impression tables, for example, had relationships to each other but I could not import their serializers at the same time as that would create a circular import. To resolve this, I wrote additional minimal serializers that would serialize only the data needed to create these relationships.
4 Creating the Client

4.1 Tools

For the frontend, I chose React as it is currently one of the biggest user interface libraries and has a large network of integrated tools and components. This was especially crucial as I needed to build a form that included nested fields, complex field types, and validation. Finally, React had the advantage of quick setup with the create-react-app package that automated the process.

On top of React, I had to choose a frontend component library. While Bootstrap, Semantic UI, and Foundation all came to mind, I chose Ant Design, a library originating from Alibaba. It met several key requirements, including natively integrated React components and data display components. More crucial, however, was its powerful form wrapper component that was able to automatically validate and capture form field data. While other libraries all had some kind of form and form field implementation, it was not as integrated as Ant Design. Given the importance of the form in the application, I chose Ant Design.

4.2 Process

The frontend development progressed in tandem with the backend development, meaning it also followed the incremental approach. At the same time, it broke down the CRUD (create, read, update and delete) operations from the backend into corresponding views on the frontend, such as the list view that utilized a table and the create view that utilized the form.

A large part of the development went towards experimenting with and deciding on which form component matched the model field. For instance, while a switch component (Figure 2) might appear to be the best fit for a Boolean field like is_recarved, I realized that the switch component only had two states: true or false. In contrast, the actual representation of is_recarved on the backend model was a NullBoolean field, which meant that the frontend representation had to allow for three states: true, false, and null. This was not possible with the switch component. As a result, I had to go with the less flashy but more accurate radio component (Figure 3), which allowed for no option to be selected and hence return a null value.
Another important part of the process was design thinking from the user’s perspective. For example, due to the frontend-backend separation, the edit form might not load all the resource’s data immediately. However, I had to prevent the user from clicking “submit” before the data had loaded, as this could wipe the unloaded fields. As such, I had to add loading indicators and loading state to indicate to the user that data was still being loading. Additionally, I added breadcrumbs to help the user navigate the site better.

4.3 Challenges

While the integrated components provided by Ant Design sped up my work, just like on the backend, it caused problems when dealing with non-standard cases. For example, Ant Design’s form component did not have full support for nested fields and arrays, so I had to build my own workarounds. Additionally, special fields like ISBN required me to find and add my own frontend validators which did not exactly match up to the backend validators. I had to make adjustments and compromises on both the frontend and backend in order to get them to align better.

5 Conclusion

The final step, deploying to Heroku, was just as complex, requiring me to deal with static file generation, Cross-Origin Resource Sharing, and buildpack ordering. Nevertheless, with proper commenting and documentation, the final product was a full-stack solution that could be easily deployed and modified by future developers.

Working on the project revealed many important considerations, such as how to represent real-world data in an actual database schema and how to display that data.
Communication was central to my work as I had to explain different web development concepts to non-technical users and align their vision to the realities of web development. I found that while web development frameworks and tools tended to favor fixed, clearly-defined representations, reality was often a lot messier for researchers on the ground. It was a process of negotiation between fine-grained representations and more abstract generalizations.

Ultimately, the key lesson of my project was the importance of establishing a well-designed database schema from the beginning. The schema formed the foundation of the application, which flowed down to backend and frontend development. It affected serialization, form design, and the API structure. A well-designed schema had the potential to unlock new information and help discover relationships in the data, which I hope my project has done.