# Inventorium: A RESTful Inventory Management API in Haskell

## Description and Motivation

During the Spring 2015 semester, Dr. Hayes introduced our software engineering class to XXXX and XXXX, who are involved with managing technology for Powell County Schools. In an experiment in software team management, the class organized into subgroups to design and implement a single software package to aid in tracking the significant inventory of desktop computers, laptops, tablets, and other pieces of technology that XXXX and XXXX were tasked with managing.

For various reasons, the project was unsuccessful. However, being from Eastern Kentucky myself, I was interested in contributing to the project and seeing something really useful delivered to the end users in Powell County.

During our software engineering course, my subgroup was primarily responsible for the design of the application and, later, the management of the rest of the development subteams. Our design was a typical three-tiered architecture:

* a browser-based client,
* a web API,
* a relational database.

We made specific technical decisions based on the realities of the large team. Our primary criterion was finding technologies with which our team members had prior experience, but secondarily, we needed technologies that were accessible in both paradigm, documentation, and availability of learning resources.

For these reasons, we chose to implement the web API in PHP, a widely-available dynamic programming language well-suited for web development. We knew a basic RDBMS would be the best choice for the database layer and chose MySQL for its ease of setup. Finally, we chose a combination of Twitter Bootstrap and Backbone.js to implement the browser-based client. Twitter Bootstrap is styling and behavioral library, which let the UI team focus on the UI despite relatively shallow web design skills. Among the many JavaScript frameworks, Backbone.js was the one the other front-end subteam felt most comfortable with.

Perhaps the most successful technical choice in the initial design was the use of two software systems, Vagrant and Ansible, to provide consistent, reproducible virtual development environments for the team. Vagrant offers a way to create, manage, and boot virtual machine images while Ansible is used to describe and execute provisioning strategies for remote systems, virtual or otherwise. Prior to all the subteams beginning their work, our team set up a virtual environment that could be reproduced consistently by individual team members with just a few commands.

These technologies were, for the most part, ones that I was very comfortable with. In my day-to-day work doing web application development, the so-called LAMP stack (Linux, the Apache web server, MySQL, and PHP) is very common. This was useful, since I would be able to answer any questions that came up in the process.

However, that same work experience has taught me that many aspects of this technology stack are woefully inadequate. For this reason, I chose to explore re-implementing this basic architecture while making technology choices based on technical factors rather than the expediencies required by a time-constrained, large-group project.

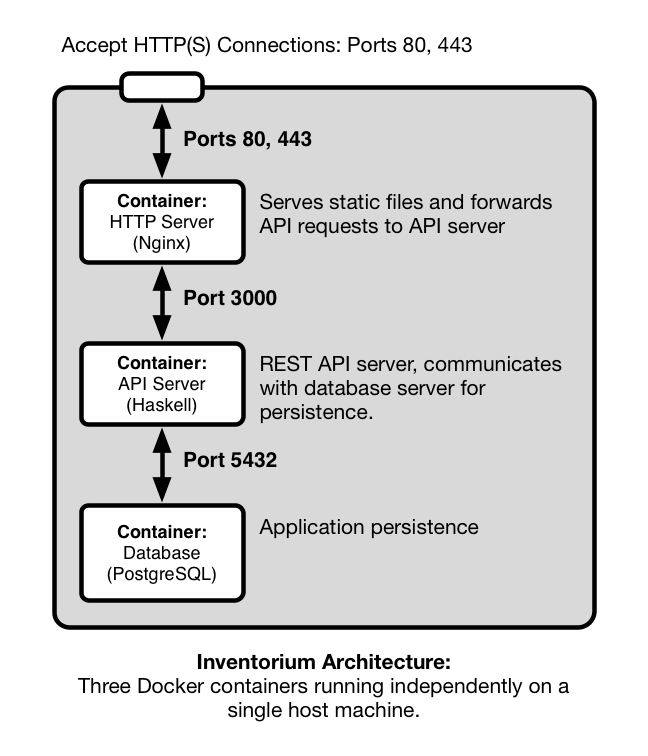
## Architecture

As I said, the notion of using virtual machines for development was really the most successful part of the original project. With surprisingly few issues, around 20 students were able to start contributing code to the project despite different operating system configurations and and experience with the technology stack.

However, the environment we used has a potentially serious drawback. It consisted of a single VM that contained both the web server, responsible for serving both the web API and static files (HTML, JavaScript, CSS, and images) for the client, and the database server. In a production scenario, this would clearly be problematic for scalability.

Ideally, the API server, the static web server, and the database server should all be separate. This offers more flexibility: as usage increases, multiple API servers could be run behind a load balancing mechanism or multiple database servers could be spun up to handle queries, via replication. Of course, if managing a single reproducible development environment is a challenge, managing several independent environments certainly requires an automated, reproducible solution.

I chose to solve this problem using a technology called Docker. Docker is a so-called "containerization" platform. Essentially, individual services (an API server, a database, etc.) can be packaged as containers: lightweight, bare-bones VMs that can (but are not required to) run on the same host machine. Containerized services are completely isolated, eliminating issues where, for example, an installed dependency for your database server is incompatible with your the API server. The containers can still communicate over the network by explicitly mapping network ports within the containers.



Inventorium Architecture

Inventorium, then, requires a container for the static web server, which was configured to forward API requests to the container that runs the API server. Similarly, the API server's container communicates with the database server. Although this was not implemented, there is very good support for deploying containers to cloud platforms like Amazon Web Services. This is an advantage, because it would allow Powell County Schools to make use of Inventorium cheaply, without needing a dedicated server, and avoiding potential issues with deploying to some shared server in the school district's IT office.

The web server container runs the Nginx web server. The original implementation used the Apache web server for simplicity. Apache uses OS-level threads to handle each web request, which introduces quite a bit of overhead for each request. This is less important when each request requires a great deal of CPU time (e.g., when responding to a complex API request), but when handling many relatively simple requests (for static files or simply proxying requests to an API server), Nginx's single-threaded, event loop-based architecture is offers higher throughput and better responsiveness in this use case.

I chose to carry on using a RDBMS for the project, rather than opting for a NoSQL solution such as a document-based database like MongoDB. Ultimately, I chose to use PostgreSQL rather than MySQL. While MySQL is an extremely common choice for small to medium sized web applications, but although it has made improvements in the past few years in terms of standards compliance, transactional safety, and data integrity, PostgreSQL is still regarded as better in these areas. One potential area

The final choice, and certainly the focus of both my development effort and the technical discussion here, regards the implementation of the API server. I chose to use Haskell, a strong- and statically-typed pure functional language that compiles to native code. Haskell's pedigree is decidedly academic and it is often derided as being overly theoretical, but as I will discuss below, it has a number of features that make it appealing for what is often called "line of business" software and not just writing compilers or experimenting with type systems and programming language theory.

## The Web API

Generally, the purpose a web API is to expose the operations of the business domain over HTTP. We use an architecture called Representational State Transfer (REST) to accomplish this. Described by Roy T. Fielding in his 2000 PhD dissertation, REST is an architectural style that emphasizes building around the fundamentals of the web: HTTP and hypermedia (i.e., navigable links between entities).

In a RESTful architecture, domain objects (and collections or other aggregates of those objects) are *entities* (in the HTTP sense), i.e., resources addressed by URLs. They may be available in many representations: an image resource might be available in a JPEG representation or PNG. HTTP's methods (verbs like GET and POST) describe the operations available. A GET request is a request for a representation of a resource, i.e., a read operation. A POST request takes a representation of an entity and requests that the application store it, i.e., a create request.

Other aspects of the API are all handled through the well-defined channels of HTTP. Options and parameters can be specified using URL query string parameters and/or HTTP headers. Similarly, errors are reported via HTTP status codes. If a user requests an entity that does not exist, the result should be the well-known 404 response. If the entity exists but the user requests an representation the server does not know how to provide, the API should instead respond with HTTP's "406 Not Acceptable" error code.

REST can be a restrictive architecture, but its principle of offering a "uniform interface" to APIs is powerful. For example, HTTP defines GET requests to be *idempotent* operation, meaning that a GET request should not change the state of an entity on the server. Thus, if an application properly implements the REST architecture, you get caching of GET requests for free via standard HTTP caching mechanisms.

The inventory management domain fits naturally within the REST scheme. The system tracks facility information (buildings and the rooms within them), inventory items and their properties (e.g., a laptop's CPU, memory, and hard drive specs), and "check-in" records that indicate a given inventory item was logged as being present in a given room at a given date and time.

The API organizes this domain with a system of hierarchical URLs. For example, facility information is organized into four entities addressed by different URLs:

1. /api/buildings: the collection of building entities
2. /api/building/:buildingID: an individual building entity with the given ID
3. /api/building/:buildingID/rooms: the collection of rooms in a given building
4. /api/building/:buildingID/rooms/:roomID: an individual room entity

URLs (b) and (d) address individual entities. HTTP GET requests retrieve a representation of that single entity, while HTTP PUT updates it (with the new details included in the body of the HTTP request), and HTTP DELETE deletes the record (if allowed by business logic rules).

URLs (a) and (c) address *collections* of entities. HTTP GET retrieves a representation of the entire collection and POST creates a new entity within the collection (with the details of the new entity in the request body). HTTP DELETE is not supported.

## Implementation

Before we discuss the specific details of the web API's implementation, I will present a short example that illustrates a few of Haskell's features. The following code sample implements uses parametric polymorphism to implement a homogeneous linked list and a function over that data type.

data List a = Cons a (List a)  
 | Nil  
 deriving (Eq)  
  
length Nil = 0  
length (Cons head tail) = 1 + length tail

We define a data type List that takes a single type parameter a. This is a homogeneous list, and every element will be of that type a. List is a recursive, algebraic data type, since it is the *sum* of two cases: a List is either a Cons cell (to use the Lisp terminology) that contains a value of type a plus a pointer to another List; or it is Nil, the empty list. Note that Cons is itself a *product* of two types, a, the type of items in the list and, recursively, List a.

The deriving keyword instructs the Haskell compiler to generate an implementation for the Eq typeclass. Typeclasses are analogous to Java interfaces and a datatype must implement Eq's interface to be usable with the == equality operator. With this automatic derivation, lists containing any data type that implements Eq can be compared for equality.

Finally we define a function length on List. We use a feature called *pattern matching* to define the two cases for the function. If length is applied to Nil, the result is zero (an empty list has length zero). Otherwise, we have a Cons cell with a head element and a tail list and the length is one plus the length of tail.

This small example demonstrates a number of Haskell language features that have important practical implications. Algebraic data types are more expressive than enumeration types and they make the pattern matching in our function definition possible. They also allow the compiler inform us when our pattern matches are not exhaustive, which often indicates a bug and to systematically generate code for the typeclass implementations as we saw with Eq above.

Parametric data types (known as *generics* in some languages) are also powerful. With our parametric list type, we can work with Lists of Strings, Integers, or any custom data type and use all the existing List code while still enjoy the security of knowing that the compiler can tell the difference between a List of Strings and a list of Integers.

Parametricity in combination with Haskell's default purity can be very powerful for reasoning about your code. A function of type a -> a, i.e., a function that takes a value of some type a and returns a value of that same type, can only have one implementation in a pure language: the identity function. This is a very trivial example, but the Haskell ecosystem is filled with libraries where parametricity and purity allow their maintainers to reason about the correctness of their code over and above simple type safety.

Consider our definition of the length function. Notice that despite all of the discussion about the language's type system, We did not actually have to add a type signature! Because Haskell makes use of *type inference*, it is actually able infer that length has a type of List a -> Integer just based on the types of the patterns and return values used in the definitions. Anyone who has written Java code that looked like HashMap<Integer, String> = new HashMap<Integer, String>() will appreciate the ability to use an expression and let the compiler infer the type and make sure it is consistent. It is standard practice to annotate top-level functions with a type signature, both for documentation purposes and as an early sanity check, but in most cases they are unnecessary.

Finally, we should take a closer look at what are perhaps Haskell's most radical features: *laziness* and *purity*. Laziness, or non-strict evaluation, means that when evaluating an expression, subexpressions are only evaluated as needed. The C language uses lazy evaluation to short circuit operators like && and ||, but Haskell uses this evaluation strategy everywhere by default. In practical terms, evaluation is driven by demand. For instance, we could cleanly define an enormous (or even infinite) data structure like a game tree, and it will only be evaluated as deeply as its consumer requires.

Implementing lazy evaluation in a standard imperative program would be quite tricky, if not impossible. As a result, Haskell functions are pure, meaning they have no side effects (direct manipulation of memory, I/O, etc.) and always return the same result given the same parameters. This property is nice for reasoning about code and it actually enables the Haskell compiler to make optimizations that would be impossible in an impure language.

Of course, real software (like web API servers) needs to do things like read user input or talk to the network. Haskell exposes these in a rather interesting way. Values of polymorphic type IO a represent instructions for the Haskell runtime to perform some operation that may have side effects that will yield a value of type a.

For example, getChar is a function of type IO Char. Calling getChar does not return a Char. Instead, getChar is a pure function (as all Haskell functions are) that returns an instruction that, when given to the runtime, will read a character from standard input and yield it to subsequent IO actions. Primitive operations like reading a character can be sequenced together to create more complex operations like reading a line from an input stream.

IO is an example of a *monad*, a common abstraction in Haskell. A monad can be thought of as a computational context that supports sequenced operations. Other examples include sequenced computation with mutable or read-only state, non-deterministic computations, or computations that may or may not yield a result. Haskell also supports the notion of *monad transformers*, which compose multiple monads together into a *monad stack*. In essence, sequencing and custom monad stacks allow us to define our own custom imperative sub-languages: I/O with read-only configuration, non-deterministic computation with short-circuit failure, or whatever the application needs.

### Libraries

Three Haskell libraries were pivotal to the implementation of the web API: Servant, for defining and implementing RESTful APIs, plus Persistent for database access.

**Servant**

The core concept in Servant is to express a REST API as a data type. That data type forms a specification of the API that can be used by the compiler to guarantee that the functions that implement the API conform to the specification. Additionally, Servant's code for dispatching HTTP requests uses this specification for routing those requests to the appropriate handlers, decoding request details into Haskell values, performing content type negotiation, validating HTTP headers, encoding resulting values into appropriate content types, and packaging the response, including using appropriate HTTP response codes.

Servant uses some of the more advanced extensions to the Haskell langauge that have been implemented in the de facto standard compiler, GHC. Importantly, it uses type-level literals allowing strings, natural numbers, and lists to be used in type definitions. I will not go into the details here, but the concepts are discussed further in [1].

Here is a fragment adapted from the project source code:

type FacilitiesApi = "api" :> "buildings"  
 :> Header "Authorization" AuthToken  
 :> ReqBody '[JSON] Building   
 :> Post '[JSON] BuildingDetail

This defines a data type that specifies a single API endpoint available at the URL /api/buildings. It specifies that the HTTP header Authorization should be captured for use by the endpoint's handler, interpreted as a value of type AuthToken. It requires a the request body contain JSON that can be decoded into a Building value. Our type also specifies that this endpoint responds to HTTP POST requests and will return a JSON encodeding of a BuildingDetail value if successful.

The value of the Authorization header and the decoded Building value will be captured and passed to the handler function, again adapted from the project source:

postBuilding :: Maybe AuthToken -> Building -> Handler BuildingDetail  
postBuilding auth building = do  
 checkAuthToken auth  
 validateBuilding Nothing building  
 buildingId <- runDb $ insert building  
 return $ BuildingDetail { building = Entity buildingId building  
 , rooms = Nothing }

The endpoint's handler is simply a function. Its first parameter is a value of type Maybe AuthToken. Maybe is a sum type that represents a value that may or may not be present. So, if the request supplied an authorization token "abc123", we would receive the value Just "abc123". If none was supplied, we get Nothing.

The second parameter is the decoded Building value we got from the request's body.

The return value is of type Handler BuildingDetail. Handler is our application's custom monad stack which supports access to read-only configuration information, exceptions, and general I/O, primarily for database access. In other words, our API endpoint handlers are pure functions that have configuration implicitly wired through them, can throw exceptions, and return instructions for how the Haskell runtime should perform I/O in order to build up a value of type BuildingDetail.

We will cover more specific details about the implementation below, but it is worth reflecting on how Servant worked in practice. There are many web frameworks available, like Ruby on Rails, Django for Python, and Laravel for PHP. One important criteria for judging this frameworks is their ability to abstract the common details of the HTTP request/response lifecycle. Servant does this as well as, and perhaps better than, any framework I have worked with. Consider this workflow:

* The request URL is parsed and mapped to an appropriate handler based on the request method (GET, POST, etc.)
* HTTP headers are parsed and, where applicable, marshalled into native data types:
  + The request Content-Type header is checked to ensure that the body of the request is in a format that our application is willing to accept
  + The Accept header is checked to ensure that the client will accept a response in a format our application is willing to deliver
* URL path fragments (such as the ID of a building) and query string parameters are parsed and marshalled into native data types
* The relevant path fragments, query string parameters, and headers are passed to the handler.
* If successful, the return value of the handler is marshalled into an appropriate encoding based on the request's Accept header.

There are two important themes here. The first is that in all cases, our application logic ought deals with semantically meaningful domain values rather than simply strings or associative arrays (probably full of strings themselves!) and our framework should abstract the details of marshalling raw strings from the request into our domain data types and from our domain data types into raw strings in the response.

The second is that the framework should abstract the details of handling as many common HTTP error cases as possible. If we can declaratively define the content types our application is willing to accept, the framework should use that information to return 415 Unsupported Media Type without our application code's direct intervention if the client gives us a content type we do not support of return 400 Bad Request if the request body cannot be parsed into a suitable value. Of course, there will always be error cases that only our application can know about. If a request is made for a record that is not in the database, we should not expect the framework to know to return 404 Not Found, but we should expect it to return that error response if the request URL does not map to any known handler.

Servant is quite successful in these two regards, and I would argue that the Haskell language contributes to that succes. It does not surprise me to see a pervasive focus on using meaningful domain data types in a strongly-typed language like Haskell, but Servant leverages Haskell's features to good effect, using typeclasses to automate marshalling to and from domain data types. For example, if you have a data type that implements the ToJSON and FromJSON type classes, that data type can be accepted in the body for requests with the Content-Type: application/json header and returned in the response for requests with the Accept: application/json header.

Similarly, any language could support a framework that offers a declarative way to define acceptable content types, required headers, and so on, but Servant's radical approach of using Haskell's type system to make that declarative specification not just a source of runtime information but a binding, compile-time contract for handler functions is extraordinarily powerful. Servant also supports using that same type-level declarative specification to drive tools that generate API documentation and client code for other languages like JavaScript.

While working with Servant, I did come across a few odd corner cases, mostly involving unexpected response codes in certain error cases. It is possible Servant understands the HTTP specification better than I do, but Servant is a relatively immature library. However, despite some shortcomings, what Servant offers is a library that takes the classic software engineering mantra, "Don't Repeat Yourself", stretches it to its limits, and lets the compiler enforce it all. I was very impressed.

**Persistent**

As we have discussed, we use PostgreSQL for our application's data storage layer. To interface with it, our application issues queries to the database server to retrieve records and execute inserts, updates, and deletes. These queries are expressed in PostgreSQL's dialect of SQL, or Structured Query Language. These textual queries are transmitted over the network where they are parsed and executed by the database server.

In many programming langauges, the available database bindings force the programmer to build up dynamic queries using standard string operations. For example, we might need to concatenate an entity ID along with some SQL fragments to build a query. In PHP:

$query = 'select \* from buildings where id = ' . $buildingID;

Unfortunately, this is a very dangerous approach. First, we have no guarantee that the SQL we generate will be syntactically valid and we will not find out about our mistake until runtime. Additionally, we have to worry about correctly escaping the strings we concatenate together. Consider this example where we search for a substring in a building's name field:

$query = 'select \* from buildings where name = "' . $searchTerm . '"';

In addition to being somewhat difficult to read, there is a more serious problem. Suppose that $searchTerm contained the string 'hello "world"'. The resulting SQL would be:

select \* from buildings where name = "hello "world""

This is a syntax error, which will only show up when a user enters that or a similar problematic string into an input field. This is even worse than just a bad user experience decision, it can be abused by a malicious attacker. Suppose $searchTerm contained 'foo"; drop table buildings; --':

select \* from buildings where name = "foo"; drop table buildings; --"

When executed, this query will happily drop the buildings table. That is unlikely to be an operation you wanted to grant to all users. This is an example of the classic SQL Injection vulnerability.

Of course, SQL injection can be prevented. The most lightweight solution is to use proper escaping when building up dynamic queries, but if this must be done manually, it will be error-prone.

Another option is to use stored procedures rather than building truly dynamic queries within application code. However, dynamic queries are sometimes the the most elegant way to express a database operation and keeping stored procedures in the database in sync with version control can require unwanted overhead in terms of build and deployment processes.

Many languages provide database bindings that help make building dymamic SQL queries safer, but we are still left with the possibility that the SQL we generate will be invalid, either because of syntactic issues or simply because we refer incorrect tables or columns. We use the Persistent library in this project to help deal with our interactions with SQL.

Persistent is a database-agnostic library that uses metaprogramming to generate a typesafe interface for basic data access. However, because Persistent was designed to support both relational database systems like PostgreSQL and document-based database systems like MongoDB, it does not directly support important relational features such as performing joins between tables.

Since we are using a relational database, we would like to be able to make use of those features. There is another library called Esqueleto which builds on Persistent to offer those relational database features at the cost of some degree of type safety, but it was not used significantly in this project. Importantly, as of the time the project was implemented, Esqueleto could not actually guarantee type safety: it is possible to write queries in Esqueleto that fail at runtime, even when the underlying Persistent schema corresponds to the schema as it exists in the database.

Persistent works by using a text-based schema definition to generate Haskell code for interfacing with those tables. Our project's schema defition is found in site/src/model. Here is an excerpt:

Room json  
 building BuildingId  
 name Text  
 description Text  
 created UTCTime default=CURRENT\_TIMESTAMP  
 updated UTCTime default=CURRENT\_TIMESTAMP  
  
 UniqueNameInBuilding name building  
  
 deriving Eq Ord Show

Here we define the schema for our rooms table. Persistent's metaprogramming will generate a new data type RoomId to represent the primary key for this table, along with a record data type that has fields for a BuildingId (the primary key in the buildings table), two text fields for the room's name and description, and two UTC timestamps for the time that the record was created and the last time it was updated.

The deriving clause tells Persistent to derive implementations for the Eq, Ord, and Show typeclasses for this record type. Additionally, the json notation tells Persisten to derive implementations for the ToJSON and FromJSON typeclasses for automatic marshalling into and out of JSON representations.

Finally, the line UniqueNameInBuilding expresses a unique key constraint on this table: no two rows should have the same name and building value. In other words, no two rooms in the same building should have the same name. UniqueNameInBuilding will now be available as data constructor that takes two Text values and returns a value that can be used to uniquely retrieve a row in the database.

We make use of this schema definition in site/src/Types/Model/Persistent.hs:

share [mkPersist sqlSettings, mkMigrate "migrateAll"]  
 $(persistFileWith lowerCaseSettings "src/model")

At compile time, GHC its metaprogramming engine, Template Haskell, and persistFileWith will read in our schema definition and generate corresponding Haskell code, with the specific mappings between generated Haskell identifiers and database tables being governed by the lowerCaseSettings value.

This file also defines a function runDb that we will see in upcoming examples:

runDb :: SqlPersistT IO a -> Handler a  
runDb query = do  
 pool <- asks getPool  
 liftIO $ runSqlPool query pool

Persistent's database access functions result in query values that need to be executed in a particular context. In particular, we need access to connection details. Our Handler monad offers access to that configuration information, so essentially runDb takes a Persistent query that would yield a value of type a and transforms it into an action in our Handler monad that will get a database connection from the application's connection pool, use it to execute the query on the database, and yield a result of type a.

Persistent provides typesafe functions for using our the generated Haskell data types to interact with our schema. For example, get retrieves a record by primary key. Its type is:

(MonadIO m, backend ~ PersistEntityBackend val, PersistEntity val) => Key val -> SqlPersistT backend m (Maybe val)

This is one of the most intimidating type definitions we have seen, but it is at the core of Persistent's database-agnostic, typesafe approach. There are two type variables, m, val, and backend.

m is the monad stack that the query will be executed in. In particular, the constraint MonadIO m means that it needs to be a monad stack built atop the full IO monad. This should not be a surprise, since we need to be able to interact outside the Haskell run to talk to the database server.

val represents the data type the query should yield. If we are retrieving a Building record, val is unified with the type Building. The constraint PersistEntity val indicates that val is a type that implements certain under-the-hood operations that Persistent requires. These implementations are generated by Persistent's metaprogramming automatically, but you write them yourself.

Finally, backend is the database backend that Persistent will be working with. The constraint backend ~ PersistEntityBackend val indicates that the backend needs to be one for which the val data type has proper implementation of the PersistEntityBackend typeclass. This allows Persistent to guarantee at compile time that the data type we are retrieving is one that can be properly marshalled from the database into a native Haskell data type.

Given those facts about the type variables, we can interpret the rest of the type signature. get takes a value of type Key val, which is the data type that represents a primary key for a value of type val. If we are trying to retrieve a Building, we need to pass a Key Building value, which is just a synonym of BuildingId.

The result is a monadic action that will yield a Maybe val, since the primary key may not exist in the database. Persistent queries execute in a monad stack that layers ready-only access to details about the backend on top of a monad that supports IO. As we saw, we will use our runDb function to lift that monadic action into an action in our Handler monad, still yielding a Maybe val in this case.

Now we can make database queries in our handlers. Here is a function in our Handler monad that attempts to retrieve a Building record from a BuildingId that fails with an HTTP 404 error if no such record exists:

fetchBuildingOr404 :: BuildingId -> Handler Building  
fetchBuildingOr404 buildingId = do  
 maybeBuilding <- runDb $ get buildingId  
 case maybeBuilding of  
 Nothing -> fail404 "Building not found."  
 Just b -> return b

Using Haskell's monadic do-notation, this almost looks like imperative code. We use runDb to execute our query get buildingId and bind the result to the identifier maybeBuilding. Then we pattern match on that value. If the result is Nothing because no value was found, we use fail404 to short-circuit evaluation and fail, which will result in a 404 response by our application. If we get Just b, we found a building that matched the primary key, namely the value b, and yield that.

### Code Organization and Walkthrough

As we walk through the code, please refer to the project's GitHub repository:

https://github.com/zachmay/inventorium

**Docker Container Definitions**

Recall that we use Docker to host isolated, virtualized components (i.e., containers) for each of the primary components of our system. The top level of the repository contains directories for documentation and other utilitarian aspects of the project, but our primary concern will be with the database/, webserver/, and site/ directories and the file docker-compose.yml.

These three directories contain the details for Docker containers that will run our database server, our web server, and the API server. Each of these directories contains a file Dockerfile that instructs Docker how to provision and set up the container. For example, database/Dockerfile simply says to build a container based on version 9.4 of the official PostgreSQL Docker image. webserver/Dockerfile is more involved, installing the nginx web server and performing setup tasks such as copying configuration files and exposing standard HTTP ports. site/Dockerfile builds from a standard Haskell image and does some configuration so that the API server container can be used as both a virtualized development environment and host the final executable.

Finally, docker-compose.yml is a configuration file for the docker-compose tool, which orchestrates bringing all three component containers into service simultaneously and ensure that they are configured to work with each other, in this case by specifying environment variables and exposing network ports between containers so that the API server can communicate with the database.

**Cabal Configuration**

Before we move to the Haskell source code, we need to look at one more configuration file: site/inventorium.cabal, a configuration file for Haskell's Cabal package manager. It describes the package's legal details, dependencies, and compiler options. Once we have a "Cabal file", we can use cabal to install all of the libraries our package depends on and compile our own code using the appropriate options.

One particular set of options, default-extensions describes all of the language extensions our package will use beyond the Haskell 2010 standard [2]. These extension are required to make use of the Servant library, but they are not part of the language standard. Because these extensions are only supported by the GHC compiler, the project is not portable across compilers. However, GHC is the de facto standard and is the only Haskell compiler that is widely used and maintained.

**Types**

Haskell's focus on strong, static typing makes the API's data types a good place to start exploring the Haskell source code. Our application's domain model entities include the following:

* Buildings that can contain a collection of Rooms.
* ItemTypes that describe a type of inventory, e.g., PCs, tablets, or laptops.
* Each ItemType also has a list of ItemTypePropertys that model the fields that a user can enter to describe that item. For example, laptops might have a field to enter the hard disk size, amount of RAM, manufacturer, and purchase year.
* An Item represents an inventory item (like a computer or tablet) that is tracked in the system. It has an ItemType and is described by a a collection of ItemPropertys that correspond to the ItemType's ItemTypePropertys.
* The inventory history of an Item is tracked by a collection of CheckIns that associate an Item with a Room at a given date and time. The Item's current assigned location is the Room of the most recent CheckIn.

This domain model is described in the file site/src/model. This definition file is used by the Persistent library's compile-time metaprogramming facilities to generate Haskell code that defines the appropriate data types. Here is the part of that file that generates types corresponding to the Building entity:

Building json  
 name Text  
 description Text  
 created UTCTime default=CURRENT\_TIMESTAMP  
 updated UTCTime default=CURRENT\_TIMESTAMP  
  
 UniqueBuildingName name  
  
 deriving Eq Ord Show

We define a type Building, instructing Persistent to generate code to allow us to automatically convert to and from JSON via the ToJSON and FromJSON typeclasses. We define four fields along with their data types and, in the case of the created and updated fields, their default values. Then we define a unique key for this entity: we can use UniqueBuildingName to query for a unique building record. This corresponds to a unique key constraint in the database definition. Finally, we instruct Persistent to use Haskell's typeclass derivation to automatically create implementations of the Eq, Ord, and Show typeclasses for the Building type.

This model definition is referenced by the following lines from site/src/Types/Model/Persistent.hs:

share [mkPersist sqlSettings, mkMigrate "migrateAll"]  
 $(persistFileWith lowerCaseSettings "src/model")

At compile time, the Haskell code generated by these lines is spliced into our own Haskell code and compiled. In particular, each of our domain model entities is modeled as a Haskell record type and a distinct identifier type is defined. In essence, even if all of the primary keys for our entities are really just integers, BuildingId and ItemId are distinct types, so the compile will not allow us to query for items using a building's ID.

Perhaps the one drawback is that all of the resulting definitions are simply dumped into the same module. For this reason, the modules defined in site/src/Types/Model/ import the module defined by Persistent re-export things as needed in a more organized way. In addition, each of these modules defines some related data types and typeclass instances. To illustrate, we will walk through site/src/Types/Model/Building.hs.

instance ToText BuildingId where  
 toText k = pack . show . fromSqlKey $ k

Here we define the ToText instance for BuildingIds. This is used by Servant to convert a building ID that we got from the database. Essentially, this is just a series of manual type conversions: we convert the BuildingId into an integer, turn that into a String, and then convert that into a Text object, where Text is Haskell's more performant string data type. For historical reasons, String is a linked list of characters and is quite inefficient.

instance FromText BuildingId where  
 fromText t = toSqlKey <$> (readMaybe . unpack $ t)

This is the corresponding typeclass instance that takes a Text representation of a building ID and returns it as a BuildingId. The typeclass definition takes into account the possibility that the parse will fail, so our implementation must do that as well.

data BuildingDetail =  
 BuildingDetail  
 { building :: Entity Building  
 , rooms :: Maybe [Entity Room]  
 }

The Building datatype generated by Persistent contains all of the detail that are stored in the corresponding database table, but not related records, in particular the rooms that are associated with the building. This composite datatype groups a building record with a list of room records. We use Maybe to allow for the possibility that the collection of rooms might not be populated for performance reasons.

instance ToJSON BuildingDetail where  
 toJSON (BuildingDetail { building = b, rooms = rs }) =  
 maybeUpdateWithAll (toJSON b) [("rooms", toJSON <$> rs)]

This implementation of the ToJSON typeclass enables Servant to automatically marshall a BuildingDetail value into a JSON encoding for returning in an API response. Essentially, we convert the building record to JSON using Persistent's automatically generated ToJSON implementation and update the resulting record with a field for the building's rooms if present.

data BuildingSortBy = BuildingSortByDateCreated  
 | BuildingSortByDateUpdated  
 | BuildingSortByDescription  
 | BuildingSortByName  
 deriving (Bounded, Enum, Eq, Ord, Show)  
  
instance ToText BuildingSortBy where  
 toText BuildingSortByDateCreated = "created"  
 toText BuildingSortByDateUpdated = "updated"  
 toText BuildingSortByDescription = "description"  
 toText BuildingSortByName = "name"  
  
instance FromText BuildingSortBy where  
 fromText "created" = Just BuildingSortByDateCreated  
 fromText "updated" = Just BuildingSortByDateUpdated  
 fromText "description" = Just BuildingSortByDescription  
 fromText "name" = Just BuildingSortByName  
 fromText \_ = Nothing

Here we define a data type that will describe how a list of building results will be sorted. So, if we receive a request for /buildings?sort=name, we have a type-safe value to represent that option. We define ToText and FromText so that BuildingSortBy values can be converted to and from text when Servant parses a request URL.

data BuildingExpand = BuildingExpandRooms  
 deriving (Bounded, Enum, Eq, Ord, Show)  
  
instance ToText BuildingExpand where  
 toText BuildingExpandRooms = "rooms"  
  
instance FromText BuildingExpand where  
 fromText "rooms" = Just BuildingExpandRooms  
 fromText \_ = Nothing

Similarly, we want the users of our API to be able to manually select what related information is returned by the API. In this case, API clients can specify whether or not to retrieve the rooms attached to a building, saving a database query if the room information is not needed. As with BuildingSortBy, we define how to convert BuildingExpand values to and from Text so that Servant can parse them and hand over type-safe values to our implementation code.

We will see how these types are used when we look at the type-level definition of our application's API in the following section.

**Type-level API Definitions**

For organizational reasons, the project's type-level API definition is spread across three files. However, these three sets of definitions are combined and exported by the Types.Api module defined in site/src/Types/Api.hs. We will look at a few examples dealing with authentication and managing facility information.

In site/src/Types/Api/Authentication.hs, we define a single endpoint:

type AuthenticationApi =   
 "api" :> "auth"  
 :> ReqBody '[JSON] AuthRequest   
 :> Post '[JSON] AuthResponse

The official GHC documentation on type-level literals [3] goes into more detail, but here the "api" is a string literal at the *type* level, not the *value* level. Servant defines a type-level operator :> that joins two types together. The result is a chain of elements that our type-level API definition is concerned with.

In this case, "api" :> "auth" corresponds to an API endpoint that will match a URL like /api/auth. Going further down the chain of :> operators, ReqBody declares that we require requests to this endpoint havea request body, in JSON format to be decoded into a value of type AuthRequest. Then Post declares that the endpoint should respond to HTTP POST requests and that the response should be a value of type AuthResponse that can be encoded into JSON.

AuthRequest and AuthResponse are two data types defined in site/src/Types/Misc.hs. An AuthRequest is just a record containing a user name and a password, while an AuthResponse is a record with a status indicator and the authentication token or error message, as appropriate.

While there is only one endpoint definition for authentication, the facility management definitions have several. The file site/src/Types/Api/Facilities.hs exports the type-level API definitions for managing rooms and buildings. Here is a fragment adapted from that file:

{- /api/buildings -}  
"api" :> "buildings"  
 :> Authorized  
 :> QueryParams "sort" (SortField BuildingSortBy)  
 :> QueryParams "expand" BuildingExpand  
 :> Get '[JSON] [BuildingDetail] :<|>  
  
{- /api/buildings/:buildingId -}  
"api" :> "buildings" :> Capture "buildingId" (Key Building)  
 :> Authorized  
 :> QueryParams "expand" BuildingExpand  
 :> Get '[JSON] BuildingDetail

Here we see another operator defined by Servant: :<|>. This operator combines two endpoint definitions into a larger data type, ultimately resulting in a single data type that defines our application's entire API.

These two definitions are similar, but introduce a few new features. Authorized is a shorthand for Header "Authorization" AuthToken (defined in site/src/Types/Misc.hs) that declares that an endpoint cares about the HTTP Authorization header. In this case, we expect the plain-text value of that header to be converted to a value of type AuthToken and for that AuthToken to be exposed to our handler function, which will check it to ensure that the user is authenticated before completing the request.

We also see QueryParams, which is one of Servant's options for handling URL query string parameters, the the key/value pairs after the ? in a URL. For example, QueryParams "expand" BuildingExpand defines an endpoint is that the Servant framework will handle by looking for expand= followed by a comma-separated list of strings that correspond to BuildingExpand values. That list will be exposed to the handler function which must accept a list of BuildingExpand values as one of its parameters. A similar option, QueryParam is available that will parse a single optional parameter, passing it as a Maybe value, since the parameter may not be specified.

Finally, the second endpoint demonstrates the use of Capture to expose a fragment of the URL to the handler. In RESTful APIs, it is very common to address resources using a hierarchical path, not unlike a directory structure. In our application, /api/buildings represents a collection of building resources (analogous to a folder containing several files), and /api/buildings/123 is the address of the building with ID 123. Capture instructs Servant to capture that fragment of the URL, convert it to a value of type Key Building (i.e., the data type of the primary key of Building records), and to pass that into the handler.

**Handlers**

Now we will look at how our API endpoints are handled. It is important to note that essentially all the work dealing with HTTP requests will be done at this point. In other words, the type-level API definition instructed Servant which headers and query string parameters to look at, how to parse things, and when to return many types of HTTP errors. That same type-level definition was also used to type-check our handler functions at compile time, but we still need write the code that takes the values extracted from the request and use them to make updates and return a response.

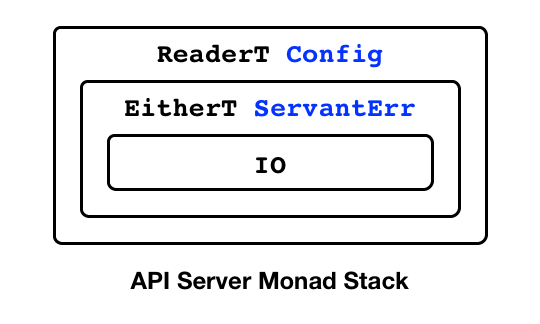
Much like our type-level API definitions, the handler definitions are spread across three files. In this case, the Handlers module in /site/src/Handlers.hs exports a combined value that groups all of our application's endpoints together. Of note, it appears that both of these files use the same operator, :<|>, to group things, but they are actually quite different. When grouping the type-level API definitions, :<|> is being used as a type-level operator. When grouping handlers, it's a value-level operator, not so different from + or \*, that combines functions.

In particular, given two endpoints a of type A and b of type B, a :<|> b will have type A :<|> B. This is part of the machinery for how Servant's type-level definitions get used to type-check handlers at compile time.

Consider this handler function for retrieving an individual building resource, corresponding to the type-level definition for the /api/building/:buildingId endpoint:

getBuilding :: BuildingId -> Maybe AuthToken -> [BuildingExpand] -> Handler BuildingDetail  
getBuilding buildingId auth expand = do  
 checkAuthToken auth  
 building <- fetchBuildingOr404 buildingId  
 rooms <- fetchRooms  
 return $ BuildingDetail { building = Entity buildingId building, rooms = rooms }  
 where fetchRooms = if BuildingExpandRooms `elem` expand  
 then do  
 rooms <- runDb $ selectList [RoomBuilding ==. buildingId] []  
 return $ Just rooms  
 else return Nothing

Referring back to the corresponding type-level definition, we know that Servant will capture a BuildingId from the URL, an optional AuthToken (hence Maybe), and a list of BuildingExpand options. As we can see, that corresponds to the three parameters of our handler function. The return type is perhaps unusual though: rather than return a BuildingDetail value, the return type is Handler BuildingDetail.



The Handler Monad Stack

This is because our handler functions are *monadic*. We will not address monads here, but our handlers actually return computations that execute in a particular context and yield the appropriate values. The definition of the Handler type is:

type Handler = ReaderT Config (EitherT ServantErr IO)

Handler is a *monad stack*. Reading from the inside out, Handler builds on the IO monad, which allows computations that perform arbitrary I/O, extends that with EitherT which allows computations to potentially fail with some error value (in this case, Servant's ServantErr type), then extending that with ReaderT allow read-only access to configuration information (in this case, a type that contains information about a pool of database connections our handlers can use).

Looking back at the definition of our getBuilding handler, we see that it uses Haskell's do notation to sequence monadic computations. Without going into detail, do allows us to sequence monadic computations in much the same way we would statements in an imperative program.

In this case, we call checkAuthToken to validate the supplied authentication token:

checkAuthToken :: Maybe AuthToken -> Handler ()  
checkAuthToken Nothing = do  
 consoleLog "Failed auth: nothing supplied"   
 failWith $ ServantErr 401 "Unauthorized" "Invalid authorization token" []  
checkAuthToken (Just \_) = do  
 {- Validation code omitted -}  
 return ()

This function also operates in the Handler monad, but it returns (), or "unit", a meaningless value we do not care about. In this incomplete implementation, we simply use pattern matching to determine whether an authentication token was supplied at all. If not, our computation fails with a ServantErr corresponding to HTTP 401 Unauthorized. This error result short-circuits the evaluation of subsequent computations and that ServantErr value is used by Servant to issue an appropriate HTTP response.

When an authentication token is present, we simply yield () which further computations can ignore and proceed normally. A full implementation would need to check against some data store to ensure that the authentication token is valid. If not, the computation would again fail with a similar ServantErr.

From there, we call fetchBuildingOr404, another computation in the Handler monad. This function takes a building ID and will query the database for it, either yieldng the resulting Building value or failing with a ServantErr corresponding to HTTP 404 Not Found.

Then we attempt to fetch the rooms associated with the building. We use a where clause to define that operation without cluttering the main flow of the function. If BuildingExpandRooms is among the options in expand, we pull in the associated rooms, if not we just yield Nothing to indicate that we did not attempt to retrieve them.

Finally, we create a BuildingDetail record that contains our Building (using Persistent's Entity type which ties a record with its database ID) along with the associated rooms if requested, and yield that as the result of our Handler computation. At this point, our code is done. Servant uses the type-level API definition to marshall values from Haskell data types into whatever encodings are needed and builds up the HTTP response to return to the client.

**Wiring Things Together**

With all these details in place, wiring our application together is not all that complicated. The API server executable starts up when the API server container is brought online. Its entry point is the main function in src/site/Main.hs.

main :: IO ()  
main = do  
 port <- lookupSetting "PORT" 3000  
 putStrLn "Starting Inventorium API server..."  
 pool <- makePool  
 let config = Config { getPool = pool }  
 runSqlPool doMigrations pool  
 putStrLn $ "Listening on port " ++ show port  
 run port $ logStdout $ app config

In Haskell, the main function always has type IO (): intuitively, main is a pure function that lazily returns a computation that will result in the value (), the "unit" value we have seen before. Our program actually gets executed when the Haskell run-time forces the full evaluation of that lazy value.

First, we need to know what port to listen to for API requests. In our application's architecture, API requests are forwarded from the HTTP server to the API server on a port defined in docker-compose.yml. This port number is assigned to an environment variable in the API server container. We use lookupSetting to retrieve this value:

lookupSetting :: (Read a) => String -> a -> IO a  
lookupSetting env def = do  
 p <- lookupEnv env  
 return $ case p of Nothing -> def  
 Just a -> read a

This IO action takes an environment variable name and a default value of type a. The type a must implement the typeclass Read so that we can convert any possible String value we retrieve into the native Haskell type a. We use lookupEnv to look up the value of the variable env in the current environment, yielding a Maybe String. If the environment variable was not defined, we get Nothing and yield the default value. If we find a value, we parse that string into a native value using the read function provided by the Read typeclass and yield the result.

Now we need to configure our database connection pool. Rather than repeatedly opening and closing database connections, we instead create a pool of them. When a handler needs to communicate with the database, it requests a free connection and releases that connection when it is finished. The makePool function creates our connection pool:

makePool :: IO ConnectionPool  
makePool = do  
 dbUser <- lookupEnvironment "DBUSER" "postgres"  
 dbName <- lookupEnvironment "DBNAME" "postgres"  
 dbPass <- lookupEnvironment "DBPASS" "postgres"  
 dbHost <- lookupEnvironment "DBHOST" "database"  
 dbPort <- lookupEnvironment "DBPORT" "5432"  
 let connectionString = pack $ intercalate " " $ zipWith (++)  
 ["host=", "dbname=", "user=", "password=", "port="]   
 [dbHost, dbName, dbUser, dbPass, dbPort] in  
 runStdoutLoggingT $ createPostgresqlPool connectionString 4

We use lookupEnvironment to retrieve the database connection details. It works like lookupEnv but takes a default value to return if the environment variable is not defined. We use those details to create a PostgreSQL connection string and use the PostgreSQL library's createPostgresqlPool to create a pool of 4 connections.

With our connection pool created, we create a Config value. Our application's Config type is just a record that stores a reference to our connection pool. We also pass that connection pool to doMigrations, which will automatically update the PostgreSQL schema on the database server based on the Persistent schema defined when we compiled the executable.

Finally, we use the function run to start up our application server. This function is provided by the Warp library, which offers a high performance, event-driven implementation of low-level HTTP server needs like network access, request parsing, and so on. We simply pass in our own request handling code.

The value app wraps our Servant API handlers into a value that Warp can interact with and we inject our read-only configuration value so that our handlers have access to the database connection pool. Additionally, we pass this into logStdout to direct any application logging to standard output.

## Testing

Due to time constraints, an exhaustive automated test suite was not implemented. Manual testing was performed on implemented endpoints based on a known initial database state. This could have been implemented as a series of automated integration tests, but there was not enough time to complete that.

Although we did not have enough time to implement any form of automated tests, two popular Haskell testing frameworks were researched.

QuickCheck is a property-based testing library. Relevant properties are expressed as predicates and QuickCheck randomly generates test cases and asserts that each of those properties holds. In the event a property fails to hold, QuickCheck is capable of searching through the space of test cases to find a minimal instance that still fails.

Property-based testing is a natural fit for Haskell. Because programs tend to focus on pure transformation of values by function application, the algebraic properties of those functions are often more relevant than test cases that set up some pre-conditions, execute a procedure, and then make an assertion about the post-conditions. Moreover, Haskell's purity by default means that repeatedly running tests is safe and we can be confident that the same code will not mysteriously stop working in a production environment.

Perhaps due to my inexperience with Haskell, it seemed like the bulk of my code was still fundamentally impure (i.e., it was ultimately executed in the IO monad), which to my knowledge makes QuickCheck unsuitable.

Another option is the HSpec library. In the vein of Ruby's RSpec, HSpec supports a testing and development paradigm known as *behavior-driven development* (BDD). In BDD, requirements are expressed in natural language and paired with an automated test that should prove the requirement is properly implemented. HSpec defines a DSL for describing requirements and related tests, including typical assertion-based tests and property-based tests. HSpec is suitable for testing impure code and can run set-up procedures before executing each test, e.g., resetting a database to some known state. Additionally, there is an experimental library called HSpec-WAI that aids in describing the behavior of web application endpoints.

## Challenges

Haskell is, in many ways, a strange language. Going from writing PHP at work to writing Haskell for this project, it seemed as though Haskell made a conscious effort to do *everything* differently: the functional paradigm rather than imperative or object-oriented code, a strict type system, lazy evaluation. Working with such a radically different technology stack created a high barrier to productivity.

Frequently during the course of this project, I came across a task that I did not know exactly how to accomplish. Often this was something that might have been easy to do in PHP, although probably without Haskell's safety. However, with a bit of research or experimentation, I found a solution and learned a lot in the process. In a school project, this is only a positive. In an industrial setting, these delays would certainly be more costly.

Another frustration that comes from my relative inexperience with functional programming is the issue of code organization. In object-oriented programs, there exists a very basic mechanism for organizing your code: the class. A data type and the operations on it are kept together in a class definition. Haskell has a module system that can provide a similar organizational framework, but I found that I had a tendency to freely define functions with much less attention to how they were organized. I suspect this would improve with more experience and Haskell at least makes refactoring a safer proposition.

## Lessons Learned

Despite the frustrations described above, I found that working with Haskell was a great development experience. I want to share one particular experience that really impressed me.

Most of the way through the process of developing the API, it became clear that there were some pretty serious shortcomings with the codebase, particularly how it was organized. There was too much code in the main file that was not relevant to bootstrapping the application server. A file describing the domain model and auxiliary datatypes was strateching into the hundreds of lines (which is a lot in a language as expressive as Haskell).

I decided to do a major refactor to imporve the situation. I moved a few definitions around, essentially "sketching out" how the code would be organized. Of course, the project no longer compiled. From there, I fixed compiler errors (there were a lot!) until the program compiled. The really impressive part was that one it did compile, all of the API endpoints worked just as they did before the refactor.

In my experience with PHP, a refactor of this magnitude would have a similar project a complete disaster. Coincidentally, we implemented a similar set of refactorings on a PHP web API at my job at the same time I was working on this project and the contrast between the experiences with those two code changes was profound. Seeing how pleasant refactoring could be made struggling with PHP very frustrating!

I think that the biggest factor that makes refactoring in Haskell so pleasant is the strong, static type system. PHP code frequently makes use of untyped associative arrays and uses strings or integers to pass around options or flags. Even when using a feature called *type hints* to be explicit about what data types a function parameter should be, type mismatches are not reported until runtime.

With Haskell, untyped associative arrays are replaced with strongly-typed record data types and a set of flags can be encoded as a sum type. Moreover, all of this is checked for consistency before the program even starts to run. With the Servant library and its DSL for specifying the type-level definitions of your API, the compiler is able to make even higher level guarantees about your code. I do not mean to be hyperbolic, but coming from PHP, this is a revelation and makes me a bit envious!

## Future Work

Much more work would still need to get this project to a point where it were useful to end-users. My main focus during the semester I spent working on this was the development of the API using non-standard techniques. As a result, the front-end is still entirely unimplemented.

However, in the time between starting the project and completing this report, my team at work began exploring an interesting technology for developing front-end user interfaces: the Elm programming language.

JavaScript is the standard for developing interactive user interfaces on the web because it is the language that every modern browser understands. However, JavaScript has its limitations. Like PHP on the server side, JavaScript is a dynamically-typed language and has the same problems in terms of maintainability and lack of tooling to support correct code.

Elm's approach is to sidestep the problems of JavaScript by offering a language that is heavily influenced by Haskell (in fact, its compiler is writtein in Haskell), that compiles to JavaScript that can run natively in any browser. In addition, it offers powerful libraries for implementing user interfaces using a technique known as *functional reactive programming* (FRP).

In Elm's implementation of FRP, values that may change over time are modeld as *signals*. In particular, mouse clicks, text entries, and so on are viewed as a signal of user interface events. The current application state is modeled with a function that takes the entire past history of the signal of UI events and yields a signal of application state. Finally, the UI is rendered as a function that maps the current application state to the user interface, represented using a virtual implementation of the browser's document object model (DOM).

So, in essence, Elm models a user interfaces as a signal of UI events that derives a signal of application states that maps to a signal of UI states, with event handlers ready to inject new values into the signal of UI events. This is a very clean model and, though it may seem inefficient, Elm manages some very impressive performance metrics. For example, each step in the rendering phase generates a fully updated data structure with the new DOM, but performs a diff between that and the browser's actual DOM to make the actual redraw as efficient as possible.

In my experience with Elm, the most important feature is the strong, static type system. Like in Haskell, I have been able to write safe, expressive code and make often quite invasive refactorings with the confidence afforded by a compiler that is able to catch the errors that many type systems allow at compile time but that are fatally erroneous at run time.

Moreover, Elm represents an interesting development in the human side of programming language design. Haskell's benefits often come at the cost of an obtuse programmer experience with cryptic error messages and a preference for abstraction ad absurdum over practical clarity. While Elm is still somewhat immature, its developers have put a strong emphasis on accessibility. Error messages are incredibly helpful and libraries are designed with practical usability in mind as much as leveraging the deep abstractions available to pure, functional programming.

## Conclusion

Despite difficulties using a relatively unfamiliar technology, I found the experience of using Haskell to be an extremely positive one. In my professional experience as a software engineer, the vast majority of my time has been on tasks that can best be described as software maintenance. Even adding new features to an existing piece of software is in many ways a maintenance task, since that new code has to coexist with pre-existing code.

In that light, dynamic languages like PHP, Python, or Ruby, that currently dominate server-side web application development, are ill-equipped to facilitate software development that occurs largely in the maintenance phase of the software development life cycle. Being unable to depend on compile-time type checking is a serious problem. In that regard, Haskell is a step up, but no more than a language like Java.

However, Haskell's other benefits are also valuable. The Servant and Persistent libraries demonstrated that its advanced type system could be leveraged to great practical effect in the domain of web API development. In fact, even if it only came down to eliminating the possibility of null pointer exceptions, Haskell would be a huge improvement, but its pure semantics and functional paradigm are powerful tools for writing code that is easy to reason about and refactor, factors that are particularly useful during software maintenance.

While it may be too soon to recommend moving development teams to a language as radically unfamiliar as Haskell, it was quite exciting to see how effective it could be in the sort of programming that makes up the day-to-day work in my corner of the industry. My hope is that languages like Elm help the paradigm of strongly-typed, pure functional programming languages to make in-roads into the web application development world.

## References

1. "Type-level Web APIs with Servant" (http://www.andres-loeh.de/Servant/servant-wgp.pdf)
2. "Haskell 2010 Language Report", Simon Marlow, ed. (https://www.haskell.org/onlinereport/haskell2010/)
3. "Type-Level Literals" (https://downloads.haskell.org/~ghc/7.10.1/docs/html/users\_guide/type-level-literals.html)