Applied Science Private University
Faculty of Information Technology

Graduation Project (1) Report

Heavenly-x

Prepared by:
Muhammad Tabaza 201610106

Supervised by:
Dr. Mohamed Hijawi

Jan 2020

Contents

Abstract

Software consists of data and processes. With different combinations of these two primitives, we can construct all kinds of software systems, from large-scale machine learning pipelines, to simple command line applications. People coming into the field of software engineering might think the industry had already established practical standards for the most common software functions, but they would be disappointed to see that there are hundreds of ways to implement one thing.

The industry is in dire need of better standards and development tools. Much of the difficulty and tedium software engineers face is caused by the difference in abstraction levels between the software requirements, and the tools used to implement them.

Heavenly-x is a Web application development language and framework that provides abstractions which make it incredibly simple to implement common server-side Web application functionality. It takes much of the "plumbing" off of the hands of engineers; an engineer using Heavenly-x should not be concerned with server API design, software availability, or interservice communication.

The long-term goal of Heavenly-x is to establish a platform and a set of tools that enable developers and data scientists of any experience level to build world-scale software with ease. The name "Heavenly-x" is an abbreviation of "Heavenly Experience." There is huge room for improvement in development tools, but the task of creating a comprehensive and joyful development experience is daunting.

Chapter 1: Introduction

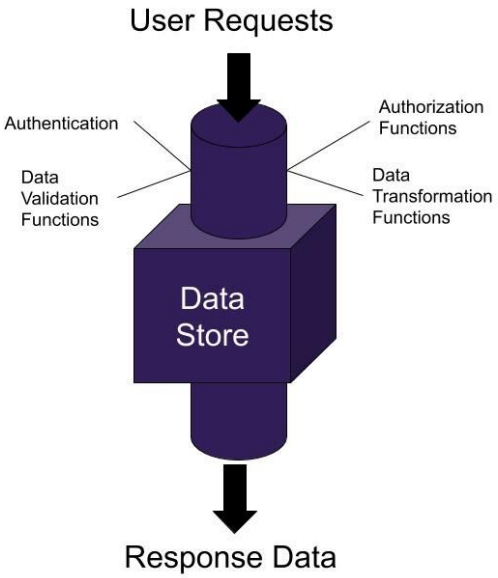
A *Web application* is a software system that can be accessed through the Internet. Web applications are usually comprised of *front-end*, and *back-end* applications. The front-end of a Web application is the part of the system that is exposed to end-users through a Web browser[1]. A server-side application is a set of processes that are executed on one or more remote computers called *servers*, which respond to *client* requests[2].

It is very common that a software application requires a database for persistent storage of data. It is also common that an interface is exposed directly to the clients of the software that allows them to manipulate the stored data in a controlled manner. Most server-side Web applications are simple software layers on top of a database where the following is performed:

1. **User authentication**: The verification of a user's identity.
2. **User authorization**: The verification of a user's application access privileges.
3. **Data validation**: The verification of input data conformity.
4. **Data transformation**: Processing data to produce new data.

Heavenly-x is a language and framework that automates and streamlines the construction of server-side Web applications. It provides a data modeling mechanism, and constructs for combining polyglot user-defined functions that can be used in the data processing pipeline.

[1] "Browser - MDN Web Docs Glossary: Definitions of Web ...." 11 Nov. 2019, https://developer.mozilla.org/en-US/docs/Glossary/Browser. Accessed 16 Jan. 2020.
[2] "Client-Server Overview - Learn web development | MDN." 7 Sep. 2016, https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview. Accessed 16 Jan. 2020.



Benefits of this approach range from automatically distributing user code across networks of servers for scalability and availability, to autogenerating the exposed interface of the application to save huge amounts of engineering time, all while the application remains incredibly easy to comprehend and maintain.

Traditionally, organizations would provision their own data centers on premises, where they host their server-side applications and databases. As *cloud technologies*[3] came into the picture, organizations started renting managed hardware infrastructure, and managing software on their own, either using orchestration tools (e.g. Apache OpenStack,) or completely manually. Such services are known as *IaaS* (Infrastructure as a Service.) Later, cloud service providers started offering *managed virtual machine instances* that dynamically allocate computing resources. However, the problem remains; developers still need to manage the software installed in the

[3] "Cloud computing - Wikipedia." https://en.wikipedia.org/wiki/Cloud_computing. Accessed 16 Jan. 2020.

virtual machines, and for large scale applications, one machine simply isn't enough to handle the load, no matter how powerful.

In 2014[4], Amazon introduced the first public *serverless computing* service (sometimes called Functions as a Service.) The idea behind serverless computing is that a software application should comprise of small, simple, and stateless functions (or *lambdas*.) Such constraints allow cloud service providers to run user functions in a managed environment with little to no input from the user other than the source code. Since the lambda functions are stateless, the service provider can replicate (create multiple instances of) the process running the function, therefore scaling up and down depending on load, with certain guarantees of software correctness and availability. The entire scaling and load balancing process is completely automated, which enables developers to focus more on functional business requirements, and delegate most of the rest.

Heavenly-x is a serverless framework. All user code is completely managed by the runtime, and the user wouldn't need to be concerned with underlying infrastructure (unless the users host Heavenly-x themselves.)

One of the design goals of Heavenly-x is to maintain a low barrier to entry. It only takes knowing one of the very popular programming languages (e.g. JavaScript, Java, or Python,) and understanding the data modeling concepts of Heavenly-x.

When dealing with large amounts of data, it is often useful to structure the data in a way that best encodes the domain requirements, or makes data retrieval and processing more efficient; simple data types, such as numbers and strings of characters, cannot be effectively used to model complex data on their own, instead, they are often used in combination. In many programming languages, there exist constructs such as *classes* and *structs*, which are used to define data shapes, or *schemas*. Classes and structs can have *fields* used to store data attributes, where a field can be of arbitrary data type.

[4] "AWS Lambda - Wikipedia." https://en.wikipedia.org/wiki/AWS_Lambda. Accessed 16 Jan. 2020.

Heavenly-x has a *model* construct for defining data schemas, which are the core of a Heavenly-x application. A very common model is that of a user:

```
@user model User {
   username: String @publicCredential
   password: String @secretCredential
}
```

This model has two fields: username, and password. Both are given the type `String` (plain text.) Both these fields are annotated with *directives*, which add certain behavior to the fields, and therefore modify the behavior of the generated application. The `secretCredential` and `publicCredential` directives are used along with the `user` directive to specify that this model is that of a *user* who's to be authenticated using the specified credentials.

The syntax of Heavenly-x is heavily inspired by GraphQL[5], which should help anyone familiar with GraphQL to get started using the language much more quickly. The generated interface is also a GraphQL API[6], which makes it more flexible and lucrative than a RESTful API[7].

Models can be used as field types instead of primitive types to express relations between models. For example, a `User` can have a list of `Cats`:

```
@user model User {
   username: String @publicCredential
   password: String @secretCredential
   cats: [Cat]
}

model Cat {
   name: String
}
```

[5] "GraphQL spec." https://graphql.github.io/graphql-spec/June2018/. Accessed 16 Jan. 2020.
[6] "Application programming interface ...." https://en.wikipedia.org/wiki/Application_programming_interface. Accessed 17 Jan. 2020.
[7] "Representational state transfer - Wikipedia." https://en.wikipedia.org/wiki/Representational_state_transfer. Accessed 16 Jan. 2020.

Once data modeling is complete, data access rules can be specified in the form of an ACL[8]:

```
import "./authorizers.js" as auth

acl {
   role User {
       allow [UPDATE, DELETE] Cat auth.userOwnsCat
   }
}
```

Import statements are used to read functions from files; in this example, a JavaScript file is imported. The group of functions defined within a file can be given a name (`auth` in this example,) and the functions can be accessed using the dot notation seen in `auth.userOwnsCat`. Imported functions can be passed as arguments to *data access rules*. Access rules can be specified for every defined user model as a way of specifying the *role* of each user of the application. Here, a `role` block is defined for the `User` model, containing one access rule which states:

"Any user can read, update, or delete any `Cat` as long as the cat satisfies the predicate `auth.userOwnsCat`."

The authorization predicate `userOwnsCat` can be defined as:

```
function userOwnsCat(cat, context) {
   return context.user.cats.contains(cat)
}
```

Users of the application can perform authorized *actions*, which include reading, creating, updating, and deleting data, based on their role as defined in the `acl` block.

For model validation, a function can be passed to the model-level `@validate` directive. The function is a predicate of the model type that is annotated. The `@set` directive can be used on the field level to specify data

[8] "Access-control list - Wikipedia." https://en.wikipedia.org/wiki/Access-control_list. Accessed 16 Jan. 2020.

transformation operations that are to be performed on user input data before it is inserted into the database. Similarly, the `@get` directive can be used to transform data going out to the user.

When the application is started, the models defined will be translated to a GraphQL API. Appropriate GraphQL queries, mutations, and subscriptions will be generated and exposed to the client based on model field types, relations, and user access permissions.

Heavenly-x generates and manages the database where the application's data resides. Since Heavenly-x uses Prisma[9] under the hood, it supports many database systems (i.e. PostgreSQL, MySQL, and MongoDB.) This enables direct access to the database for administration purposes, and users of the technology would have a high degree of freedom regarding their choice of the database management system.

Heavenly-x is meant to be a great development experience (hence the name.) Therefore, the back-end application generation is only a small part of the big picture. The long-term goal of the project is to minimize the effort required to realize software, and to make the process of building it truly heavenly.

1.1 Description of the current situation and opportunity

Many Web application development frameworks exist today (e.g. Django, Express.js, and Ruby on Rails.) However, each of these frameworks is coupled to a particular language, and does not accommodate composition of different languages, or the distribution of applications across network node. It's very common that an application reaches a level of adoption where the entire architecture of the application would need to be redesigned to accommodate growth.

[9] "Prisma.io." https://www.prisma.io/. Accessed 16 Jan. 2020.

Web frameworks are built to simplify the process of implementing and integrating common functions (e.g. HTTP routing and request parsing.) However, most features of existing Web frameworks are for low-level networking. Using these features directly introduces tedium and frustration for engineers trying to satisfy a business requirement under time constraints.

Aside from Web frameworks, the serverless services offered by cloud providers (e.g. AWS Lambda,) offer a development experience that leaves a lot to be desired, and lock users into the cloud provider's ecosystem.

The opportunity lies in creating a Web framework with none of the above mentioned drawbacks. Heavenly-x provides high-level constructs that can immensely accelerate the development of Web applications, while maintaining desirable software properties, namely portability, scalability, security, and simplicity.

1.2 Related work

Firebase[10] is a cloud solution by Google. It is integrated with Google Cloud Functions, and FireStore[11], which allows an entire back-end application to be built on top of it without the need for any additional layers of software on top of it. However, unlike Heaven;y-x, Firebase is a closed-source solution that cannot be self-hosted. Firebase also doesn't have native support for GraphQL.

8base[12] is very similar to Heavenly-x in that it generates GraphQL APIs. However, it takes a graphical approach for application design (unlike Heavenly-x, which offers a textual language.) Similarly to Firebase, 8base is also limited in terms of portability.

[10] "Firebase." https://firebase.google.com/. Accessed 16 Jan. 2020.
[11] "Cloud Firestore | Firebase." 3 Oct. 2017, https://firebase.google.com/docs/firestore. Accessed 16 Jan. 2020.
[12] "8base." https://www.8base.com/. Accessed 16 Jan. 2020.

AWS AppSync[13] is a managed solution for creating GraphQL APIs which integrate with AWS services. The key difference between AppSync and Heavenly-x is that AppSync uses GraphQL as the application design language, while Heavenly-x uses its own language for higher flexibility. AppSync is so far the most similar existing solution to Heavenly-x.

1.3 Problem statement (limitation of current systems)

Current development tools and frameworks are not simple or flexible enough. The primary focus of Heavenly-x is to simplify the process of constructing data processing pipelines to the point where any software developer (regardless of their background,) can build world-scale software applications in minutes.

1.4 Problem solution

Heavenly-x offers a simple language for system requirement specification and data modeling. It also does not impose any limitations regarding choice of programming languages that can be used to implement business functions. On the non-functional side, Heavenly-x achieves a high level of software security, reliability, and scalability by default. This achieves the goal of simplifying back-end Web application development to a satisfying extent.

1.5 Project objectives

- Lowering the barrier of entry to software development
- Producing technology that potentially saves thousands of engineering hours
- Empowering businesses and engineers by reducing the effort of realizing their ideas

1.6 Technology and tools used

- The Scala programming language[14]

[13] "AWS AppSync - Amazon Web Services." https://aws.amazon.com/appsync/. Accessed 16 Jan. 2020.
[14] "The Scala Programming Language." https://www.scala-lang.org/. Accessed 16 Jan. 2020.

An open-source multi-paradigm general-purpose programming language that targets the JVM.

- GraalVM[15]

  A JVM implementation that offers language runtime interoperability between multiple languages, and a plethora of features.

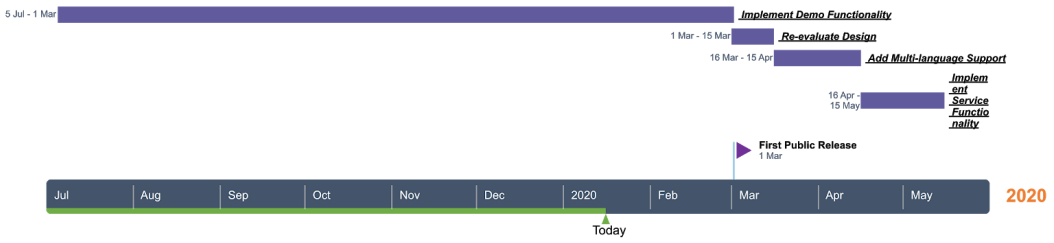
- Prisma

  A tool for wrapping multiple database technologies with GraphQL APIs.

- Apache OpenWhisk[16]

  An open-source serverless framework.

1.7 Project plan for GP2 (Gantt chart, PERT chart)



[15] "GraalVM." https://www.graalvm.org/. Accessed 16 Jan. 2020.
[16] "Apache OpenWhisk - Apache Software." https://openwhisk.apache.org/. Accessed 16 Jan. 2020.

Chapter 2: Requirements and Analysis

2.1 Software Process Model

The team uses SCRUM[17] as the software process. Every week, a sprint planning meeting is held to report progress, and set new objectives. Since the team is made up of two people at the time of this writing, SCRUM made a great fit, because the traditional roles of tester, architect, and others, are split between the two team members. The team uses GitHub issues as the backlog, which keeps discussions and task tracking close to the code.

2.2 System Scope

The first public release of Heavenly-x shall take place before March 1st, 2020. The released system shall have the following functionality:

1. Heavenly-x shall be able to start an HTTP server that exposes the GraphQL API
2. Heavenly-x shall include a data modeling construct
3. Heavenly-x shall include an authorization mechanism (an ACL construct)
4. Heavenly-x shall include a construct for importing JavaScript functions and using them in the data processing pipeline
5. Heavenly-x shall be able to connect to, and manage MongoDB as the primary persistent storage device

Heavenly-x is very ambitious, and all of its potential functionality cannot be implemented in the time specified above. The scope is merely a small subset that can be implemented in time for the graduation project's deadline. Development of this technology will continue long after this scope had been reached.

2.3 List of Functional Requirements & Non-Functional Requirements

Functional Requirements:

1. The Heavenly-x compiler shall validate user code

[17] "What is Scrum? - Scrum.org." https://www.scrum.org/resources/what-is-scrum. Accessed 16 Jan. 2020.

1.1. The compiler shall validate the types of model field default values

1.2. The compiler shall validate references to user-defined types and functions

1.3. The compiler shall validate the existence of imported files and the functions contained within them

2. The compiler shall report code validation errors in a readable format
3. Heavenly-x shall generate a GraphQL API based on user code

   3.1. Heavenly-x shall generate authentication queries for user models

   3.2. Heavenly-x shall generate suitable queries and mutations for each model

   3.3. Heavenly-x shall generate subscriptions for each model for real-time notifications

4. Heavenly-x shall perform database migrations whenever necessary after data model changes
5. Heavenly-x shall support the execution of JavaScript functions as authorizers, validators, transformers, and validators
6. Heavenly-x shall have a “development” mode of operation

   6.1. In development mode, Heavenly-x shall expose a GraphiQL endpoint

   6.2. In development mode, Heavenly-x shall report detailed error responses for unauthorized usage

   6.3. In development mode, Heavenly-x shall report detailed execution traces in the event of an internal error

   6.4. In development mode, Heavenly-x shall provide seed data based on the data model

   6.5. In development mode, Heavenly-x shall listen to any changes to user files and update the running application in real time

7. Heavenly-x shall have a “production” mode of operation

7.1. In production mode, Heavenly-x shall disable all development consols

7.2. In production mode, Heavenly-x shall not return any detailed authorization error messages to clients of the application

7.3. In production mode, Heavenly-x shall disable any execution tracing features

Non-Functional Requirements:

1. The generated application shall be performant

   User requests shall be processed within 10 milliseconds of reaching the server in addition to any latency incurred by the database

2. The generated application shall be scalable

   The application shall utilize any available system threads, CPU cores, or network nodes to scale both vertically and horizontally based on the load

3. The generated application shall be reliable

   The application shall not produce any internal errors that are not related to hardware or fatal network failure

4. The generated application shall be highly available

   The application shall have no down time if no hardware or fatal network failures failures occur

5. Heavenly-x shall provide a good development experience

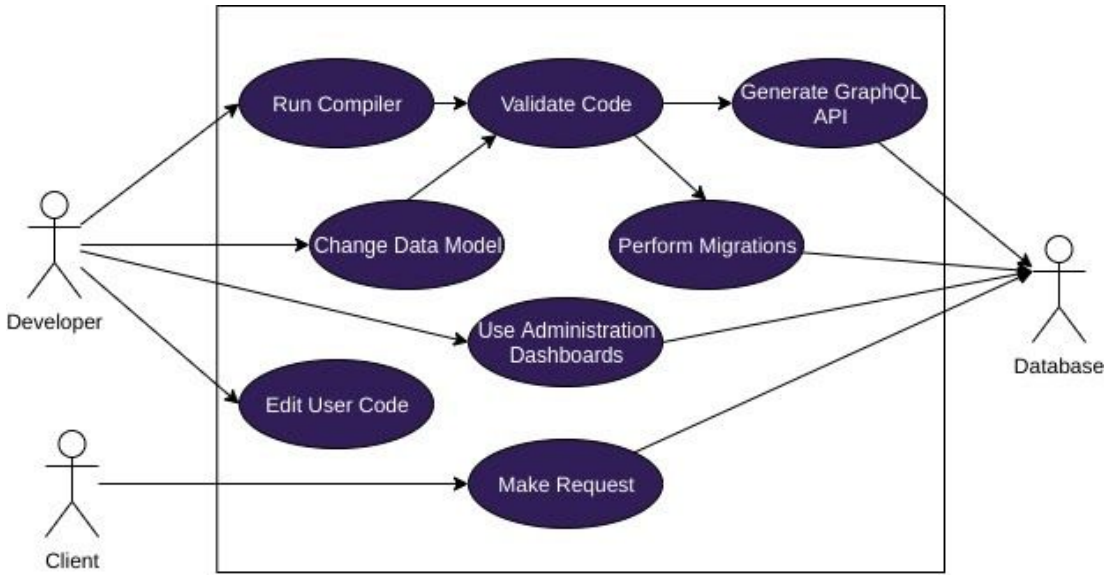
   5.1. In development mode, Heavenly-x shall respond to any user command or file change within 100 milliseconds

   5.2. In development mode, Heavenly-x shall provide simple and descriptive error messages

6. The generated application shall be efficient

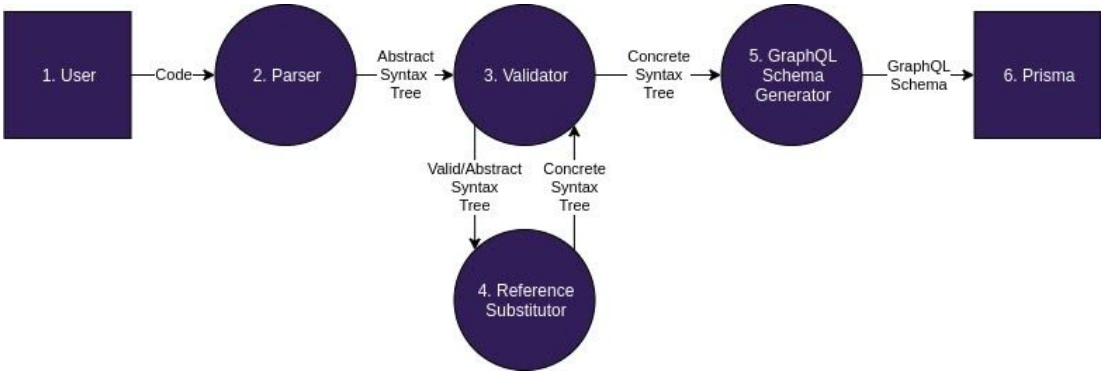
The generated application shall allocate any available resources at its disposal when they are needed, and deallocate them when they are no longer needed

2.4 Use Case Diagram



Chapter 3: Design

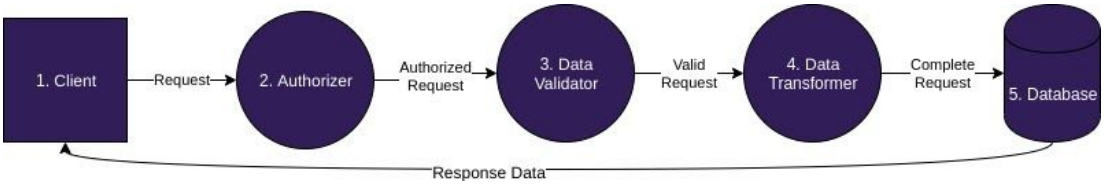
3.1. Compile-Time Data Flow Diagram



1. The compiler user (e.g. the developer, engineer, or data scientist,) feeds Heavenly-x code text to the compiler.
2. The first step of processing user code is parsing[18] it. The parsing process outputs an abstract syntax tree (AST)[19].
3. The AST is validated for any type errors, reference errors, and missing file errors.
4. The AST is processed by substituting imported function references with concrete function values, and adding defaults parameters where the user does not specify a value. The validation and substitution processes are incrementally performed, and they interleave with each other.
5. The GraphQL data types and API schema are generated based on the data model definitions in user code.
6. The GraphQL schema is fed to Prisma to generate a database, and perform migrations if necessary.

[18] "Parsing - Wikipedia." https://en.wikipedia.org/wiki/Parsing. Accessed 17 Jan. 2020.
[19] "Abstract syntax tree - Wikipedia." https://en.wikipedia.org/wiki/Abstract_syntax_tree. Accessed 17 Jan. 2020.

3.2. **Runtime Data Flow Diagram**



1. The application client (enduser) makes requests to the generated API to retrieve or mutate data.
2. The authorizer makes sure that the client is authorized to access the requested data using the rules specified in the ACL block.
3. Any data provided by the client is validated using the functions specified in `@validate` directives.
4. Data transformations are performed using functions specified within `@set` and `@get` directives.
5. A final query is constructed and made to the database and the response data is sent back to the client.