Managed Game Server Hosting

by

Walker Christie

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science

by

Walker Christie

September 2019

APPROVED:

Professor Jonathan Weinstock

Professor Therese Mary Smith

Abstract

Multiplayer video games have permeated our everyday lives, allowing people on other sides of the world to play with each other. While some video games offer dedicated server hosting as a feature, others require players to host their own video game servers.

This project describes the process of creating the basis for a managed video game hosting service. Using primarily AWS resources, the following was created: a backend allowing for the automated deployment of video game servers, group-based database interaction, a Python command line utility for deploying and destroying stacks, and a website featuring main and admin pages.

Acknowledgements

I want to thank Professor Smith and Professor Weinstock for guiding me through this project and for agreeing to be my advisors at the last moment.

I also want to thank my parents for their endless love and support and for giving me the opportunity to attend WPI.

Contents

1 Introduction 5

2 Project Scope 7

3 Prototyping 8

4 Requirements Analysis 9

4.1 Risk Assessment 9

4.2 PERT Chart 10

5 Development 11

5.1 Website 12

5.2 Database 14

5.3 Authentication 16

5.4 Distributed System 18

5.5 Video Game Containers 21

5.6 Control Script 25

6 Results 26

7 Appendix 29

8 List of Figures 36

9 References 38

1 Introduction

From their culmination in the early 1970s, video games have grown from simple 2D games to complex, highly-interactive experiences that permeate our culture and everyday lives. One of the most pervasive aspects of gaming has been that of online games. Instead of playing alone, players can now share a gameplay experience with friends and strangers in an interactive online environment.

Online games allow players to remotely interact and play with each other through a client/server architecture. In this architecture a player, or client, connects to a remote host that acts as the game server. This server accepts the client's connection and adds it to a pool of currently connected players. The server then coordinates sending information about the client such as world position, rotation, interactions, played sounds, animations, etc. to all other clients connected to the server. By having a server distribute a single player's actions to all other players, and vice versa, each player can participate in the same online game world.

The problem with online games is that game developers have two options for creating a multiplayer service; either pay for server hosting or allow players to set up and host their own servers. The first option requires game companies or indie studios to host dedicated game servers for their player base. This is convenient for players because they don't have to think about the intricacies of hosting a game server, all they need to do is join the server from their game client. The problem with this option, however, is the cost. Hosting game servers costs money, and if the game's player base is large, the cost of hosting servers can become too much for a game company to pay. The alternative to this method is to allow players to host their own game servers.

Player-hosted game servers are a good alternative for game companies that don't want to pay for the cost of hosting servers themselves. In order for a player to host their own game server, an executable separate from the game client must be downloaded. This executable, when run, creates a game server that other players (clients) can connect to. The problem with player-hosted servers is the amount of computational resources and technical knowledge required to set one up. Game servers need dedicated RAM and CPU resources as well as a fast network connection in order to coordinate connected players. As for the required technical knowledge, hosting a server is not as simple as launching the game client itself. Hosting can involve port forwarding routers, editing multiple configuration files, changing file permissions, setting rules in a firewall, and much more. Despite the challenges that come with self-hosting game servers, there are plenty of games where that is the only multiplayer option, one of them being Minecraft (Mojang, 2011).

Minecraft is a sandbox video game where players explore, build, and gather resources in a procedurally generated world. The game is extremely popular, being the second highest selling game of all time with over 176 million copies sold. As of 2018, Minecraft has over 91 million active players and has continued to grow in popularity every year since its culmination. In order to host a Minecraft server, a user has to download the server jar file, execute it via the command line, and specify how much RAM to allocate to the server. A user must also port forward their router so outside connections can enter, as well as allow Minecraft through their firewall. In addition to the setup process, running a Minecraft server requires a lot of dedicated RAM, a consistently high network bandwidth, low network latency, and a computer that can constantly stay turned on. Because of the number of prerequisites that come with hosting a Minecraft server, many players join servers hosted by other people, rather than hosting their own.

For my Major Qualifying Project, I'd like to create a paid game server hosting service for players who would like to host their own game servers but do not have the time, resources, or knowledge to do so. Players will login, choose the game they want hosted, and select the amount of computational resources to allocate to the server. The hosting choices will be a list of games each with their own automated startup and configuration logic. The price of hosting is determined by the length of hosting time, number of resources, and choice of game.

2 Project Scope

The scope of this project can be divided into the analysis, design, development, and testing of 3 individual systems. These systems loosely follow the model-view-controller (MVC) design pattern.

The first system is a distributed system used for serving an arbitrary amount of game servers to the registered users. Using AWS (Amazon Web Services) EC2 instances and Docker, multiple game servers can be deployed and run simultaneously in their own isolated environments. This allows for a set amount of game servers to run on a single EC2 instance, each with a set RAM allocation and max CPU utilization.

The next system is the web client (view) which presents an interactive way for users to purchase, configure, deploy, and monitor their own game servers. The main goal of the website is to make purchasing and deploying a game server as easy as possible for beginners while simultaneously providing extensive configuration options for more technologically advanced users. The website will be written in HTML/CSS and paired with a frontend JS library for rendering data from the backend.

Finally, the controller system is used for interfacing between the AWS VPS instances and the frontend website. Commands will be processed through a set of REST endpoints, ensuring a clean separation between the front end and the back end. It also ensures loose coupling between the view and the backend, making command addition/subtraction easy.

3 Prototyping

Prototypes are useful for demonstrating that certain features of the project are possible to create before building other dependent pieces. Prototyping is an important step in the Software Development Life Cycle (SDLC) as it can save time in the future. In the event that a feature isn't possible to create, the SDLC can be adjusted to remove or replace the infeasible feature before extensive development is done.

In order to demonstrate that the automated creation of a game server is possible, a prototype of the deployment system was created. The purpose of this prototype is to demonstrate the automated setup and deployment of a containerized video game server on an arbitrary Virtual Private Server (VPS). The chosen game was Minecraft with Amazon Web Services (AWS) as a VPS. Docker was chosen as a containerization software, allowing for a set amount of game servers to live on the same VPS without interacting with each other. The result of this prototype was a playable Minecraft game server hosted on an AWS EC2 VPS.

4 Requirements Analysis

4.1 Risk Assessment

Because this project requires paying for AWS EC2 instances, database hosting, web hosting, and a domain name, there is an inherit risk of losing money. In order to minimize this risk, most of the development will be done locally rather than on a paid hosting service. Development and testing of the distributed system portion will be done on an AWS EC2 "micro instance". AWS micro instances are virtual private servers with low computational resources that are free for one year. While these servers aren't very powerful, they are strong enough to deploy a few small game servers on a single instance for testing purposes.

Once development is completed, all services will be relocated to remote hosts. At that stage, entering credit card information will be required to pay for the hosting of the various created services. While the expectation is that paying users will offset the cost of hosting, the risk of losing money is still there. Regardless of how the financial aspects of this project turn out--- I, Walker Christie, take full responsibility for all project related financial outcomes.

4.2 PERT Chart

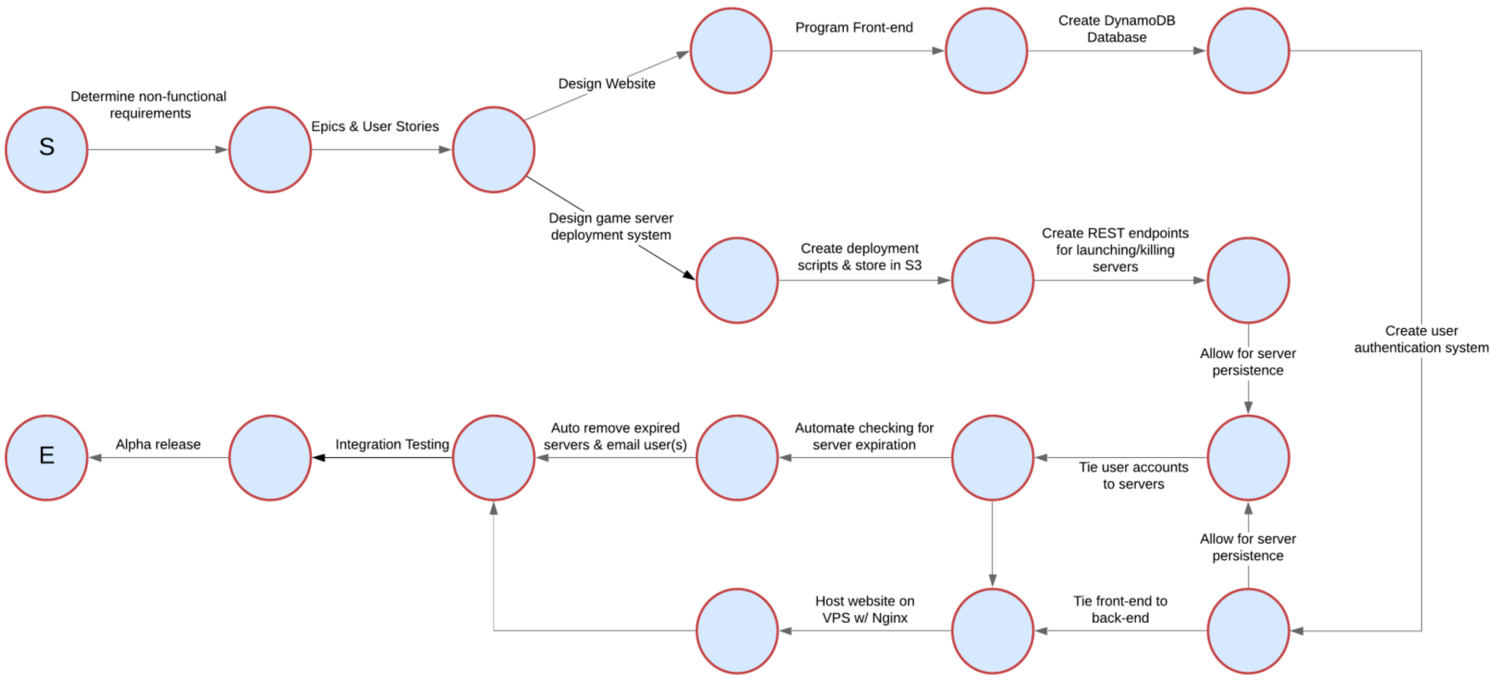

S
Determine non-functional requirements
Epics & User Stories
Design Website
Program Front-end
Create DynamoDB Database
Design game server deployment system
Create deployment scripts & store in S3
Create REST endpoints for launching/killing servers
Create user authentication system
Allow for server persistence
Tie user accounts to servers
Allow for server persistence
Tie front-end to back-end
Host website on VPS w/ Nginx
Automate checking for server expiration
Auto remove expired servers & email user(s)
Integration Testing
Alpha release
E

Figure 1: PERT Chart

Because this is a large project with multiple systems and moving parts, it was important to devise a method of tracking progress. To do this, a PERT chart was created at the beginning of the project, providing a clear timeline for the development process. The PERT Chart, as seen in Figure 1, flows from the starting node (S) to the ending node (E). Each node in the chart represents a project requirement and the arrows represent dependencies. This choice of chart is useful because it displays what tasks are dependent on one another as well as which tasks can be done simultaneously. It also helps track progress as the project is developed over time. While this PERT chart wasn't entirely followed during the development of this project (as seen in the results section) it provides an easy method of prioritizing work over the lifetime of the project.

5 Development

In order to develop a managed video game server hosting service in a timely manner, the project timeline is divided into three objectives. These objectives loosely correspond with the Model View Controller pattern mentioned in chapter 2.

5.1 Website

The first objective is to create the website (or view in the MVC model) that is presented to the client. Because the website is the first thing a prospective client sees, it is imperative that the product being offered is simple to understand and appears attainable. In other words, the website should present a clear solution to the problems that come with hosting one's own game server.

The technical stack for the website extends beyond the standard HTML/CSS markup. In order to increase extensibility and allow for reuse of components, I decided to use the React Javascript library. React is a front-end web framework used for building user interfaces. React code is organized by *components*, which are modular elements of a website. An example component may be a navigation bar that gets displayed at the top of every page. In traditional HTML/CSS, the programmer would have to manually place the code for the navigation bar on every page of the website. With React, however, one can create a navigation bar *component* a single time and include it across each page, removing duplicate code.

React also has a concept of component *state*. A component's state determines how a component renders and behaves. In other words, state makes components dynamic, allowing them to adjust to user input, network responses, timers, any other type of event. One example of a component using state would be a clock. A clock is a component that renders its text every second, displaying the current time to the user.

With React, this component could have a state object containing the current time where the state is updated every second.

In order to decrease development time, Typescript was added to the project. Typescript is the same language as Javascript except that it contains strong typing. A strongly typed language is one in which variables are bound to a specific data types and will result in errors if the types don't match up as expected. This is advantageous because it allows the compiler to catch syntax errors before the program is run, reducing unexpected errors down the line.

The organization of the website itself is divided into two main sections: the main site and admin pages. The main sites' pages are meant for unauthenticated users to learn more about the service and what features are available. The admin pages allow authenticated users to monitor existing servers, monitor resource usage, view invoices, and any other task related to the management of a purchased server. On the main site the following pages are available: home, games, pricing, and support. The home page displays a gallery of available games, some servers and their prices, hosting features, and a step-by-step graphic that illustrates how easy it is to launch a game server. The games page lists all available game servers and their lowest prices. The pricing page displays all available game servers and their various pricing plans. Finally, the support page contains a contact form, allowing visitors and customers to ask questions about the hosting service.

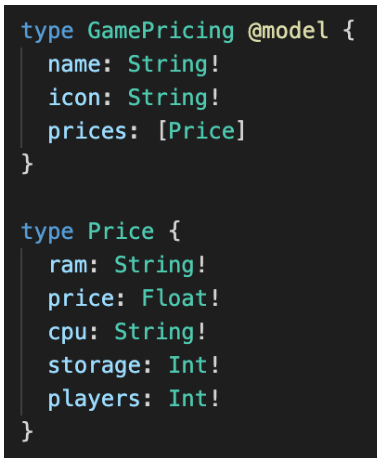
The admin panel contains the following pages: invoices, account, server list, and manage server. The invoices page displays which servers the user has purchased. The account page lists user information such as: their email address, name, and user ID. The server list page contains all servers currently registered to the user. Each server can be turned on or off from this page. Finally, the manage server page contains various options for controlling and visualizing a video game server. Some options include: viewing server resource usage, viewing console activity and sending commands, FTP account management, and configuration file editing. Figures for all web pages on the website can be viewed in the appendix of this document.

5.2 Database

While there are plenty of database options out there, it was important to select one that fit with the website's use cases. Having a database that can be distributed across many servers at once is important, as there may be users on opposite sides of the world accessing the database at the same time. Next, the database schema should be easily adjustable. Since the database is so tightly coupled to many systems, it’s likely that the schema will have to change as more systems are implemented and connected together. In order to satisfy these two constraints, a NoSQL database was chosen.

NoSQL databases are unlike traditional SQL databases because they do not work through tabular relations. There are many different choices for a NoSQL database, but for this project AWS' DynamoDB was chosen. DynamoDB is a *document store* database, where data are organized into documents and collections. In this database, documents (similar to rows in a SQL DB) are stored within collections (similar to tables). The big difference between a document store database and a SQL database is that every row in a SQL database has the same sequence of fields (or columns). In a document store database, it is possible to store an arbitrary number of fields within a single document.

In order to interface the database with the website, AWS AppSync with GraphQL is used. AWS AppSync is a development platform used for building GraphQL-driven databases. It provides features for interfacing between GraphQL and DynamoDB, as well as restricting table access to certain groups of users (discussed in depth in chapter Authentication5.3). GraphQL is used for interfacing with all kinds of AWS database solutions, including DynamoDB. GraphQL works by defining a database schema that is then used to automatically configure DynamoDB on AWS' servers. An example query for the pricing schema is shown in Figure 2.

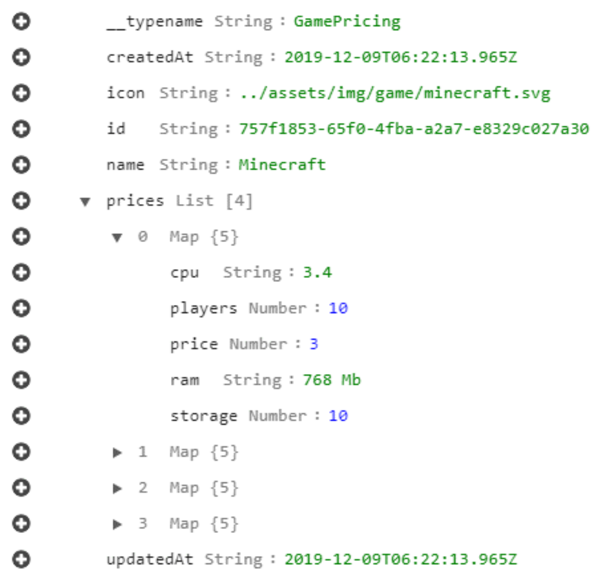


```
type GamePricing @model {
  name: String!
  icon: String!
  prices: [Price]
}

type Price {
  ram: String!
  price: Float!
  cpu: String!
  storage: Int!
  players: Int!
}
```

Figure 2: GraphQL pricing schema

In the pricing schema there are two defined types, *GamePricing* and *Price*. GamePricing is denoted with *@model* meaning the type acts as its own collection in the database, similar to a table. The Price type is not its own collection. Rather, it serves as the type for the *prices* array within the GamePricing collection. This schema, when uploaded to AWS through AppSync, is provisioned and created in DynamoDB as a collection. An example item in the GamePricing collection is seen in Figure 3. Any attempts to modify the collection with data that does not fit the defined schema are rejected by GraphQL. This is useful because it guarantees data consistency by rejecting malformed insert operations.



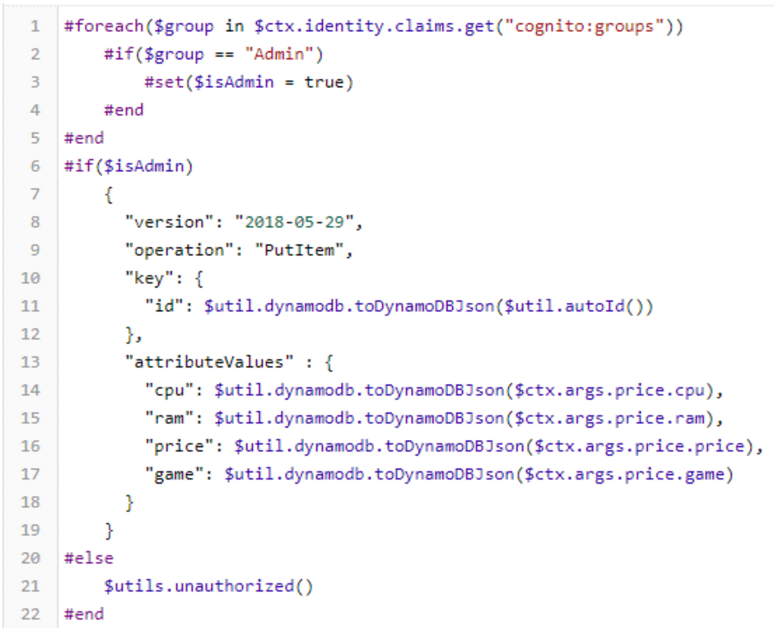
```
    __typename String : GamePricing
    createdAt String : 2019-12-09T06:22:13.965Z
    icon  String : ../assets/img/game/minecraft.svg
    id    String : 757f1853-65f0-4fba-a2a7-e8329c027a30
    name  String : Minecraft
  ▼ prices List [4]
    ▼ 0  Map {5}
           cpu   String : 3.4
           players Number : 10
           price Number : 3
           ram   String : 768 Mb
           storage Number : 10
    ▶ 1  Map {5}
    ▶ 2  Map {5}
    ▶ 3  Map {5}
    updatedAt String : 2019-12-09T06:22:13.965Z
```

Figure 3: GraphQL schema provisioned on DynamoDB

5.3 Authentication

Authentication is done through AWS Cognito, a managed user authentication service that handles sign in, sign up, password recovery, and two factor authentication. Cognito allows for users to be sorted into groups which define what resources they are allowed to interact with. There are three groups defined for this project:

- Guest – Users in this group have the most limited access. Guests may only read public-facing database tables, such as the prices table. Every unauthenticated client that visits the site is automatically assumed to be in this group.



```
#foreach($group in $ctx.identity.claims.get("cognito:groups"))
    #if($group == "Admin")
        #set($isAdmin = true)
    #end
#end
#if($isAdmin)
    {
      "version": "2018-05-29",
      "operation": "PutItem",
      "key": {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId())
      },
      "attributeValues" : {
        "cpu": $util.dynamodb.toDynamoDBJson($ctx.args.price.cpu),
        "ram": $util.dynamodb.toDynamoDBJson($ctx.args.price.ram),
        "price": $util.dynamodb.toDynamoDBJson($ctx.args.price.price),
        "game": $util.dynamodb.toDynamoDBJson($ctx.args.price.game)
      }
    }
#else
    $utils.unauthorized()
#end
```

Figure 4 AppSync Resolver for Prices PutItem operation

- Member – Users are automatically assigned to this group after successfully signing up and confirming their email address. Members are able to get data from database tables pertaining to their own account. For example, members may read rows from the *servers* table that have a user ID that matches their own.
- Admin – Users in this group have the most access to resources. Admins may read and write data to every table in the database.

Group permissions are enforced using AWS AppSync resolvers, small programs that tell AppSync how to translate incoming GraphQL requests and transform response data. Figure 4 shows one such resolver for the prices table. This resolver is used for saving items in the prices table, an operation that should be done only by administrators. By using AppSync resolvers, unauthorized requests (i.e. requests from Guests or Members) are rejected by returning a 500 status code and an unauthorized error message.

Authorized Admins, however, are able to insert items into the database simply by passing their session ID. The following table outlines each operation and their associated group requirements:

Table	Operation	Allow Groups
Servers	Put Item	Admin
Servers	Get Item	Member (own ID), Admin
Servers	Get All Items	Admin
Servers	Delete Item	Admin
Prices	Put Item	Admin
Prices	Get Item	Guest, Member, Admin
Prices	Get All Items	Guest, Member, Admin
Prices	Delete Item	Admin

Figure 5 Database Group Permissions Table

5.4 Distributed System

The distributed system portion of the application is comprised of multiple AWS resources contained within a single "stack". A stack in AWS is a collection of resources (i.e. AWS Lambda, DynamoDB, IAM, EC2, etc.) that are created or destroyed simultaneously. Stacks are modeled and provisioned through AWS CloudFormation. CloudFormation is a service that allows developers to create a single template file that models all the resources that make up a stack. CloudFormation then reads these template files (written in JSON or YAML), creates the requested resources, and treats the deployment as a group (or stack) of resources. This removes the process of manually creating AWS resources, configuring them to communicate with each other, and deleting the provisioned resources when they're no longer needed. Using stacks also makes it easy to develop, test, and deploy instances of the application within isolated environments. For example, staging, testing, and production environments could be created, each one being a fully-functional instance of the application. This also means that every developer working on the application could have her own development environment, all while the live production environment remains untouched.

A visual representation of this project's template/stack can be seen in Figure 6. Each rounded white box represents a single resource while the arrows between them represents a relationship between resources. Resources are also labeled with a logical identifier to uniquely identify a resource (i.e. LHSITE). There are six resources on the left that represent database tables (*LHDBREGISTRY*, *LHDBPRICES*, and *LHDBSERVERS*), the website S3 bucket (*LHSITE*), container registry (*LHECR*), and game servers cluster (*LHECS*). The other resources to the right are used to enable user authentication and interfacing with the database.

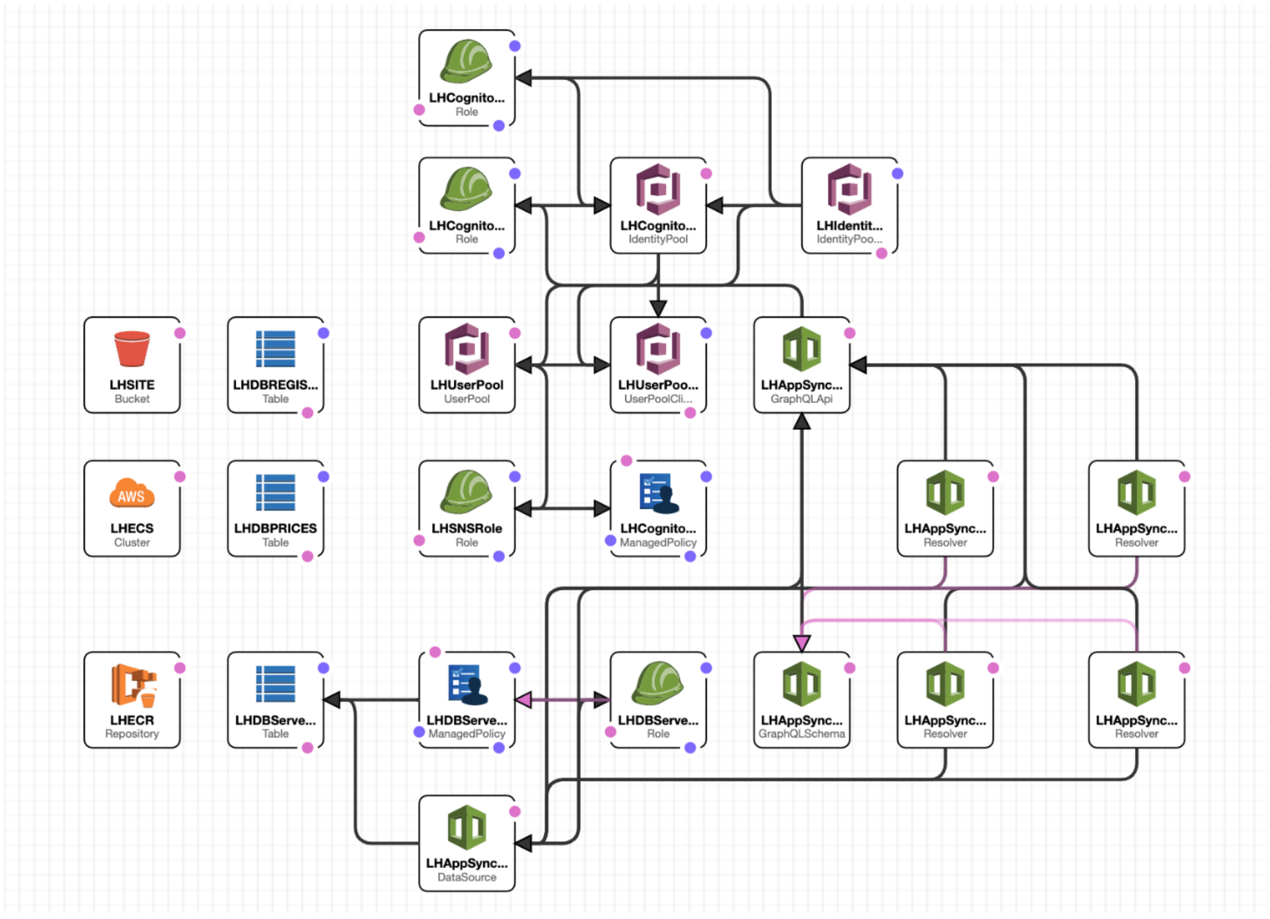


Figure 6: Project CloudFormation Template

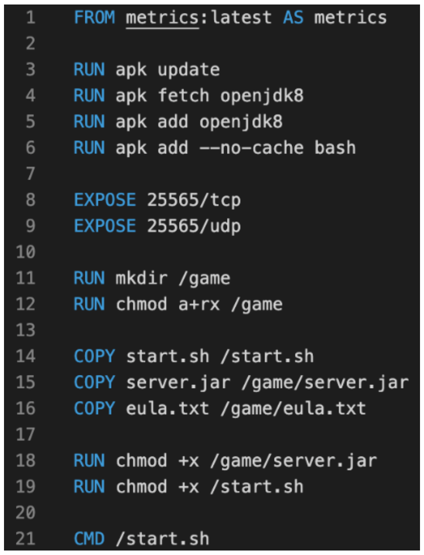
The *LHSITE* resource is an S3 bucket that is used for hosting the website. When creating a stack using this template, all website files are copied to *LHSITE* and web hosting is enabled on the bucket. *LHECR* is an AWS *Elastic Container Registry* (ECR) resource. ECR, like its name implies, is an AWS-managed docker container registry and is used to store video game container images. *LHECS* is an AWS *Elastic Container Service* (ECS) resource that creates a cluster where video game container images are deployed. ECS works by pulling container images from ECR and starting them in the specified region. This means video game servers can be deployed across the world, allowing the customer to choose which region is closest to themselves, minimizing network latency. The functionality of *LHECR* and *LHECS* is explained in greater depth in section Video Game Containers5.5. *LHDBREGISTRY*, *LHDBPRICES*, and *LHDBSERVERS* are each AWS DynamoDB tables. *LHDBREGISTRY* stores information about container images uploaded to *LHECR. LHDBPRICES* stores pricing information and is read every time a user opens the pricing page on the website. *LHDBSERVERS* stores data about provisioned servers. This includes the container image that was launched, amount of computational resources requested, region the server is running in, and the ID of the user that launched the server.

5.5 Video Game Containers

Video game servers across different games often have unique configuration and resource requirements. The game Minecraft, for example, is very RAM heavy while the game Counter Strike is more CPU intensive. Similarly, the steps taken to setup a game server differ widely between the two games. In order to automate the setup process for individual video game servers, each game was containerized using Docker.

Docker (Docker (software), n.d.) is a set of Platform as a Service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries, and configuration files. There are multiple advantages to using Docker containers for server hosting. Because containers are isolated, running game servers cannot interact and accidentally break each other. Docker also makes it easy to test, build, and run containers very quickly. When building a container image, Docker reads a *Dockerfile*— a text document that describes how a container is built and runs. When a container image is run, Docker automatically creates a small virtual machine and runs the commands specified in the *Dockerfile*. This means configuring how the container functions can be automated and quickly extended.

Each video game version receives its own unique *Dockerfile* that describes server dependencies, how the server is configured, and what script starts the server. Figure 7 is an example of the Minecraft version 1.15.1 *Dockerfile*. Line 1 references the Docker base image, metrics, which is described in the next paragraph. Lines 3-6 pull the dependencies needed to run a Minecraft server. In this case, Java and bash are installed. Lines 8-9 open the necessary port, 25565, for both UDP and TCP. Lines 11-19 copy the required files to the container and apply executable permissions to them. Finally, line 21 tells Docker which script to run (in this case start.sh) when the container image is started.



```
1   FROM metrics:latest AS metrics
2
3   RUN apk update
4   RUN apk fetch openjdk8
5   RUN apk add openjdk8
6   RUN apk add --no-cache bash
7
8   EXPOSE 25565/tcp
9   EXPOSE 25565/udp
10
11  RUN mkdir /game
12  RUN chmod a+rx /game
13
14  COPY start.sh /start.sh
15  COPY server.jar /game/server.jar
16  COPY eula.txt /game/eula.txt
17
18  RUN chmod +x /game/server.jar
19  RUN chmod +x /start.sh
20
21  CMD /start.sh
```

Figure 7: Minecraft 1.15.1 Dockerfile

One important feature in server hosting is the ability to monitor the resource usage of the server. If a server is frequently crashing or lagging it may be simply running out of RAM or CPU bandwidth. In order to track these resources (or metrics) and display them to the user, two scripts were created. The first script, metrics.sh, is launched when the container starts and is used to continuously write resource usage to a file every 30 seconds. This is accomplished by referencing the Linux cgroup assigned to the container and reading the pseudo-files for memory and CPU usage. The second script, metrics.py, exposes a set of HTTP endpoints that can be contacted to retrieve this resource usage file. The website can then contact the endpoint at the server IP address and retrieve up-to-date information about the server's resource usage. Each incoming metrics request must also pass a session ID which is checked against the user database, ensuring that only the server owner has access to her own metrics.

Docker containers are packaged and distributed in container *images*– lightweight, standalone executables that contain everything needed to run an application's code. These images are generated locally using the *docker build* command. The images are then stored in an Elastic Container Registry (ECR) and are deployed when needed to the Elastic Container Service (ECS) cluster. Servers are created and destroyed by contacting REST endpoints created in AWS Lambda.

AWS Lambda is a service that allows developers to run code without having to provision or manage servers. The biggest advantage to this is cost. With Lambda, you only pay for the compute time used while the program is running, unlike traditional REST endpoints which run on a 24/7-uptime dedicated server. Lambda is used in this project to deploy game servers stored in ECR to an ECS cluster located at a particular region. The function that does this is called *lh-add-server*, which exposes a REST endpoint with a set of input arguments that deploy the requested game server. Arguments include:

- Stack: Name of the deployed stack (i.e. production).
- Family: Combination of the game name and version.
- Image: Container image URI stored in the Elastic Container Registry.
- CPU: Amount of CPU resources to allot to the server.
- RAM: Amount of RAM to allot to the server.
- Region: AWS region to deploy the server.
- PortMappings: A dictionary containing which ports to open across either TCP or UDP.

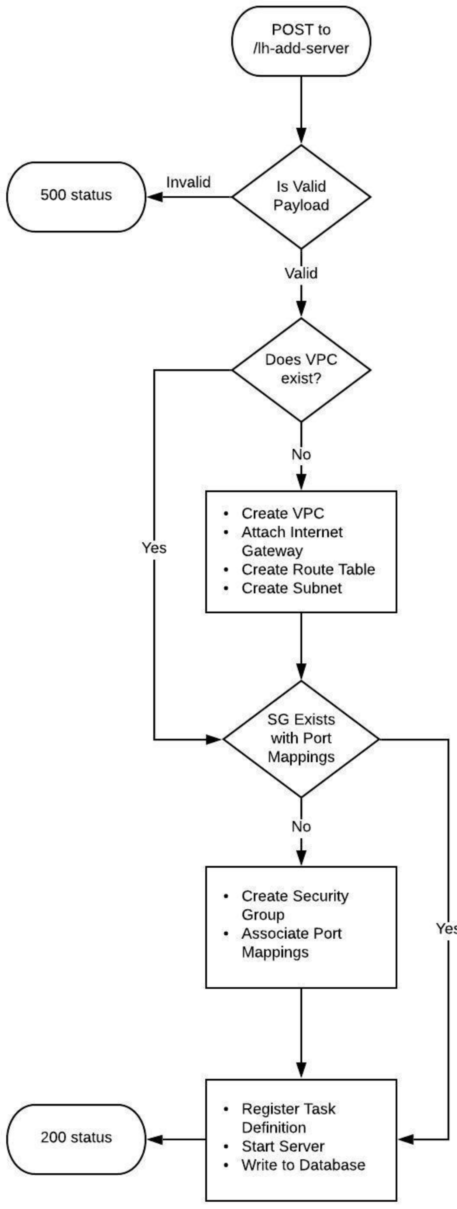


Figure 8: Add Server Flowchart

Figure 8 shows a flowchart of deploying a video game server to an ECS cluster. In order to avoid duplicate resources, both the VPC and Security Groups for the passed configuration are checked to see if they already exist. Security Groups are only checked against the passed port mappings and created if the requested ports do not exist in any security group. This means that server connections are limited to the minimum required opened ports. For example, a game server hosting Minecraft, which requires port 25565 to be open, cannot be connected to via Counter Strike's open port, 6003.

5.6 Control Script

While CloudFormation helps get the bulk of the distributed system deployed, it cannot deploy everything. Uploading the website files and container images, for example, must be done after the stack has been deployed. Deleting deployed stacks also requires some more work other than simply pressing delete on the AWS console. Running ECS tasks, for example, must be stopped before a stack can be deleted. The same goes for the ECR, which must be cleared of any container images before being deleted. In order to automate these tasks, a Python command line utility was created.

Control.py is a command line utility used for automating the creation, deletion, and updating of stacks. It features *help* information for every command, as well as detailed error messages should any task fail. The script works on both Windows and Mac OSX systems and features the following commands:

- deploy – Issue stack deployment commands.
    - template [-h] path name – Deploy a stack via a template file specified at the passed path.
    - site [-h] [--upload-prod] [--upload-dev] [--upload-bundle] [--build-prod] [--build-dev] name – Upload website to the passed stack's S3 bucket. Optionally run a production or development build before uploading and upload either the development build or production build.
    - registry [-h] [--game GAME] [--version VERSION] name – Upload a game version to the ECR within the passed stack. If version isn't specified, all versions for the passed game are uploaded. If game and version aren't specified, all games and their versions are uploaded to the ECR.
- delete [-h] name – Delete the stack with the passed name.
- list [-h] [--all] [--name NAME] – Either list information about all deployed stacks or list information about the stack with the passed name.
- seed [-h] table file name – Seed database tables with content from a local file. This command takes: the name of the table to seed (i.e. prices), a path to a JSON file to seed the table with, and the name of the stack containing the table.

6 Results

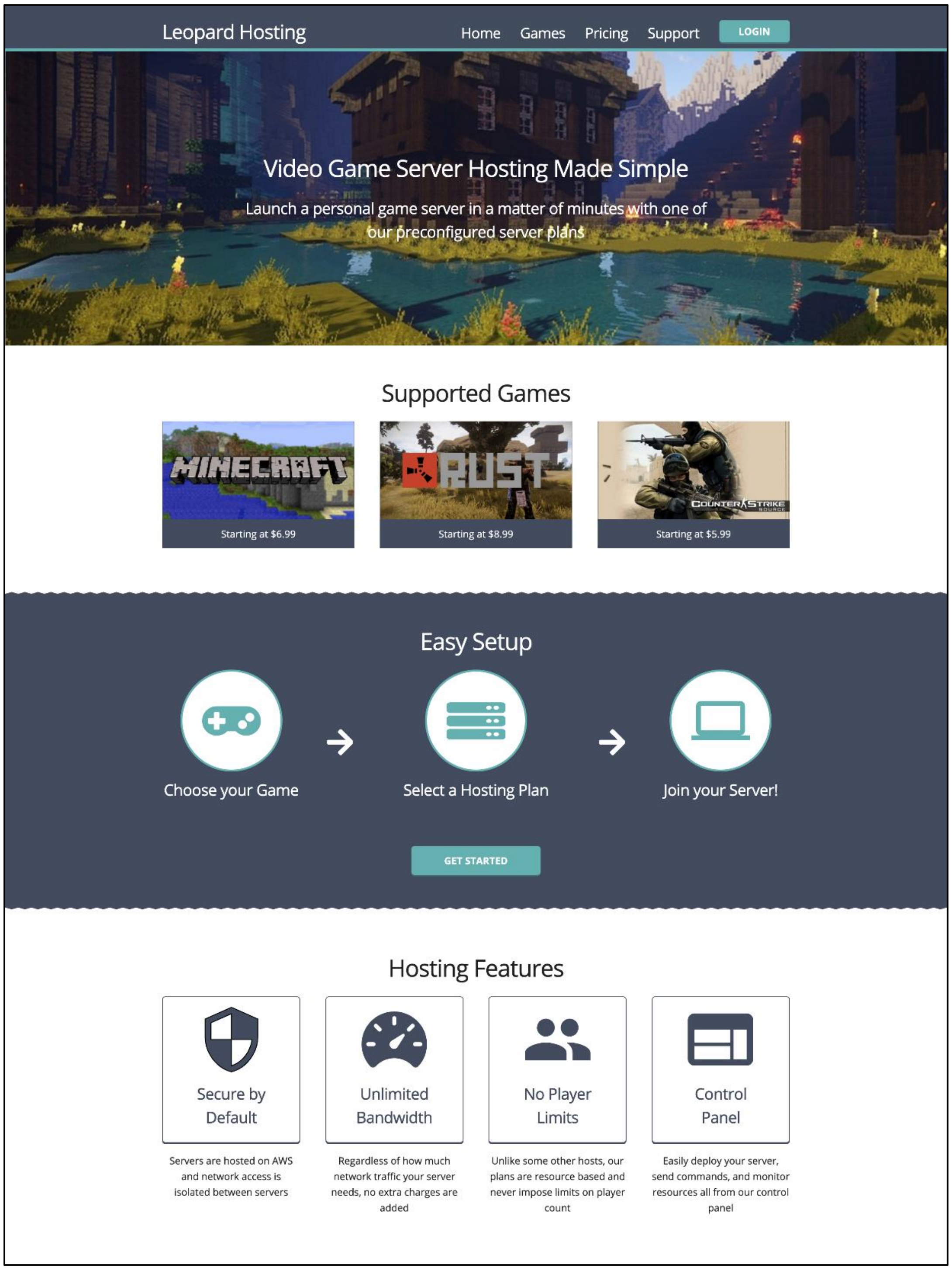
At the beginning of this project, I wanted to ensure that every step taken followed the SDLC guidelines as close as possible. It was important to me that the project was carefully planned and created in a way that ensured loose coupling between systems. By following these ideas, I was able to create an application that can be actively extended by multiple developers simultaneously in an Agile environment.

The culmination of this project resulted in a website and backend able to launch and configure a single type of video game server. Developers are able to deploy, delete, and update stacks through a command line utility allowing for quick development iterations. Through this system of multiple deployment stacks, developers can easily work on this project without having to worry about breaking other developers' deployments or the production deployment.

Other features, such as user authentication and table operation permissions were implemented as well. Users are currently able to sign in, sign up, and recover their account through password reset emails. Database table operations are also limited by the group a user is in, securing the database through the principle of least privilege.

While a lot was done over the three terms, there are still many features that need to be implemented. As of now, users have no way of paying for a game server, website staff cannot update prices without manually changing table rows, there is no way to send/receive support tickets, users can only deploy one game (Minecraft), and much more. The purpose of this project, however, was not to create this entire hosting service over three terms. Instead, it was to use the principles of good software engineering to construct the basis for a video game hosting service website. The result of this project is a backend and frontend that can be quickly extended by multiple developers. While these are good results for an MQP, the plan is to continue development of this project and release a beta version in a few months.

7 Appendix



Video Game Server Hosting Made Simple

Launch a personal game server in a matter of minutes with one of our preconfigured server plans

Supported Games

Starting at $6.99

Starting at $8.99

Starting at $5.99

Easy Setup

Choose your Game

Select a Hosting Plan

Join your Server!

Hosting Features

Secure by Default

Servers are hosted on AWS and network access is isolated between servers

Unlimited Bandwidth

Regardless of how much network traffic your server needs, no extra charges are added

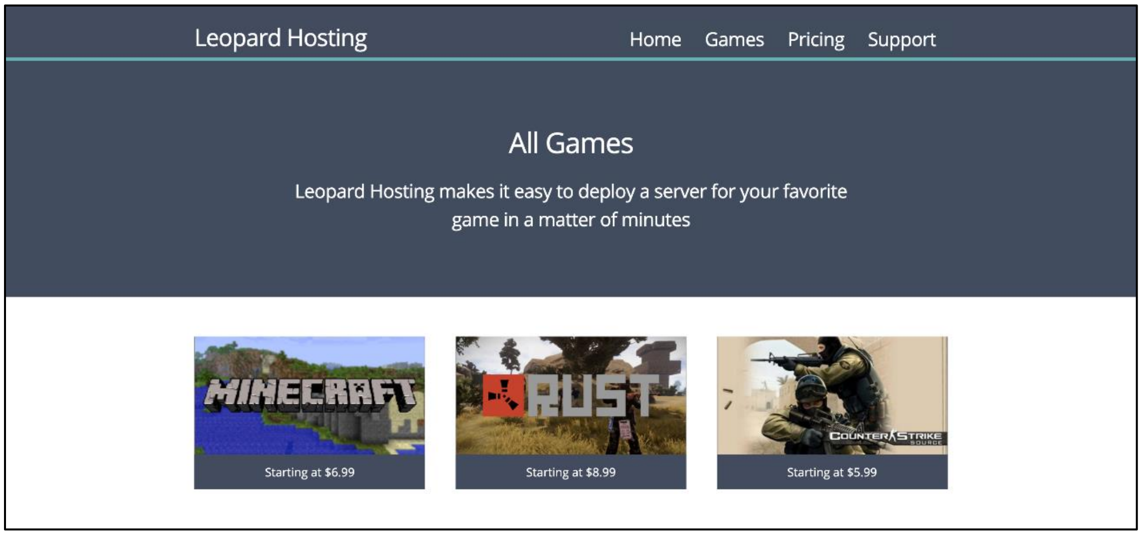
No Player Limits

Unlike some other hosts, our plans are resource based and never impose limits on player count

Control Panel

Easily deploy your server, send commands, and monitor resources all from our control panel

Figure 9: Home page



All Games

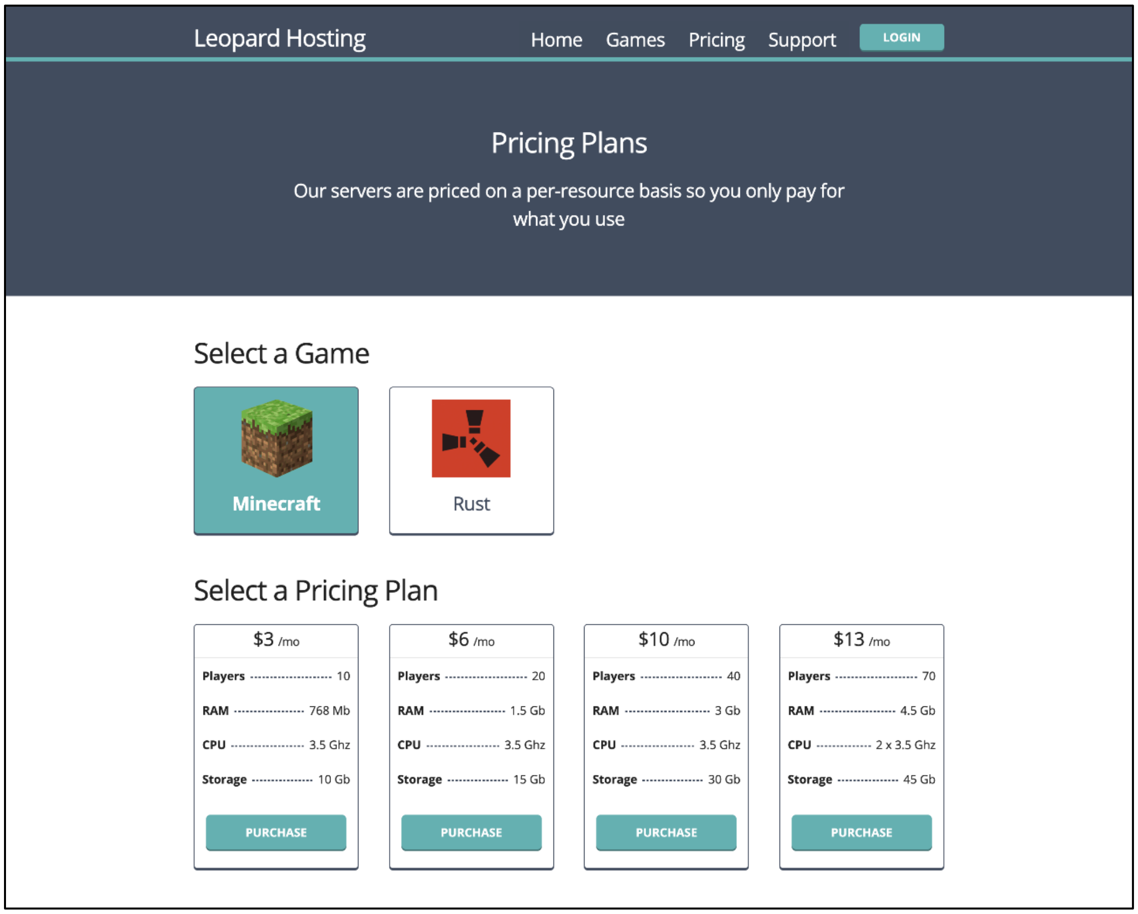
Leopard Hosting makes it easy to deploy a server for your favorite game in a matter of minutes

Starting at $6.99

Starting at $8.99

Starting at $5.99

Figure 10: Games page



Pricing Plans

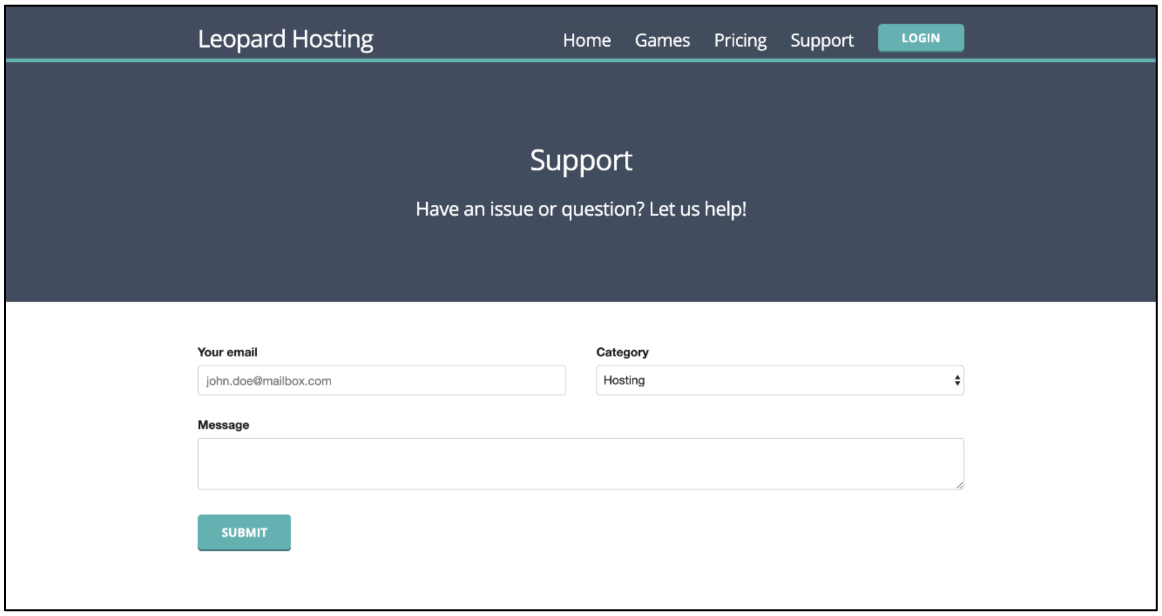
Our servers are priced on a per-resource basis so you only pay for what you use

Select a Game

Select a Pricing Plan

	$3 /mo	$6 /mo	$10 /mo	$13 /mo
Players	10	20	40	70
RAM	768 Mb	1.5 Gb	3 Gb	4.5 Gb
CPU	3.5 Ghz	3.5 Ghz	3.5 Ghz	2 x 3.5 Ghz
Storage	10 Gb	15 Gb	30 Gb	45 Gb

Figure 11: Pricing page



Support

Have an issue or question? Let us help!

Your email

john.doe@mailbox.com

Category

Hosting

Message

Figure 12: Support page

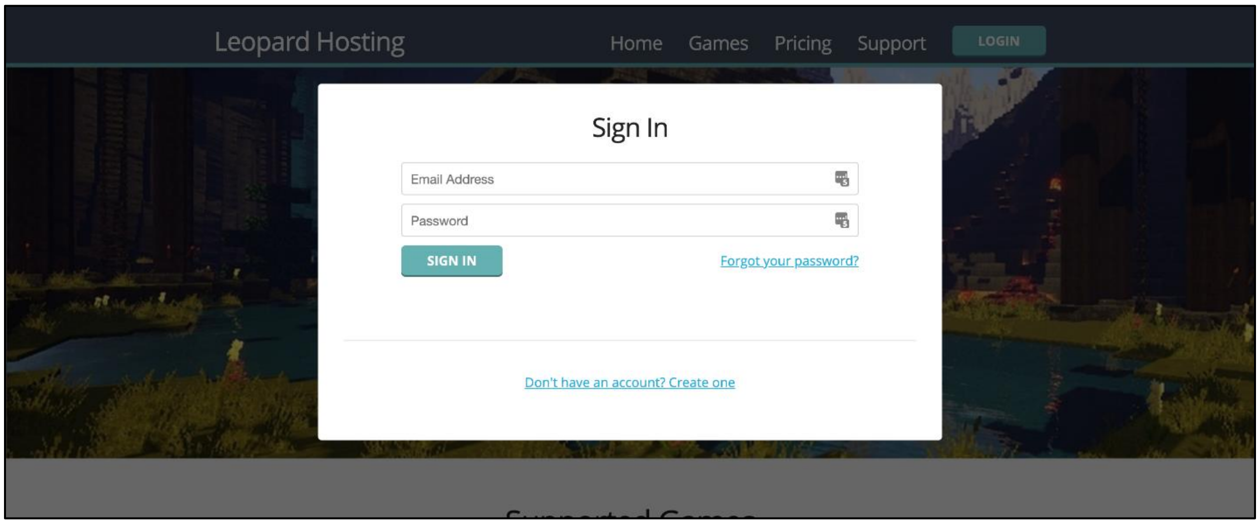
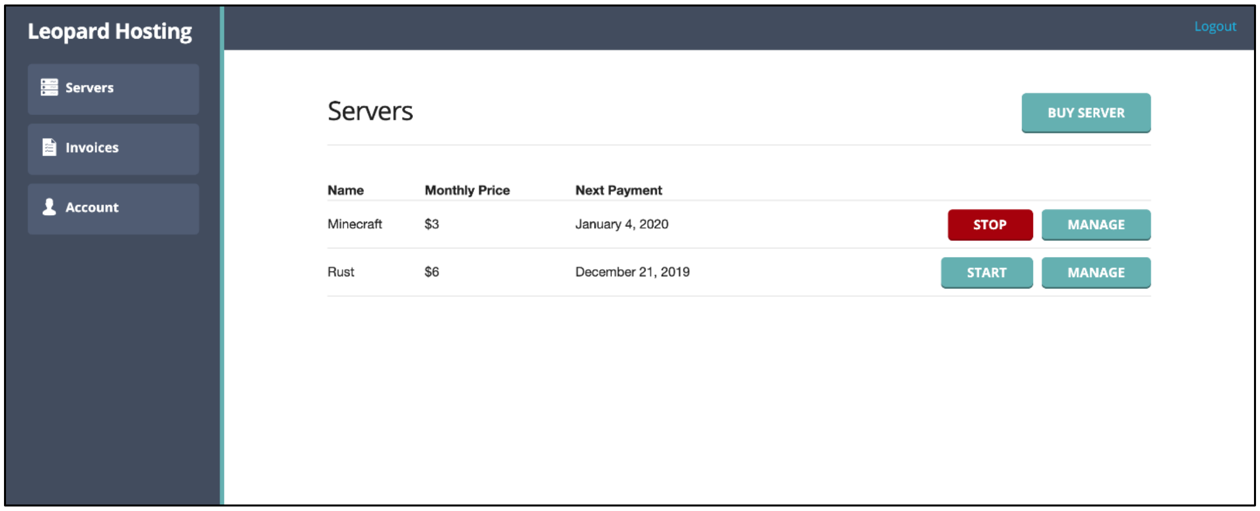


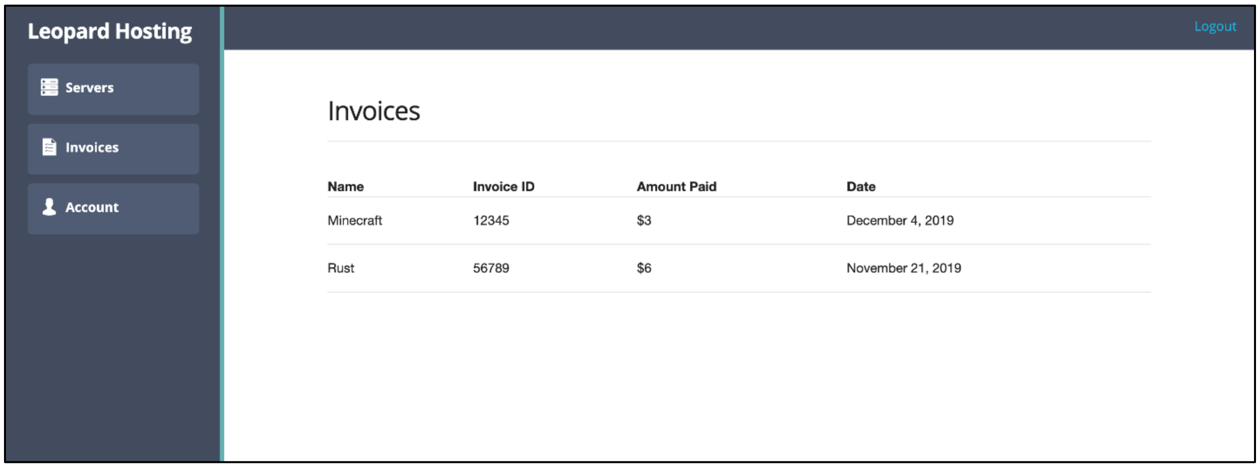
Figure 13: Sign in dialog



Servers

Name	Monthly Price	Next Payment
Minecraft	$3	January 4, 2020
Rust	$6	December 21, 2019

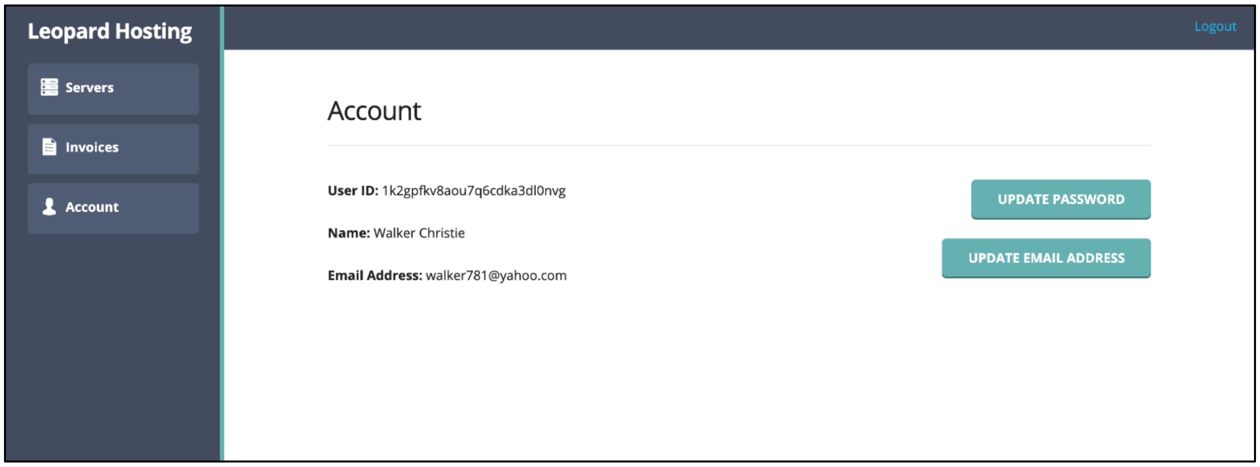
Figure 14: Servers list page



Invoices

Name	Invoice ID	Amount Paid	Date
Minecraft	12345	$3	December 4, 2019
Rust	56789	$6	November 21, 2019

Figure 15: Invoices page



Account

User ID: 1k2gpfkv8aou7q6cdka3dl0nvg

Name: Walker Christie

Email Address: walker781@yahoo.com

Figure 16: Account page

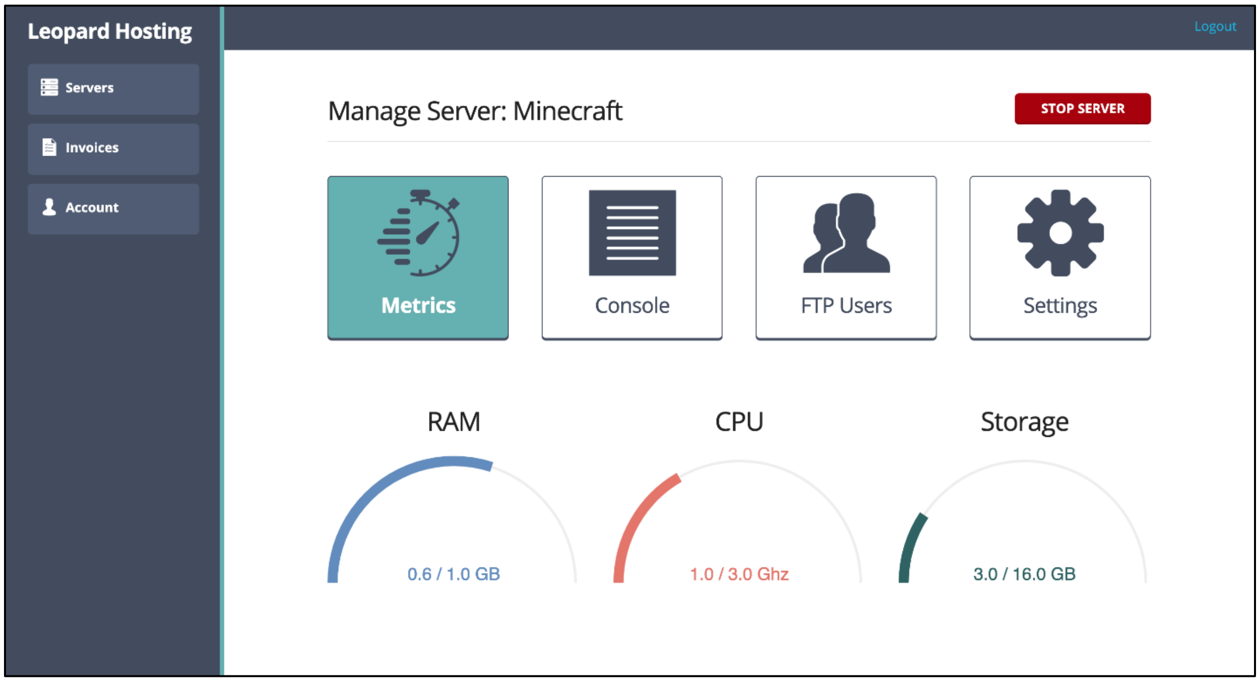
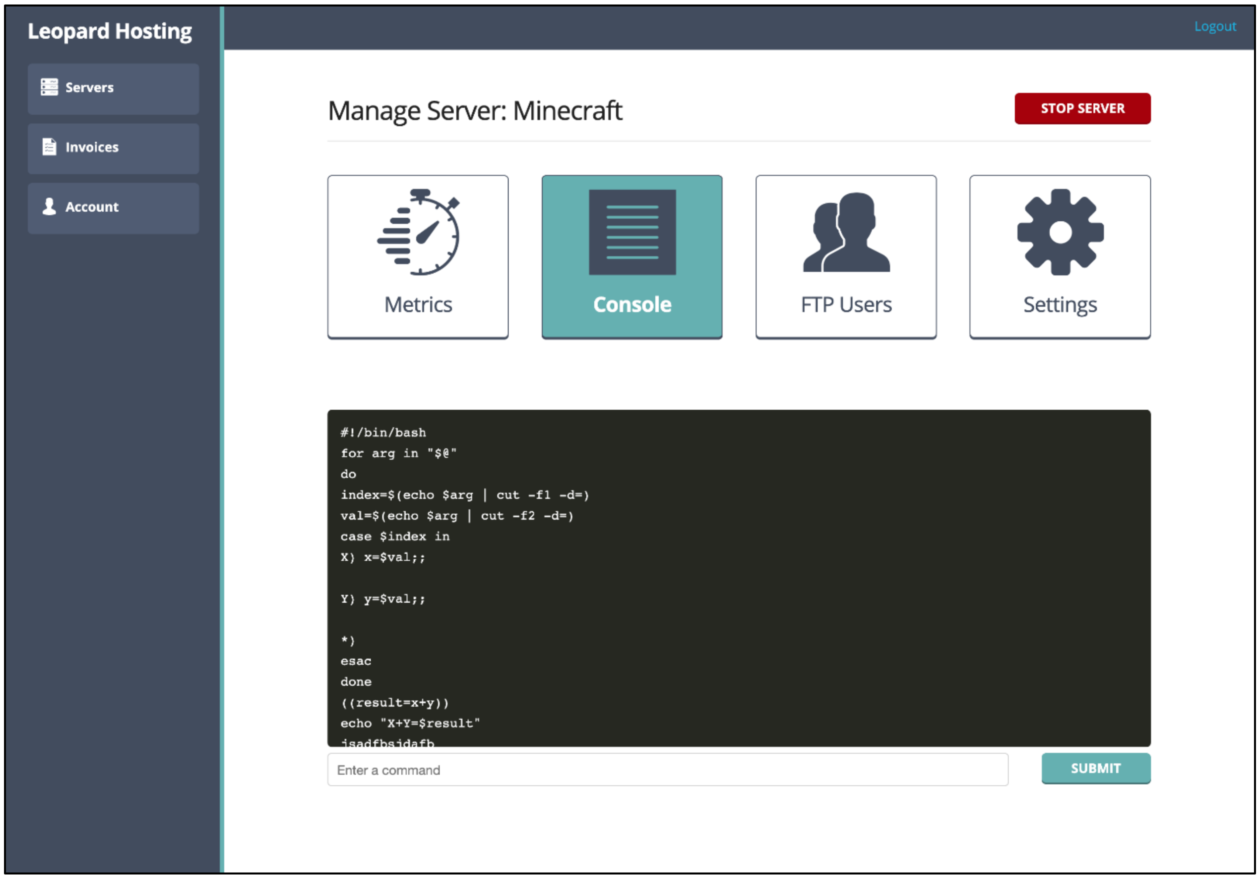


Figure 17: Manage server metrics page

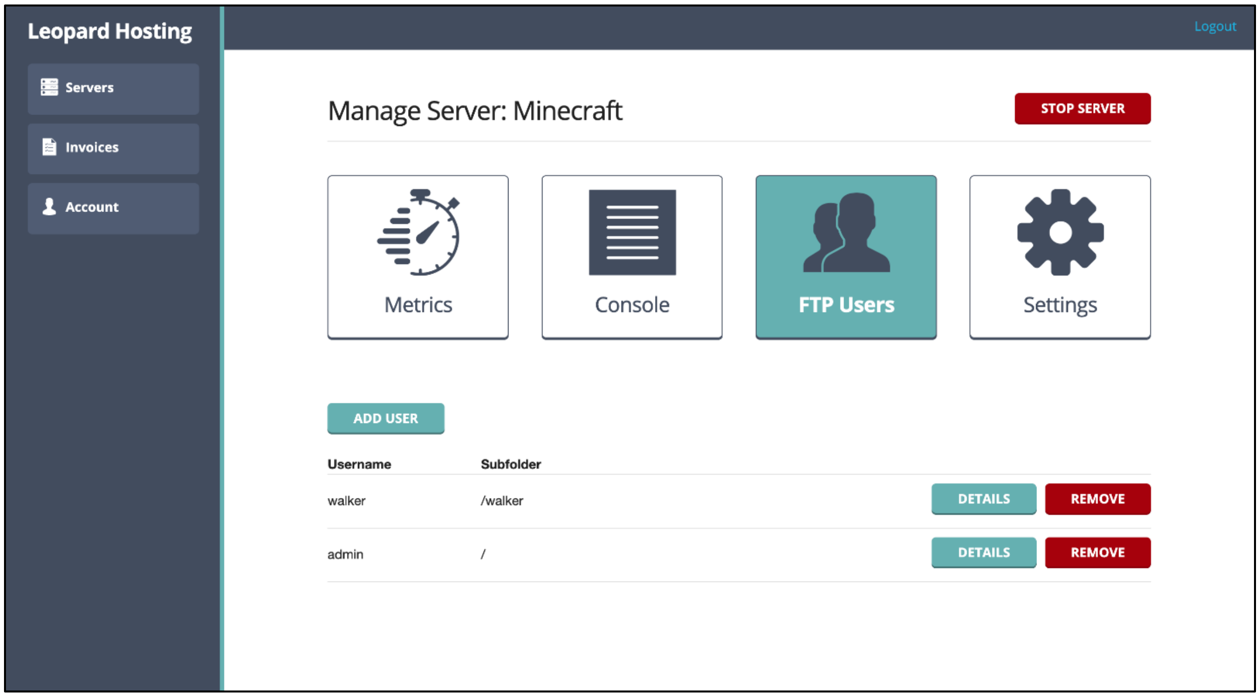


```
#!/bin/bash
for arg in "$@"
do
index=$(echo $arg | cut -f1 -d=)
val=$(echo $arg | cut -f2 -d=)
case $index in
X) x=$val;;

Y) y=$val;;

*)
esac
done
((result=x+y))
echo "X+Y=$result"
jsadfbsjdafb
```

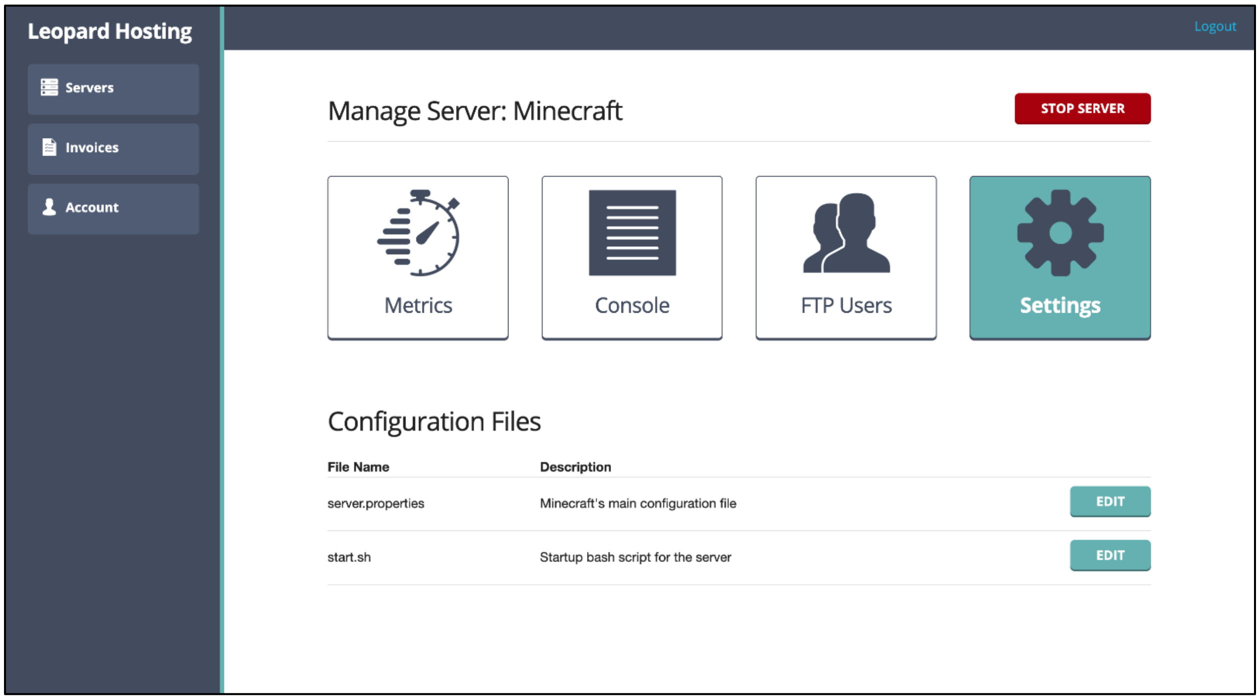
Figure 18: Manage server console page



Manage Server: Minecraft

Username	Subfolder
walker	/walker
admin	/

Figure 19: Manage server FTP users page



Manage Server: Minecraft

Configuration Files

File Name	Description
server.properties	Minecraft's main configuration file
start.sh	Startup bash script for the server

Figure 20: Manage server settings page

8 List of Figures

Figure 1: PERT Chart 10

Figure 2: GraphQL pricing schema 15

Figure 3: GraphQL schema provisioned on DynamoDB....................16

Figure 4 AppSync Resolver for Prices PutItem operation 17

Figure 5 Database Group Permissions Table....................18

Figure 6: Project CloudFormation Template....................20

Figure 7: Minecraft 1.15.1 Dockerfile....................22

Figure 8: Add Server Flowchart 24

Figure 9: Home page....................29

Figure 10: Games page 30

Figure 11: Pricing page....................30

Figure 12: Support page 31

Figure 13: Sign in dialog....................31

Figure 14: Servers list page....................32

Figure 15: Invoices page 32

Figure 16: Account page....................33

Figure 17: Manage server metrics page....................33

Figure 18: Manage server console page....................34

Figure 19: Manage server FTP users page....................35

Figure 20: Manage server settings page 35

9 References

Amazon AWS Documentation. (n.d.). Retrieved from Amazon AWS: https://docs.aws.amazon.com/

Atwood, J. (n.d.). *Understanding Model-View-Controller*. Retrieved from Coding Horror: https://blog.codinghorror.com/understanding-model-view-controller/

Client–server model. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Client%E2%80%93server_model

Docker (software). (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Docker_(software)

Gilbert, B. (n.d.). *'Minecraft' is still one of the biggest games in the world, with over 91 million people playing monthly*. Retrieved from Business Insider: https://www.businessinsider.com/minecraft-has-74-million-monthly-players-2018-1

How to install Minecraft Server on Ubuntu. (n.d.). Retrieved from Foss Linux: https://www.fosslinux.com/18063/how-to-install-minecraft-server-on-ubuntu.htm

Introduction to GraphQL. (n.d.). Retrieved from GraphQL: https://graphql.org/learn/

Mojang. (2011, November 18). Minecraft. Retrieved from Wikipedia.

React Documentation. (n.d.). Retrieved from React JS: https://reactjs.org/docs/getting-started.html

SDLC - Overview. (n.d.). Retrieved from Tutorials Point:

https://www.tutorialspoint.com/sdlc/sdlc_overview.htm

SQL vs NoSQL: What's the difference? (n.d.). Retrieved from Guru 99:

https://www.guru99.com/sql-vs-nosql.html

TypeScript for JavaScript Programmers. (n.d.). Retrieved from Typescript Lang:

https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html