
PINBALL
LIZARD
How to build an Azure-powered game

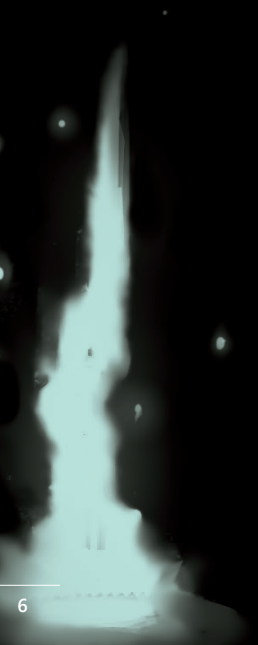
Pinball Lizard: Game Manual

How to build an Azure-powered game

Contents

“We wanted to make a game that was both a blast to play and also a good code teaching tool.”

David Holladay
Sr. Product Manager, Azure gaming
Microsoft



Foreword

About this manual

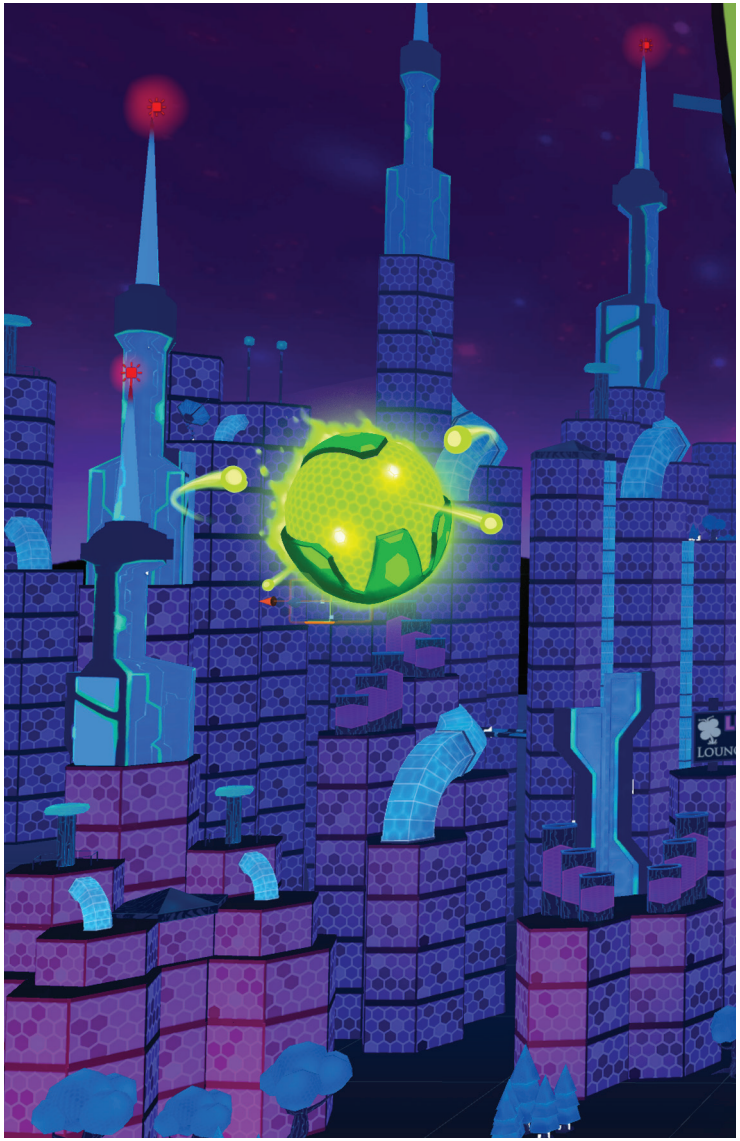
Pinball Lizard started as a challenge. We wanted to explore the gameplay possibilities of the new Windows Mixed Reality headsets and how we could use Microsoft Azure, PlayFab, and Unity to drive those possibilities. In this manual, you'll see how we tackled this challenge and made a proof of concept that we're excited to make public! We hope you play with it, extend it, and make it your own.

Whether you want to check out the underlying game architecture, the gameplay decisions based on mixed reality, how we established communication between Azure and Unity, or how Mixer and PlayFab services are integrated into the game—you'll find our approach documented here.

Chapter 1

Why we chose Azure

When we started thinking about building Pinball Lizard, we knew that modern games are made and played differently. They require advanced development tools, global and flexible multiplayer support, and new revenue models. We also knew that Azure could help—because Azure was made to modernize game architectures.



Here's what Azure can do

Develop faster

Get to market faster with a complete set of services and tools designed for game development.

Scale globally

Scale your game with building blocks from Azure and launch on a global infrastructure.

Monetize intelligently

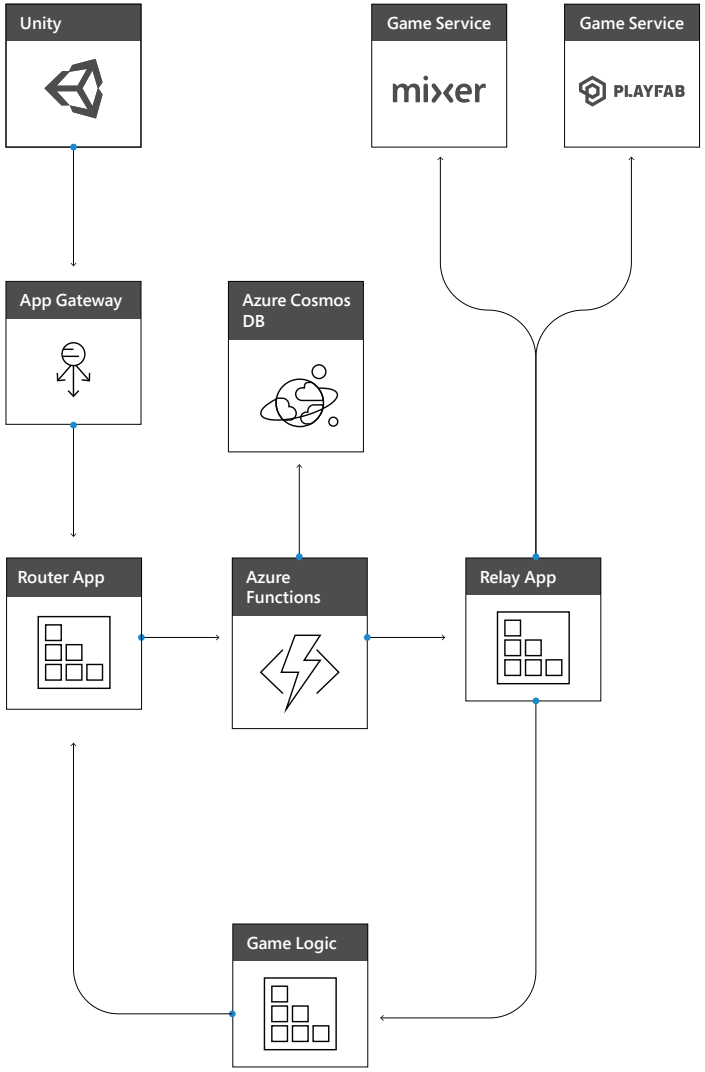
Monetize your games intelligently with LiveOps services, analytics, and machine learning.

Gaming heritage

Build your infrastructure on a gaming cloud with years of experience powering Xbox games and services.

An aerial view

During the design and development of Pinball Lizard, we made architectural choices that showcase the services and features of Azure as a back-end service. We know that some of these choices may not reflect the ideal solution for some game types, but we'll show our reasoning for our architecture, along with sample scenarios that fit each component service.



Let's break it down

Behind the application gateway are the services that act as the brains of Pinball Lizard. It all starts with the request router, which keeps tabs on internal and external service endpoints, and forwards both requests and responses throughout the game system. Serverless functions perform actions based on incoming requests from the game client or containers.

fig 1. **Router App**

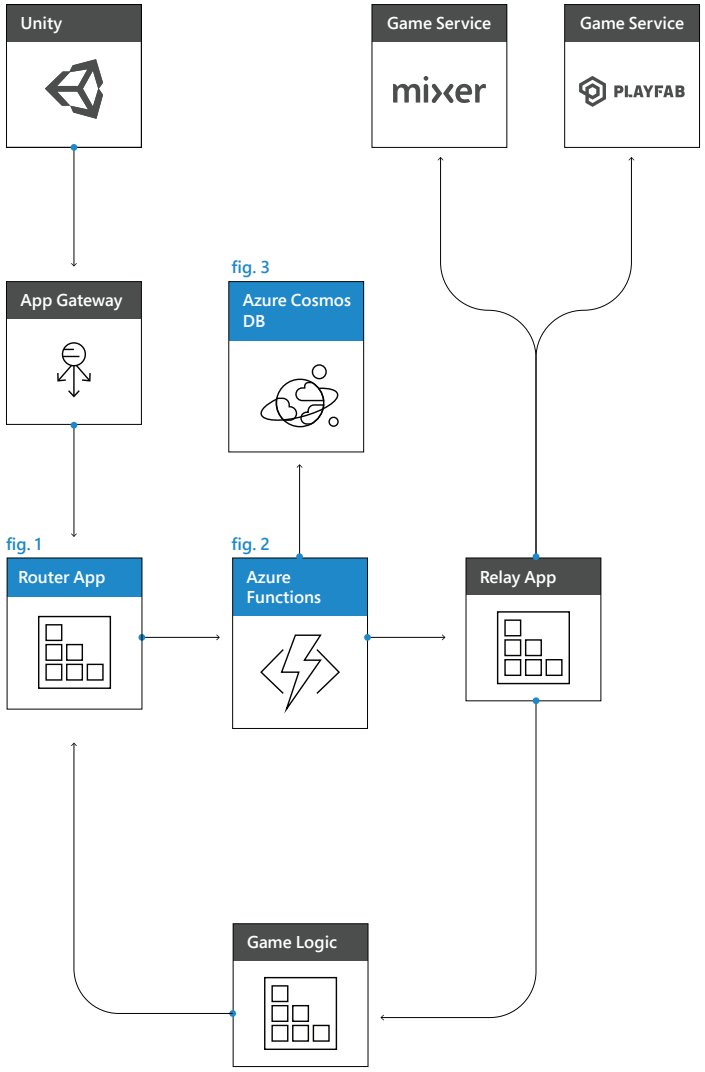
Behind the gateway is a request router that parses requests and responses to and from other endpoints in the game. This provides a single exposed endpoint for the client, which means there's no need to configure the client for all back-end services.

fig 2. **Azure Functions**

Serverless functions make up the bulk of the game's operational capabilities. Functions provide hooks to spawn and despawn units, track building damage, and issue tactical commands to the Unity client.

fig 3. **Azure Cosmos DB**

Azure Cosmos DB stores all game data—from game sessions, to NPC and building states, to telemetry and logs. Session data drives the leaderboards, while telemetry data provides the source for application insights and dashboards.



“We ended up with world-class features and infrastructure by using PlayFab and Azure, and these services can grow to meet our needs.”

Oliver Löffler
Founder and Chief Technology Officer
Fluffy Fairy Games

Chapter 2

Game design

Creating a monster destruction game in VR was tricky. Keeping players immersed meant isolating gameplay to the hands, head, and voice. This restriction drove how the enemies interact with players and how they can destroy the city (sorry, no stomping on buildings—yet).

Windows Mixed Reality headsets

Windows Mixed Reality headsets are really fun, so we wanted to explore the possibilities of these devices and figure out how to merge reality with virtual worlds.

For Pinball Lizard, we focused on the headset's microphone, but we had a lot of ideas for how to leverage the full capabilities:

1. Pestopolis is procedurally generated based on the real room around the player.

2. Real foam objects are transformed into enemy projectiles when thrown at the player.

3. More advanced monster controls make full use of the mixed reality controller's capabilities.

Player abilities with VR

Creating a monster game in VR is an interesting proposition. You need to make players feel powerful, but they can't move around a lot, and they can only interact with their hands. For Pinball Lizard, this meant that stomping on buildings and moving around a city was out of the question. Instead, we chose to play to the medium's strengths and focus on that iconic King Kong scene where he bats airplanes out of the sky. What better way to destroy a city than with the forces they send to try and stop you?

Controls

We chose to ground the game in natural movements that people do on a daily basis. This style would require little player onboarding and let us take advantage of the Windows Mixed Reality headset's voice input functionality.

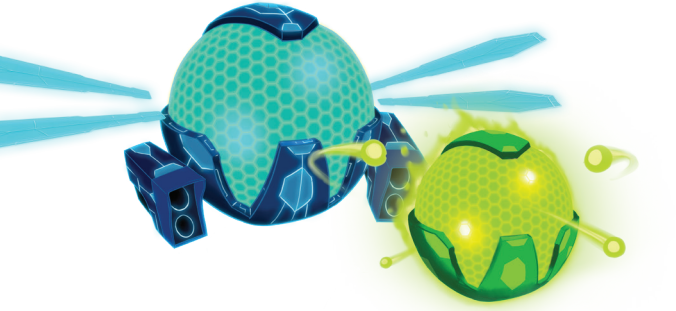
Opening and closing a fist. A simple pull of the trigger opens and closes the player's fist, allowing them to grab and hold a bug. They can then throw the bug by releasing the trigger. A closed fist can also be used to punch and slap enemies.

Eating. To eat, the player holds a bug close to their mouth and makes a chewing and swallowing noise. Devouring an atomic bug charges up the monster and activates the breath weapon.

Breath weapon. An atomic beam that does damage over time to buildings, the breath weapon is sustained by the player's roar. By default, the weapon will fire for one second, but the player can extend the duration for as long as they keep roaring.

The enemies

Because of issues with impact feedback, we decided early on to make the monster invincible and the bugs ultimately unsuccessful in battle. To ensure the monster felt powerful, we also made every bug instantly destructible. We then started thinking about how to vary gameplay because one action becomes stale pretty quickly. Different enemy behaviors helped to create these different beats in the gameplay.

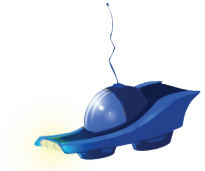


Ice Lice & Atom Ants. The Ice Lice provide the main interaction with the player. Most of them swarm near the city, where they wait to do their attack run. A small group of Ice Lice then charges, giving the player a brief window to slap them all away before they spread out into a large pattern. They can also transform into Atom Ants, which give the monster some seriously bad breath.

Fire Fly. The city's bombers, these guys hang out in a holding pattern above Pestopolis until they are ready to charge, shooting distracting lasers at the player's face.

Arachno Tanks. The city's heaviest firepower, the Arachno Tanks warp in and shoot a huge missile that slowly moves toward the player. After the shot, they warp out and reload.

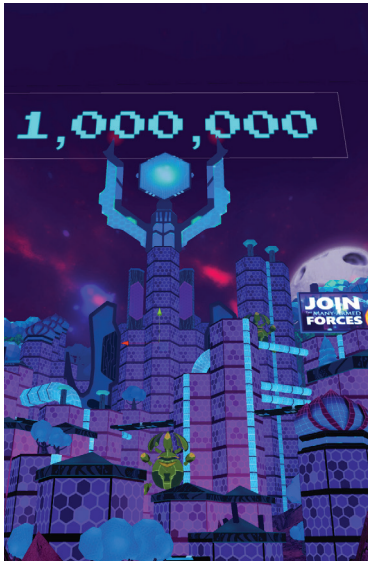
Non-hostile NPCs. To make the city more lifelike, we added non-hostile bug types that move through the scene—but also add opportunities for ricochet.



Taking down the city

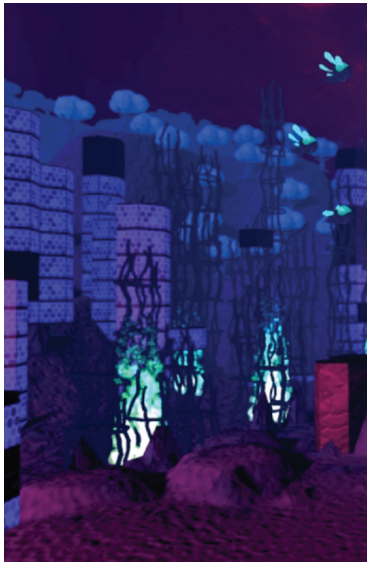
When planning Pinball Lizard, we knew that good feedback response was critical because it would serve as the core gameplay mechanism. That's why the buildings themselves act like bounce pads that accelerate the speed of the bugs thrown at them. As each bug ricochets within the city, the player gains a sense of achievement—and an exponentially bigger score with every bounce.

We also exaggerated destruction to hilarious levels to create an impactful feedback response that would offset the busy, in-your-face gameplay. We divided the destruction into three phases: anticipation, action, and aftermath.



For anticipation, the player can size up the buildings to get a good sense of how much damage each one needs until it's destroyed.

For action, the effects are over the top so the player is sure to take notice, even if the explosion happens on the fourth or fifth bounce of a bug.



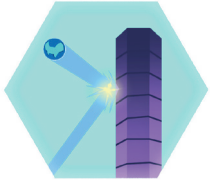
For aftermath, the player can see the remnants of battle—clear evidence of their total and complete dominance over Pestopolis!

Bouncing

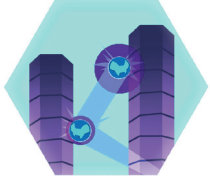
Why are bugs bouncing through Pestopolis? Partly because it's such a cool effect. But also because we wanted to design a core gameplay mechanism that would give players a bounce multiplier score. In testing, we found that using the natural velocity thrown by the player was not enough to generate a lot of bounces. To compensate for this, we multiplied the velocity for every collision a bug has with a building. We took it one step further by allowing the multiplier only for active buildings; wreckage from destroyed buildings delivers no score to discourage the player from decimating other parts of the city.

Destruction

Not that we think decimating the city is a bad thing. In fact, our main destruction goal was to ensure that every player could take out at least 50 percent of Pestopolis in a playthrough. To guarantee these results, we used three tactics: an increasing bounce explosion radius, visual feedback on building health status, and bug ricochets.



Bug ricochets. We placed the main swarm of bugs in front of the city specifically to amplify impact. Hitting bugs in the swarm sends them flying in different directions, creating more opportunities for bounces.



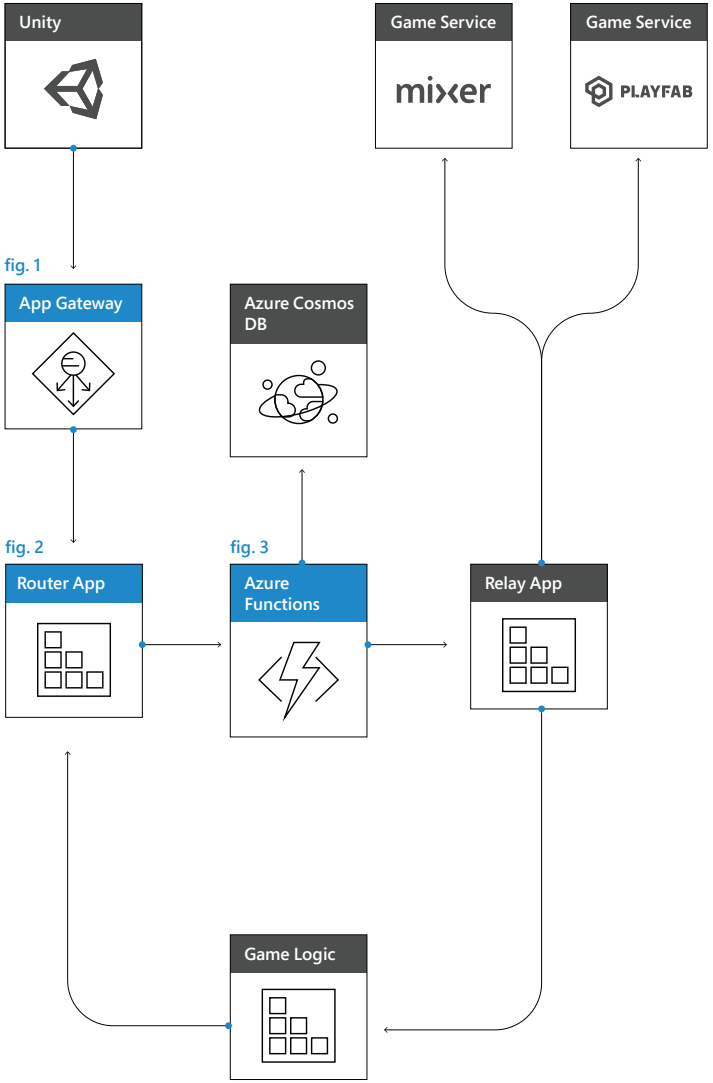
Explosion radius. With each bounce, the impact radius increases additively. This means that the bug starts affecting adjacent buildings with each impact. The first impact is the size of the bug, and each additional impact doubles the size of the radius.

Visual feedback. The buildings are built of modular "blocks," each of which has its own health. Once a block has no remaining health, it explodes and is replaced with a destructed model, showing the player to aim elsewhere to inflict more damage.

Chapter 3

On the server side: Azure

Let's talk about what's going on behind the application gateway. We'll take a look at how Unity communicates with Azure components, how requests and responses are handled, and potential areas for expanding into other services and functionality.



Server-side components

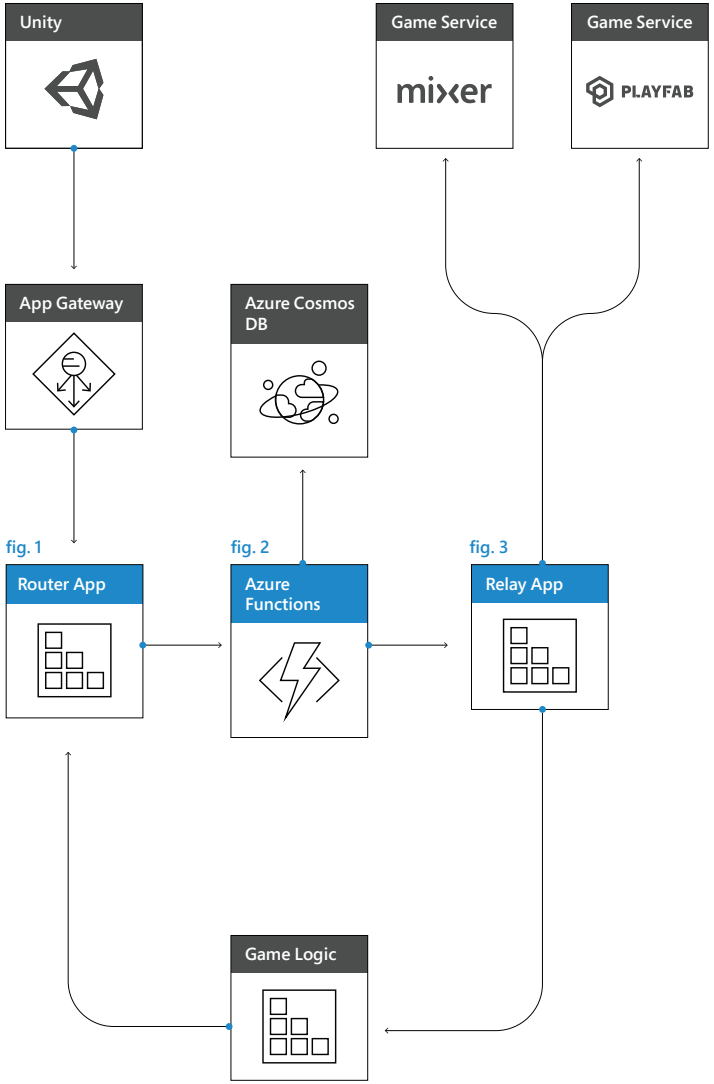
Multiple cloud services make up the "server side" of Pinball Lizard. It all starts with the application gateway and includes application logic, serverless functions, and data storage.

Here's a high-level look at server-side components:

fig 1. **App Gateway**
The application gateway provides the front-facing endpoint for client connections. In our game system, this is how Unity "talks" to our server.

fig 2. **Router App**
The router app manages inbound and outbound communications across the game.

fig 3. **Azure Functions**
Azure Functions provides serverless functions that execute a variety of calls to other services and containers.



Connecting the dots

While Unity contains libraries for network communication, it's a cross-platform game engine, so some libraries aren't available. To keep things simple on the client side, the application gateway serves as the single communication point, with messages sent and received in JSON. Behind the gateway, a routing app parses the request, relays messages to the appropriate services, and then responds to the client.

Here's a high-level look at the communication channels:

fig 1. **Router App**
The router app runs on a container, which manages a list of endpoints related to the game system. The app interprets incoming messages and forwards them to the intended recipients.

fig 2. **Azure Functions**
Azure Functions receives requests from the router app; it then returns a response to the router or forwards results to the relay app.

fig 3. **Relay App**
External calls are routed to the relay container, which interprets internal messages into API calls for other REST endpoints.

App gateway configuration

The application gateway is set up through the Azure portal. In our game system, the gateway provides a web socket connection for the game client. Currently, only Unity is supported, but other authorized clients could be created to use the same service for additional platforms.

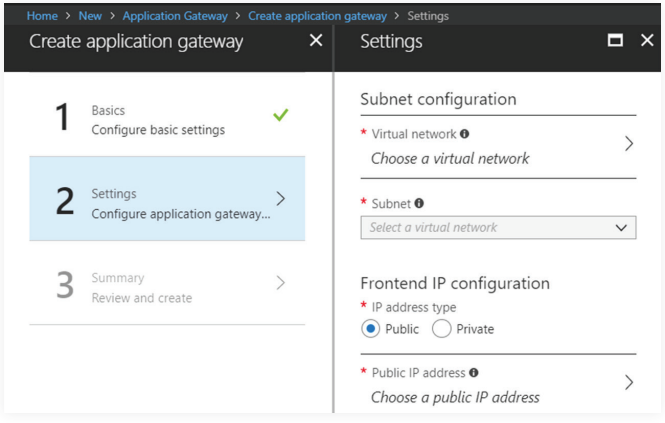


fig 1. **Azure Portal**
You can provision an application gateway through the Azure portal.

Sending your requests

The application gateway hosts a web socket to enable two-way communication between the Unity client and the Azure back end. The singleton class ensures each client instance maintains a single socket for all communication. This single-pipe approach means less multiple-endpoint management from the client perspective. The router app determines the intent of the message, and then forwards it to the correct service.

Game initialization call to the back end [C#]:

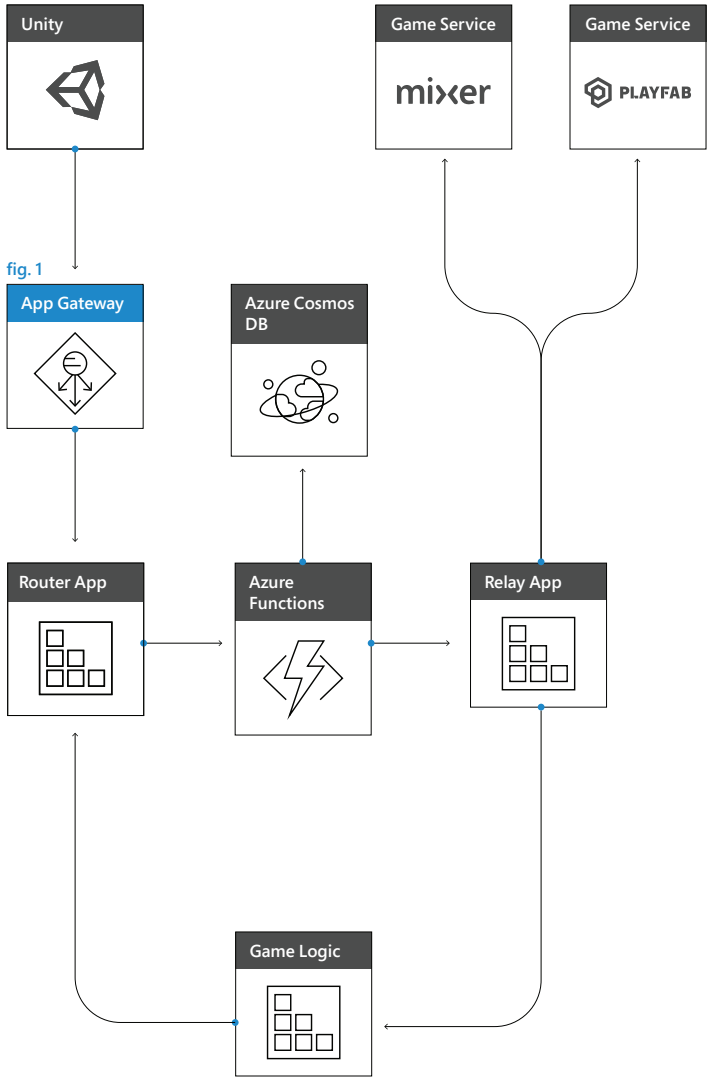
```
public class WebSocketClientSingleton : Single-
ton<WebSocketClientSingleton> {
    private WebSocket webSocket;
    ...
    private void SendInitMessage()
    {
        JsonForAzure initMessage = new JsonForA-
zure();
        initMessage.instance = string.Empty;
        initMessag 4;
        initMessage.spec = 0;
        initMessage.action = 0;
        initMessage.payload.playerName = payload-
PlayerName;

        string initMessageString = JsonUtility.
ToJson(initMessage);
        webSocket.Send(initMessageString);
    }
```

Where your requests go

When a request is made, the router app determines the correct destination for it. That's usually a specific Azure function, but sometimes it's other services—even third-party ones. This allows apps and services to send requests to any endpoint in the service architecture without specific knowledge of the endpoint URL.

Here's a high-level look at where your requests go:

fig 1. **App Gateway**
Game instance identity is part of the messaging standard. It enables the app to manage routing for multiple client instances simultaneously. If needed, the routing app can be load balanced to handle higher request volumes.



What your messages look like

Requests and responses are JSON messages that provide details on the game instance and objects impacted, along with the action to be performed. The payload property contains any specific information required to fulfill the request. For example, requesting a list of buildings would result in populating the payload property with lists for all building IDs. Another message payload may include details on spawning a new NPC or setting waypoints for a specific NPC.

Message content for all requests and responses follows this standardized format:

```
{
  "id": "/MessageSchema",
  "type": "object",
  "properties":{
    "auth":{
      "type": "object",
      "properties":{
        "key":{ "type":"string" }
      },
      "required":["key"]
    },
    "instance":{
      "type": "string"
    },
    "type":{
      "type": "integer"
    },
    "spec":{
      "type": "integer"
    },
    "action":{
      "type": "integer"
    },
    "payload":{
      "type":"object"
    }
  },
  "required":[
   "instance", "auth", "type", "action", "spec",
"payload"
  ]
}
```

Our centralized routing approach

When it came to routing, our approach was to keep things loose and extensible. The current game system uses several Azure technologies, but the routing app works with any web service endpoint, so it's easy to add more services, including third-party endpoints. Endpoints depend on environment variables. These lead to deployment-specific endpoints for each service available, like particular containers or functions.

Destination endpoints are interpreted from the JSON message properties to generate a Uniform Resource Identifier (URI) request:

<target endpoint>/<action>/<game instance id>[/<object id>

```
`${endpoints.ENDPOINT_ENEMY_CONTROL}/
status/${instance}/${npcId}`
```

How we used routing endpoints

Routing endpoints can be configured to connect to private service endpoints within the virtual network. Endpoints can be configured to public service endpoints as well—even those on other cloud platforms, provided they're publicly accessible.

Check out how we used routing endpoints in the following code snippets:

```
// configure endpoints
exports.endpoints = module.exports.endpoints = {
    ENDPOINT_PLAYER_CONTROL:process.env.ENDPOINT_
PLAYER_CONTROL,
    ENDPOINT_ENEMY_CONTROL:process.env.ENDPOINT_
ENEMY_CONTROL,
    ENDPOINT_BUILDING_CONTROL:process.env.
ENDPOINTBUILDING_CONTROL,
    ENDPOINT_INIT:process.env.ENDPOINT_INIT,
    ENDPOINT_DESTROY:process.env.ENDPOINT_DESTROY
};
```

Base URLs are established to endpoints in use from deployment environment variables.

```
ENDPOINT_INIT=https://azurefunctionendpoints.
azurewebsites.net/api/initialize
ENDPOINT_DESTROY=https://azurefunctionendpoints.
azurewebsites.net/api/destroy
ENDPOINT_PLAYER_CONTROL=https://
azurefunctionendpoints.azurewebsites.net/api/player
ENDPOINT_ENEMY_CONTROL=https://functionendpoints.
azurewebsites.net/api/npc
ENDPOINT_BUILDING_CONTROL=https://functionendpoints.
azurewebsites.net/api/building
```

With these base URLs, the router gleans intent from the message to determine any additional endpoint information, and then it builds the full-request URL.

```
'use strict'

const m = require('../constants/public-route-
constants');
const {endpoints, keys} = require('../../index');

let routes = {};
routes[m.type.ENEMY] = {};

/********** Ice Lice **********/

routes[m.type.ENEMY][m.spec.ENEMY.ICE_LICE] = {};

routes[m.type.ENEMY]
      [m.spec.ENEMY.ICE_LICE]
      [m.action.ENEMY.SPAWN] = reqBody => {
        const instance = reqBody.instance;
        return {
          ep: `${endpoints.ENDPOINT_ENEMY_CONTROL}/
spawn/il/${instance}`,
          headers: {
            'x-functions-key':keys.ENDPOINT_ENEMY_KEY
          }
        };
      };
```

Handling requests with serverless functions

Azure Functions handles requests forwarded by the routing app. Specific functions include getting a list of enemy units, spawning new units, or setting players' scores.

Azure Functions can handle requests even if multiple game instances are run on the same infrastructure. The function seeks out waypoints and, if available, creates a unit and assigns an open waypoint location.

Azure Functions works well with other services, like logging telemetry or persisting data to a data store. For an inside look at activities like tracking leaderboards, check out Chapter 5, which focuses on how functions persist data.

Below is a request for spawning an enemy:

```
const spawn = {
    allowedTypes: [
        "il", "at", "ff"
    ]
}
module.exports = function (context, req) {

    // check that session was initialized
    if( !util.checkSession( context ) ) {
        return context.done();
    }

    let npcs = context.bindings.npcs;
    const reqBody = util.parseJSON( req.body ) ||
util.fillReq();
    const count = npcs.length;
    const type = req.params.type;
    const instance = req.params.instance;

    // validate incoming type
    const valid = util.validate( type, spawn.allowedTypes );

    if( valid && instance ) {
        const id = context.invocationId;
        // determine new waypoint
        const waypoint = npc.findWaypoint( npcs, type );
        if( waypoint >= 0 ) { // valid waypoint
            // create a new npc entry
            context.bindings.spawn = JSON.stringify(
                npc.create(
                    instance, id, type, count + 1, waypoint
                )
            );
            // create output
            context.res = out.compile(
                        200,
                        c.type.ENEMY,
                        reqBody[c.labels.SPEC],
                        reqBody[c.labels.ACTION],
                        null,
                        reqBody,
                        JSON.parse( context.bindings.spawn )
);
        } else { // no more waypoints to give
            context.res = out.compileError(200, c.message.
NO_MORE_WAYPOINTS, reqBody);
        }
    } else { // invalid incoming type
        // 404
        context.res = out.compileError( 404, c.message.NOT_
FOUND, reqBody );
    }
    // complete
    context.done();
};
```

Chapter 4

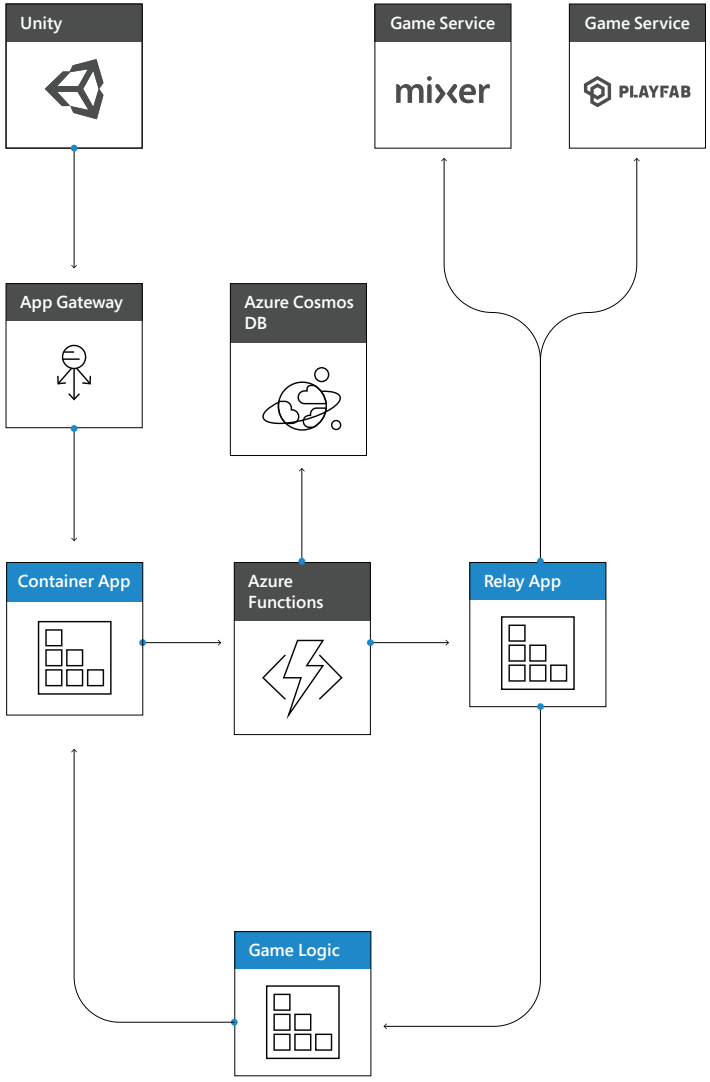
Game objects

A core design focus for Pinball Lizard was to support extensibility. We went with containers because they provide a lightweight environment to host more complex applications—at least in scenarios where a single function isn't feasible.

Using containers to host lightweight apps

In-game objects come to life thanks to Unity, but functions and containers manage these objects and provide directions on how and when they should behave. Because the game's back end is designed to work with multiple client instances, it's possible to scale out specific containers, if needed. For multiple clients, the routing container is a potential bottleneck—considering the router processes all requests—but this can be scaled and load balanced across multiple routing containers.

Containers provide endpoints for some of the game's tactical decisions. Unlike functions, the containers run apps (Node.js) to handle more complex tasks. Container images are stored in a custom repository and deployed to container groups during provisioning. Because containers are quickly instantiated, scaling can meet demands as needed with the orchestration of Azure Service Fabric or Kubernetes.



NPC control

NPCs are managed with one container, but more populous units could be managed by multiple containers. These containers keep tabs on their assigned enemy types while tracking waypoints, spawns, destructions, and overall tactics.

At the controller level, NPCs are assigned formations. Each formation has assigned units, a center point, and a list of offsets. The NPC controller assigns individual units to the formation and provides a center point that the formation moves toward. The client, given the center point and offsets, manages the movement of units to positions relative to the desired waypoint.

Example: The formation (abridged):

```
class Formation {
  constructor( id, waypoint, members,
  offsetAlgorithm )
{
      this._id = id;
      this._waypoint = waypoint;
      this._members = members;
      this._offsets = offsetAlgorithm( members.
      length );
      this._offsetAlgorithm = offsetAlgorithm;
}
  get payload() {
    return {
      formationId:this.id,
      members:this.members,
      listOfOffsets:this.offsets,
      waypoint:this.waypoint
    };
  }
      ...
}
```


As the game progresses, the controller updates instructions to formations based on the current state of the playing field. These updates may orchestrate the location of a formation, assign new formation offsets, or disband a formation completely.

Some methods for modifying formations:

```
set members( members ) {
  this._members = members;
}

set offsetsAlgorithm( oA ) {
  this._offsetAlgorithm = oA;
  this._offsets = offsetAlgorithm( this._
  members.length );
}

addMembers( members ) {
  this._members = this._members.concat(
  members );
}

removeMembers( numToRemove ) {
  let members = this._members.slice(
  0,-numToRemove );
  let removed = this._members.slice(-
  numToRemove );
  this._members = members;
  return removed;
}
```

Formations are self-correcting. If one unit in a formation is destroyed, the remaining units will move up to fill the open spots. The controller will then attempt to assign replacements, if available, to the tail of a formation to replenish the ranks.

Tactics

The NPC controller makes tactical decisions based on the state of the playing field—how many units are available, how many buildings are standing, and how much time is left. Given these inputs, the controller selects a tactical mode. The tactical mode is predefined for now, along with formations and offsets.

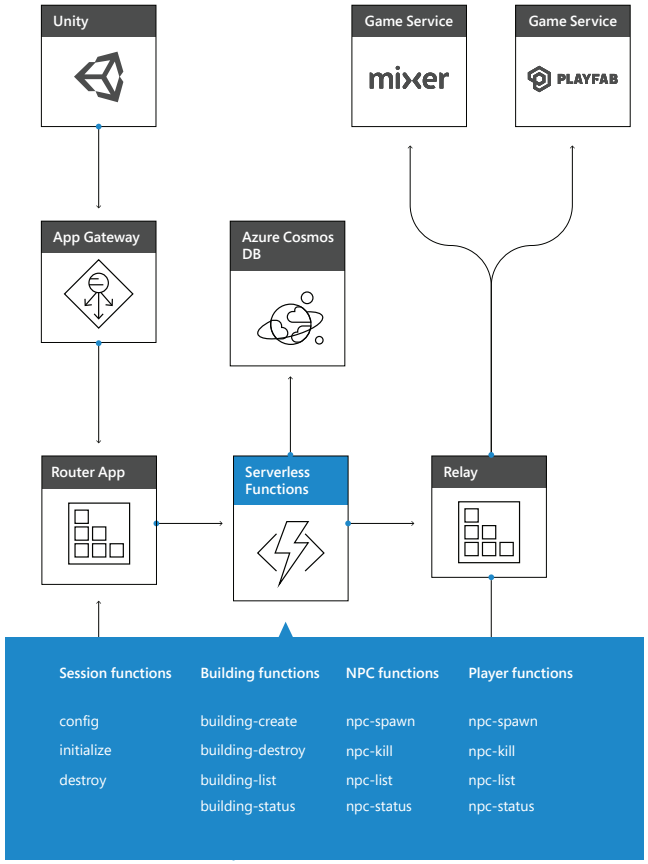
Some actions the NPC controller can prescribe to a game session, which Unity then executes in-game:

```
actionCommands.tankBarrage = instance => {
   return router.passthru( instance,
   c.FORMATION_TYPE, c.NPC_TYPE_ARACHNO_TANK,
   c.FORMATION_SPLIT_ACTION );
}

actionCommands.airstrike = instance => {
   return router.passthru( instance,
   c.FORMATION_TYPE, c.NPC_TYPE_FIRE_FLY,
   c.FORMATION_SPLIT_ACTION );
}

actionCommands.scatterIceLice = (instance,
formations) => {
   return router.passthru( instance,
   c.FORMATION_TYPE, c.NPC_TYPE_ICE_
   LICE, c.ICE_LICE_SCATTER_ACTION,
   {listOfMembers:formations} );
}
```

What's exciting is that we can easily modify this approach to be more dynamic, allowing for unique and challenging gameplay in future revisions. By coupling this functionality with saved gameplay data, we could create a model that could adapt tactics and decisions based on past player performance.

Serverless functions

Azure Functions provides single-action endpoints for the game's back end to perform specific tasks. These can range from game management functions—like initializing or ending a session—to sending gameplay commands to the client—like spawning NPCs or updating a formation.



Session functions

Session functions perform actions at an individual game session level—configuring, initializing, and destroying. For example, when a session begins in Unity, an init request is sent through the app gateway and router to the initialize function.

Building functions

Building functions take actions on buildings in a game session. In future revisions, these could be expanded to include features like map generation and building qualities, such as damage state, digital signs, particle generation, and other animated information.

Getting the status of all buildings in-game:

```
for( let x of buildings ) {
    for( let k in x ) {
    if( li.substring( 0, 1 ) === "_" ||
    blacklist.includes( k ) ) {
                delete x[k];
            }
        }
    if( x[c.labels.BUILDING_STATUS] ===
    c.message.BUILDING_INTACT ) {
            intact.push( x );
        } else {
            destroyed.push( x );
        }
    }
context.res = out.compile( 200, c.type.BUILDING,
reqBody.spec, reqBody.action, null, reqBody,
    {
    [c.labels.BUILDING_LIST]: buildings,
    [c.labels.BUILDING_INTACT]: intact,
    [c.labels.BUILDING_DESTROYED]: destroyed
    }
);
```


NPC functions

NPC functions act on the various enemies in the game. Actions include spawning/despawning, listing NPCs in-game, and getting NPC status.

Spawning a new NPC:

```
// create a new npc entry
    context.bindings.spawn = JSON.stringify(
        npc.create(
            instance, id, type, count + 1,
            waypoint, color
        )
    );
// create output
    context.res = out.compile(
        200,
        c.type.ENEMY,
        reqBody[c.labels.SPEC],
        reqBody[c.labels.ACTION],
        null,
        reqBody,
        JSON.parse( context.bindings.spawn )
    );
```

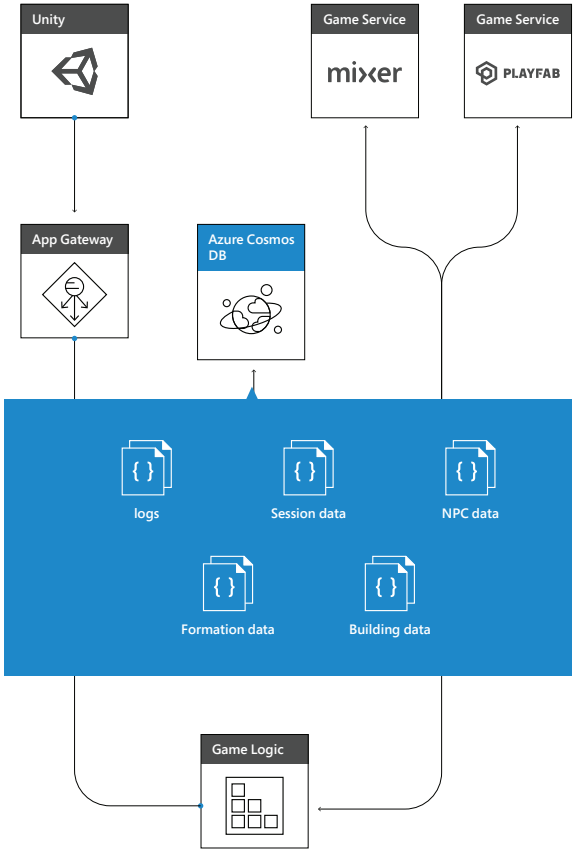
Chapter 5

Data storage

Data is essential, and not just for gameplay. It also fuels analysis for measuring performance and improving the player experience. With Cosmos DB, we can store data at various levels and collections for future review.

Minding your data with Cosmos DB

Cosmos DB supports multiple data models, making it a powerful repository for collecting different types of data—from telemetry and logs to user stats and leaderboard scores. Storing this data allows you to gain insights from gameplay and create compelling data views on infrastructure and services.



Session data

The session collection stores information for each game instance logged. Currently, initialization data and parameters are stored, but the collection can be expanded to store additional session-level data like scores and other metrics.

NPC data

The NPC collection tracks the status of all things NPC, including individual units, unit type, assigned formations, and parent session.

Formation data

The formation collection stores data about NPC group formations. Currently, this is limited to the identifier, parent session, and unit count. However, it can be expanded to include more information, such as formation shape or tactical/strategic details that the group can act on.

Building data

Similar to formation data, the building collection stores information about buildings. Currently, these documents identify which buildings are still standing, but they can be expanded to include structure-specific information or status values. In turn, this information can be used to determine how much damage to display in the client.

Logging your successes

In Cosmos DB, data can be logged to document collections through functions, container apps, or even the game client. Document collections can be accessed using the MongoDB API.

Logging a building's destruction to Cosmos DB:

```
if( building ){
    // compile response
    const msg = building[ c.labels.BUILDING_STATUS ]
        === c.message.BUILDING_DESTROYED
        ? c.message.BUILDING_ALREADY_DESTROYED
    : c.message.BUILDING_KILLED;
    // update cosmosdb
    building[ c.labels.BUILDING_STATUS ] = c.message.
    BUILDING_DESTROYED;
        context.bindings.buildingOut = building;
        context.res = output.compile( 200, c.type.
        BUILDING, reqBody.spec, reqBody.action, null,
        reqBody,
    {
        [c.labels.MESSAGE]: msg,
        [c.labels.BUILDING_ID]: building[c.labels.
        BUILDING_ID],
        [c.labels.BUILDING_STATUS]: building[c.labels.
        BUILDING_STATUS]
    }
    );
    } else {
        context.res = output.compileError(404,
        c.message.NOT_FOUND, reqBody);
    }
```

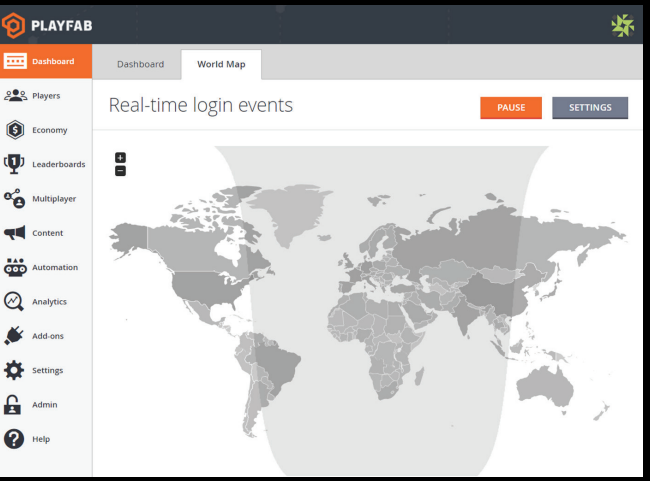
Chapter 6

Extending with Microsoft game services

While Pinball Lizard is a single player game, we're always looking to bring in new ways to interact with the environment. PlayFab adds dynamic leaderboard and dashboard capabilities, while Mixer adds a little more chaos (and fun).

PlayFab

PlayFab provides multiple APIs to jump-start game development. For Pinball Lizard, we decided to push some player and session data to drive a leaderboard. But that's just the beginning—PlayFab can plug in services like dashboards, matchmaking, reward systems, and in-game currency.



To get Pinball Lizard talking to PlayFab, we provisioned a container that takes requests from the relay app and packages player data for consumption by PlayFab.

Packaging player data for PlayFab:

```
const Playfab = require('../helpers/playfab-
utility');

  const createUserName = (username, uuid) => {const
  parts = uuid.split('-');
  return username === parts[0] ? uuid :
  `${username}_${uuid}`;
}
exports = module.exports = async (req, res, next) => {
  if( !req.body.username || !req.body.instance ) {
    res.sendStatus(400);
    return;
  }
  req.username = createUserName( req.body.username,
  req.body.instance );
  req.uid = await Playfab.getUID( req.username );
  next();
}
```

With the data logged to PlayFab, you can use the PlayFab console to work with the data to create leaderboards or dashboards.

```
app.setStats = async ( UID, stats, version=undefined )
=> {
  let pfStats = [];
  for( let name in stats ) {
    pfStats.push({
    StatisticName:name,
    Version:version,
    Value:stats[name]
  });
  }
  let res = await promisify( Server.
  UpdatePlayerStatistics,
    [{
      PlayFabId:UID,
      Statistics:pfStats
    }]
    );
    return res;
}
```

Mixer

Mixer provides the opportunity to integrate interaction capabilities into the game experience. Mixer users can view live streams from other players and interact with hosts in various ways, including commenting, voting, or using UI elements (like buttons or sliders) to impact the gameplay stream in real time.

Establishing and closing a connection between Pinball Lizard and Mixer:

```
client.on('send', payload => {
  payload = util.safeParseJSON( payload );
  if( payload.method === c.MIXER_METHOD_VALIDATE_READY
  ) {
    // check param isReady
    active = payload.params.isReady || false;
    if( payload.params.isReady ) {
      active = true;
      console.log( 'Interactive Active.' );
    }
  }
});

const openClient = async instance => {
  await client.open( { authToken, versionId } );
  instanceId = instance;
}

const closeClient = async instance => {
  if( instance === instanceId ) {
    await client.close();
    messages.participants = {};
    active = false;
  } else {
    console.log( `${instanceId} still active...` );
  }
}
```

Here we've done something simple: Mixer users can spawn an Atom Ant. This bug has some properties that—when eaten by the monster—may yield unexpected results.

Sending a request from the Mixer handler to the game client:

```
actions.mixerUserInput = ( payload, instance ) => {
  let user = pub.participants[ payload.params.
  participantID ];
  if( payload.params.input.controlID === c.MIXER_
  CONTROL_ID_PINK_ICE_LICE
  && payload.params.input.event === c.MIXER_PARAMS_
  EVENT_MOUSE_DOWN
  ) {
  console.log( `${user.username} pushed button...` );
    spawn.send( spawn.SPEC_ICE_LICE, instance,
    'atomic' );
  }
}
```


ROAR!
JOIN
THE MANY ARMED
FORCES
GNAT
KING COLE
LIVE
LUCKY
BEETLE
LOUNGE & CASINO

Take the next step

Get the game code

Access the full code for Pinball Lizard directly from GitHub! You can view or download the related files by visiting aka.ms/lizardcode.

Learn more about Azure gaming

While you're planning your game, don't forget to visit azure.com/gaming to learn more about how Azure can help.

See how easy it is to build, launch, and scale across any platform with Azure. You can start a free trial with $200 in credit by visiting azure.com/free.

Credits

David Holladay Executive Producer
Mahraan Qadir Executive Producer

Jeremy Bonner Creative Director / Game Designer
Mary Lou Ricci Project Manager
Chad Welch Concept Artist / Game Designer
Grace Galarosa 3D Artist
Jonathan Wiley 3D Artist
Majesta Vestal Designer
Andrew McCormick Unity VR Developer / Game Designer
Dustin Jorge Back-End developer
Mike MacGregor Architect
Robyn Schmitz Writer
Lisa Rosenberger Writer
Eric Munch Sound Designer
Megan Nolan Voice Actor


aka.ms/lizardcode
azure.com/gaming
playfab.com
Microsoft