CHAPTER 1

Getting Started

Game engines such as Unity and the Unreal Engine simplify the development process and allow you to start developing quickly. In this chapter, you will create a simple game in Unity, which will help you see the backend features discussed in the following chapters in action.

First, however, you learn about and set up some key technologies that you will use throughout this book—Unity, PlayFab, Terraform, and Azure, to name the most important ones.

Then, you will set up Mirror Networking (or Mirror), which enables the basic networking functionality for this game. With the help of Mirror, you can turn this simple game into a local multiplayer one.

Thereafter, you will build a minimum infrastructure in the cloud to host your game server and enable an online multiplayer game.

By the time you complete this chapter, you will have a good foundation for building backend game features. Let's get started.

Game Frontend and Backend

To begin, it is important to define the game's *backend* and *frontend,* at least within the scope of this book.

When implementing a client-server topology for networked games, the software that runs on the clients is called the game's frontend. The server contains all the services (which can be a dedicated multiplayer server as well), and this is the game's backend. See Figure 1-1.

B. Bors, *Game Backend Development*, https://doi.org/10.1007/978-1-4842-8910-5_1

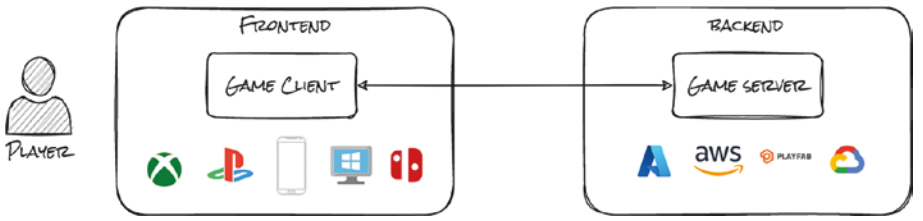

Figure 1-1. *Game frontend and backend*

This is analogous to a web application architecture, where the frontend is what you see in your browser (developed in Angular, React, etc.), and their backends (developed in PHP or Java) are running on a server in the cloud.

Game Frontend

A frontend developer of a game has to deal with many more graphical components and effects than a developer of a web application. The frontend includes your game running on your device. This device can be a PC, mobile phone, or console (Xbox, PlayStation, Nintendo Switch, etc.).

You'll use Unity in this book to implement the game's frontend. Unity is a cross-platform game engine that enables everyone to start creating games, even with limited resources. It provides out-of-the-box 2D and 3D graphics, animation, physics, virtual reality, and a lot more. You can start using it for free and pay only when your game gets traction. It is very popular among individual game developers.

Game Backend

All the servers and services are in the backend, and they are not directly visible to the clients (or players). Still, without them, the game can only operate in a very limited capacity. For example, backends allow buying items or playing with other players.

These backends are hosted in a central location, by a *cloud* or *GBaaS* (*Game Backend-as-a-Service*) providers, and the clients connect to them. Next, you'll review what a cloud is and the added value of GBaaS compared to pure cloud services.

Cloud Computing

Hosting dedicated servers was a difficult task in the early days. You had to invest in expensive IT server infrastructure, which then served the potential clients. In the worst case, the servers stood there without being used. It was hard to buy the exact necessary resources.

Cloud computing, among others, solved this problem. It allows you to reserve resources based on your actual needs. You can scale the servers up and down accordingly and pay as much as you use. A cloud gives you a flexible way to host your servers and provides a couple of important backend services that you can use for your game.

There are a lot of cloud providers nowadays. The three most prominent ones are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. The competition is fierce, and they keep on improving their services. They are always cheaper, and they give you a free tier where you can try their offerings for free or with discounts.

In this book, you'll implement the backend features on a Microsoft Azure cloud. You can use these ideas and concepts with any other cloud providers, as they are very similar.

Game Backend-as-a-Service (GBaaS)

The rising popularity of GBaaS providers enables game developers to achieve faster time-to-market and deal with the backend infrastructure more easily. Developers do not need to worry about building an infrastructure. With the help of APIs, game developers can easily access and utilize sophisticated backend features.

GBaaS is also backed by a cloud service, but it provides an additional layer on top, which implements specific services for games. In this book, you'll use PlayFab, which is backed by Azure.

Choosing GBaaS vs. a Cloud

Both have advantages and drawbacks, so it depends on your situation and requirements. Generally, building your own custom backend on the cloud is better for bigger projects and teams, while GBaaS is a great fit for smaller or individual developers. See Table 1-1.

Table 1-1. *Comparing Cloud and GBaaS (the More +, the Better)*

	Cloud	GBaaS	Comparison
Difficulty	+	+++	Learning the cloud takes more effort.
Flexibility	+++	+	Cloud's biggest advantage is that it is much more customizable
Implementation efforts	+	+++	With the cloud, you build your own infrastructure and develop the logic.
Maintenance	++	+++	GBaaS provides a fully managed backend infrastructure.
Price	++	+	The cloud provides more options to optimize cost and fit resources to your needs.
Quality	+++	++	Highly depends on providers. But big cloud providers can invest more in availability, security, etc.

In this book, you will find an implementation of each feature with both pure cloud and GBaaS providers. Sometimes you can integrate these two worlds; for example, in the case of CloudScripts, you can call Azure functions through the PlayFab API.

You can also create a hybrid solution, where you implement some of the features on the cloud and some of them with GBaaS. This book aims to help you choose the right situation for your needs and gives you a comprehensive view of both options.

Setting Up the Development Tools

Now you can move on to the practical part. First, you will learn how to set up your development environment and install the required tools.

Unity Editor

You'll start with the frontend. To experiment and learn about the features discussed in this book, you need to have Unity (at least version 2021.3.6f1). You can download it from:

https://unity3d.com/get-unity/download

Because this book focuses on the backend part, we do not discuss Unity any further here. The Unity game example is very simple, and the goal here is to learn how to use backend features. You can also use your own game as a frontend.

Visual Studio Code

Next, you'll get ready for the backend. You will develop the backend infrastructure with code, much in the same way you do for an application.

I suggest developing your infrastructure code in a separate place from your game code, which is in the sovereignty of the Unity Editor.

Basically, you can use any text editor to develop your infrastructure code. I use Visual Studio Code, as it is free, and it provides a convenient integrated development environment with its extensions.

https://code.visualstudio.com/

Note Install the HashiCorp Terraform extension for syntax highlighting, which will help you develop your code.

Terraforming Your Infrastructure

In this book, you use Terraform to provision the servers and accompanied services in the cloud. Terraform allows you to define your infrastructure as source code. It is also called Infrastructure-as-Code (IaC).

Terraform declares every resource (servers, networks, etc.) with all of their attributes in files. It supports multiple cloud providers.

Generally, learning the working mechanisms of Terraform is an extremely worthy skill. You can use it to easily build infrastructures, also on other clouds. It comes with the HashiCorp Configuration Language (HCL), which is easy to learn and use.

You just define the infrastructure components you need in one or more files. Then, you build it in the cloud with one command. With another command, you can destroy the whole thing. You can be sure that nothing is left behind. The next time, you can rebuild the same exact infrastructure. To achieve this manually through the portal or the command-line interface (CLI) is almost impossible. The more your infrastructure grows, the more you will need an IaC.

You can also use Azure Resource Manager (ARM) templates or other third-party tools (such as Ansible) to automate your infrastructure. ARM has better support for the latest Azure changes, but Terraform stands out with its cloud independence and simplicity.

Terraform comes with a single executable file. You can download and execute it:

`www.terraform.io/downloads.html`

Note Put Terraform into your PATH environment variable so that you can reach it from your project folder.

Creating a Single Player Game

After you install the development tools, you can start by creating your game. The intention here is to keep the game frontend as simple as possible. I used a freely downloadable Unity Asset, which includes some 3D characters. I use the Unity Asset called RPG Monster Dui PBR Polyart for the demonstration, because I found the characters hilarious. You can, of course, use your own game and apply the concepts described in this book.

First Steps to Building Your Game

If you choose to use the Unity Asset, here are the detailed steps to follow:

1. Create a new 3D Core project, called `MyGame`, with the help of the Unity Hub, as shown in Figure 1-2.

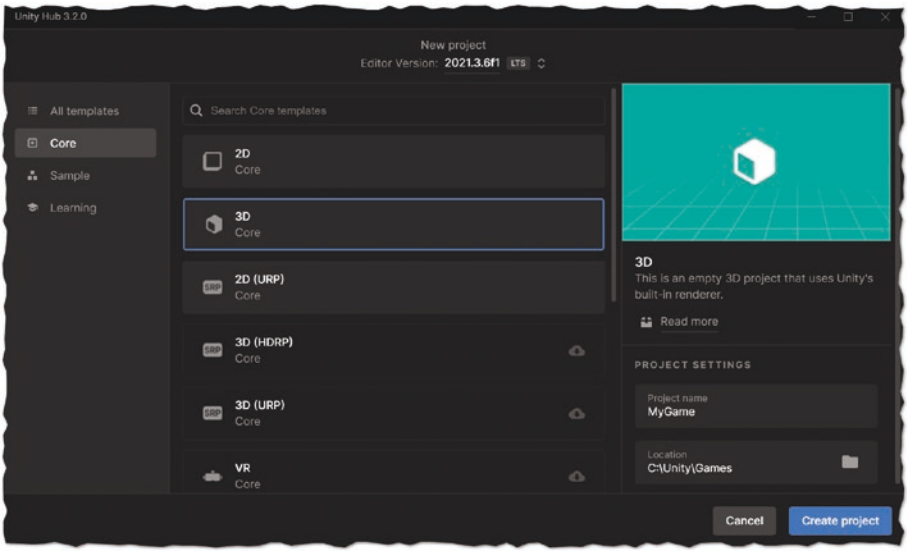

Figure 1-2. *Creating a new Unity project*

2. Go to the Unity Asset Store (choose Unity Editor ➤ Window ➤ Asset Store) and search for Unity Asset RPG Monster Duo PBR Polyart from this link: https://assetstore.unity.com/packages/3d/characters/creatures/rpg-monster-duo-pbr-polyart-157762.
3. Choose Add to My Assets ➤ Open in Unity ➤ Package Manager ➤ Import.
4. After you import the game into Unity, go to the sample scene (choose Project Panel ➤ Assets ➤ Scenes) and double-click it.

You can add a *plane* to the scene (right-click to choose Hierarchy Window ➤ 3D Object ➤ Plane) to be the floor where the players will move.

Go to a character prefab (choose Project Panel ➤ Assets ➤ RPG Monster Duo PBR Polyart ➤ Prefabs ➤ Slime) and select it. In the Inspector window, add a Rigidbody and a Box Collider component to it. You should make sure the collider fits with the character (0.5 on the Y value should be fine for this character).

Control and Animation of the Character

Create a new folder of scripts and add a new C# script called `Controller.cs`. You will put all the required logic to move and animate the character in this script. Copy the following code into it:

```
using UnityEngine;

public class Controller : MonoBehaviour
{
    private float speed = 0.01f;
    Animator animator;
    void Start()
    {
        animator = GetComponent<Animator>();
    }
    void Update()
    {
        Vector3 movement = new Vector3(Input.
        GetAxis("Horizontal") * speed, 0, Input.
        GetAxis("Vertical") * speed);
        transform.position = transform.position + movement;

        if ((movement.x != 0) || (movement.y !=0) ||
        (movement.z !=0))
        {
```

```
            transform.rotation = Quaternion.
            RotateTowards(transform.rotation, Quaternion.
            LookRotation(movement), 10000 * Time.deltaTime);
            animator.SetBool("moving", true);
        } else
        {
            animator.SetBool("moving", false);
        }
    }
}
```

Add this script to your `Slime` prefab as a new component.

The `Controller` script simply moves the character by reading the changes of the input and turns it in the right direction. It sets the `moving` parameter if the character moves.

To use this, create new transitions between Idle Normal and RunFWD in the Animator panel (choose Window ➤ Animation ➤ Animator) of the prefab. Right-click IdleNormal, click Make Transition, and choose RunFWD. Now click the transition arrow.

Then, go to the Parameters tab of the Animator and add the moving parameter as bool. After that, you can add the condition for the state transition. If the moving parameter is true, the character should start to run. You can see this configuration in Figure 1-3, after you drag-and-drop your character prefab into the Hierarchy window. See if you can control it.

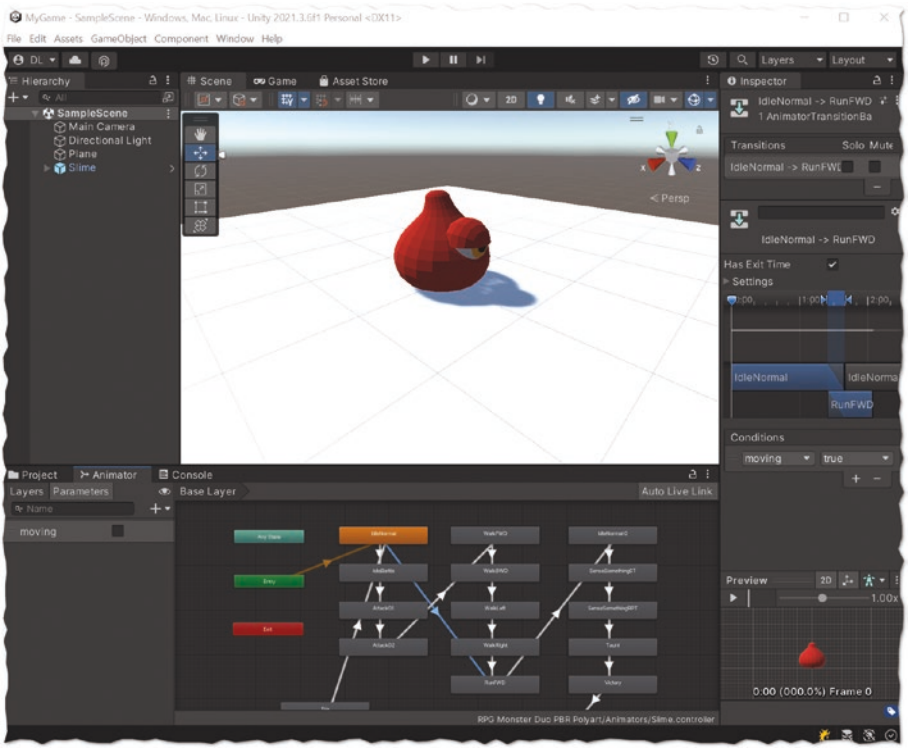

Figure 1-3. *Simple game in Unity*

You have now a very simple single player game, which you will extend in the following sections of this book.

Creating a Menu for Your Game

Now you'll create a very simple menu, so your players can choose what they want to do. In this book, you implement everything twice, once in PlayFab and once with Azure, and the menu should trigger both implementations.

Use the `GUILayout` class to implement a minimalistic menu. Of course, you should build a fancier menu for your game.

First, create two new game objects, `PlayFab` and `Azure`. We add all the scripts under the related game objects in this book.

Create another game object, called `ControlPanel`, and a script called `ControlPanel.cs`. Add this script to the `ControlPanel` game object.

The following code shows one possible structure. You can download the complete code from GitHub. We also extend this code in the following chapters with more features.

This allows you to start functions by clicking the buttons. It also makes it possible to open a new window and assign multiple buttons or other input elements.

```
using UnityEngine;

public class ControlPanel : MonoBehaviour
{
    public const int ROOTMENU = 0;
    public const int PLAYFAB_LOGIN = 1;
    public const int AZURE_LOGIN = 2;
    // TBD: Additional features

    int selection;

    GameObject playFab;
    GameObject azure;

    private void Start()
    {
        playFab = GameObject.Find("PlayFab");
        azure = GameObject.Find("Azure");
    }

    void OnGUI()
    {
        if (selection == ROOTMENU)
```

```
        GUILayout.Window(0, new Rect(0, 0, 300, 0),
        OptionsWindow, "Options");

    if (selection == PLAYFAB_LOGIN)
        GUILayout.Window(0, new Rect(0, 0, 300, 0),
        LoginWithPlayFabWindow, "Login with PlayFab");

    // TBD: Additional features
}

void OptionsWindow(int windowID)
{
    if (GUILayout.Button("Login with PlayFab"))
        // TBD: Implement actions to buttons

    if (GUILayout.Button("Login with Azure"))
        selection = AZURE_LOGIN;

    GUILayout.Space(10);

    // TBD: Additional buttons
}

void LoginWithPlayFabWindow(int windowID)
{
    GUILayout.Label("Display name:");
    // TBD: Implement each sub-window
}

void LoginWithAzureWindow(int windowID)
{

    if (GUILayout.Button("Login with Azure"))
        // TBD: Implement actions to buttons, such as
        // azure.GetComponent<AzureAuth>().
           LoginWithAzure();
```

```
            if (GUILayout.Button("Cancel"))
                selection = ROOTMENU;
    }
}
```

Now you have a simple single player game with a menu that allows you to invoke different backend functionalities. You can move on to the backend.

Creating a Local Multiplayer Game

Before you learn how to implement an online multiplayer game, we investigate how to implement a local multiplayer game. You will see it is much easier, because the server is one of the local machines. Later, you will extend this solution to a dedicated game server in the cloud, so understanding these basic components is important for the following chapters.

What Is a Local Multiplayer Game?

In this context, "local" means all the players connect to the same local network, for example, all are on your home network. These machines are not accessible from outside of your home, through the Internet.

Figure 1-4 shows four players with their clients. All the clients are connecting to a local network. Player 1 hosts the game. Under the hood, on Player 1's machine, besides the client, there is also a server.

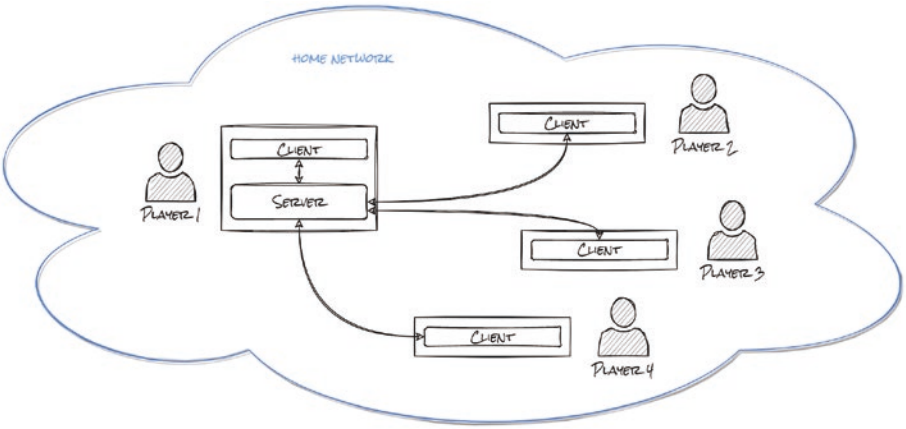

Figure 1-4. *Local multiplayer game*

Getting Started with Mirror Networking

Mirror is an open source networking library for Unity. It ran originally as the own networking API for Unity under the name of UNET, but after Unity deprecated it, the community developed it further under the name Mirror Networking. It is free, which is an advantage over its opponents.

You can get Mirror from the Asset Store (choose Unity Editor ➤ Window ➤ Asset Store, and search for "Mirror"). Then choose Add to My Assets ➤ Open in Unity ➤ Import. You can uncheck the Examples folder, as you will use your own game.

If having trouble finding Mirror, try this link: `https://assetstore.unity.com/packages/tools/network/mirror-129321`.

Be aware that Mirror keeps on developing quickly, so you might find deprecated or nonexistent functions. You may need to make some changes in your code if you use later versions of Mirror. This book uses version 66.0.9.

After you import Mirror into Unity, add its key components to your game. With that, you have enabled networking functionality in your game. You will need at least Mirror's core components—the `NetworkManager`

and the NetworkIdentity assigned to objects to be synced on the network. Mirror contains some additional components as well, such as NetworkDiscovery, which helps to find servers on the local network. In this book, we focus on the key components that you will need to implement the backend features.

Figure 1-5 summarizes the mirror components for the local multiplayer game. Be aware that certain components, such as the NetworkAnimator, NetworkIdentity, and the NetworkTransform, have to be assigned to your game object that you want to sync with other players on the network.

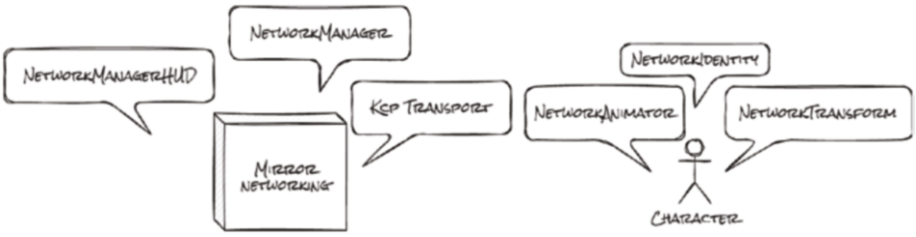

Figure 1-5. *Mirror components*

The Core Components: NetworkManager and NetworkIdentity

Add a new game object called NetworkManager to your hierarchy in Unity Editor. Then add the NetworkManager component to it. This is a core component of your multiplayer game. It is responsible, for example, for spawning (creating) new game objects. For example, putting a new player on the scene when the player joins to the game.

When you add the NetworkManager, Mirror will also add the KcpTransport component to your game object. You can change this to other transport protocols like TCP or UDP. Mirror uses KCP by default and claims that it is faster and reliable than the others.

Make sure that you remove the `Smile` game object from the hierarchy. We will work with the prefab (choose Assets ➤ RPG Monster Duo PBR Polyart ➤ Prefabs ➤ Slime).

Add a `NetworkIdentity` component to it. Mirror will only know game objects that have the `NetworkIdentity` component assigned to them.

Drag-and-drop your character prefab to the `NetworkManager` component's `Player` prefab. From now on, Mirror will spawn the player automatically when a client starts.

Synchronize Moving and Animation

You also need to add the `NetworkTransform` component to the character, as you want to synchronize the transform parameters (position, rotation, scale) over the network, so that other players can also see the changes in their local machine.

Check the Client Authority checkbox in the `NetworkTransform` component. By default, the server has authority over all game objects, but if you give authority to the client on a certain game object, the client itself will send the movement information to other clients. Be aware that you should always add these networking components to the prefab and not to the game object in the hierarchy.

Mirror does not synchronize the animation between the two clients by default. The characters belonging to remote clients are just sliding without animation on the scene.

To resolve this issue, go to the prefab and add a `NetworkAnimator` component to it. Check the Client Authority checkbox. You also have to assign the animator that will sync the animation over the network. To achieve this, drag-and-drop the prefab to the `NetworkAnimator`'s `Animator` property.

Network Manager Heads-up Display

I advise you to add the `NetworkManagerHUD` (heads-up display) to the `NetworkManager`. You can disable the `ControlPanel` game object for now, as it can overlap the heads-up display.

The `NetworkManagerHUD` will generate UI buttons that allow you to become the host (both server and client at the same time), only the server, or be the client and join a server IP.

After you finish adding and configuring Mirror, you can start the game. Notice that your character is not visible anymore. You can choose to be the host on the `NetworkManagerHUD`, whereby you would start the server and client, and Mirror will spawn the character to the scene. In this case, your client joins the local server on your own machine.

Setting Up Unity for Multiplayer Games

To develop games that are used by multiple players, you need to have multiple Unity instances running locally. Unfortunately, you cannot simply run multiple Unity Editor on one project.

As a workaround, you can use some tools (such as SyncToy) or Unity add-ons to clone your project folder to another folder regularly, after every change.

Alternatively, you can apply *symbolic links,* which are pointers to the original files and folders, for the second Unity Editor instance.

To configure symbolic links, follow these steps:

1. In Windows, start the Command Prompt as Administrator (search for `cmd`, right-click it, and choose Run as Administrator). In MacOS, start a Terminal window.

2. Create a clone folder:

   ```
   mkdir MyGame-Clone
   ```

3. In Windows, create the symbolic links by using the `mklink` command on the `Assets`, `Packages`, and `ProjectSettings` folders (in macOS, use the `ln -s` command instead):

```
mklink /J C:\Unity\Games\MyGame-Clone\Assets
C:\Unity\Games\MyGame\Assets
mklink /J C:\Unity\Games\MyGame-Clone\
Packages C:\Unity\Games\MyGame\Packages
mklink /J C:\Unity\Games\MyGame-Clone\
ProjectSettings C:\Unity\Games\MyGame\
ProjectSettings
```

Note After you clone your project, you will want to avoid making changes to the clone project instance. Only work on the original project.

In the Unity hub, add the new cloned folder, select the same Unity version as the original one, and load the project. For the cloned version, you have to open the scene in Unity Editor.

Starting Your First Multiplayer Game

In the cloned Unity Editor, you can reload the project (a popup window will always inform you about changes in the original project) and start it. If you click client with localhost IP (since your server is also on the same machine), it will join the server and you should see the two players in one game. Congratulations, you made your first multiplayer game! You still have a major issue to fix.

Controlling Only Your Character

If you try to move your character, you notice the inputs apply to all players. This is because, with each new joining player, Mirror clones the character prefab where the same controller scripts will run. You need to somehow restrict the application of the keys to the local player.

You can achieve this by using the `hasAuthority` variable in the `NetworkBehaviour` class. This class is inherited from `MonoBehaviour`, so you can change `MonoBehaviour` to `NetworkBehaviour` in your `Controller` class.

Change the `Controller.cs` script using the `hasAuthority` condition. This will cause Unity to only consider the input for the game objects on which the player has authority:

```
using Mirror;

public class Controller : NetworkBehaviour
{
   ...
    void Update()
    {
        if (!hasAuthority) return;
...
```

Now you have a simple local multiplayer game. You will learn how to extend this to an online multiplayer game in the coming chapters. You can experiment further with Mirror Networking and its features. It is important to optimize the parameters of Mirror to your current game requirements in order to deliver a great game experience to your players.

Getting Started with PlayFab

Before creating your first online multiplayer game, you need to access the cloud. If you don't have a Microsoft account, you need to create one. You will use it throughout this book:

https://signup.live.com/

You can use this account to:

- **Sign in to PlayFab**. You can use up to ten titles in development mode, with up to 100,000 users per title for free.
- **Sign in to Azure.** If you are a first-time Azure user, you will get 12 months free usage of a couple of services. This includes 750 hours of B1s virtual machines, or 750 hours of a PostgreSQL database.

Create your PlayFab account now. Simply go to https://playfab.com/ and use your Microsoft account to create a PlayFab account. (Choose Sign in with Microsoft.)

In PlayFab, enter your contact information and set your studio name. By default, it is "My Studio Name, and you will get a default title called "My Game." You can change this name later and add studios and titles to your account.

Note PlayFab currently does not support deleting studios. So if you create one, it will be always there.

Import PlayFab SDK in Unity

The next step is to download the PlayFab Software Development Kit (SDK) and import it into Unity:

https://aka.ms/playfabunitysdkdownload

Note This link does not go to a download page, but directly downloads the `unitypackage` file. If you have trouble importing the package into Unity, visit this site for information: https://docs.unity3d.com/Manual/AssetPackagesImport.html

You will find a new folder in your Project window called `PlayFabSDK`.

You can use the PlayFab Unity Editor extension, download it, and import it into Unity. However, this is optional. It helps you download and upgrade the SDK and configure your title automatically. It brings you some convenience, but you can integrate Unity with PlayFab without the extension, so you will not use it in the next steps. You can find the extension here:

https://aka.ms/PlayFabUnityEdEx

Configure PlayFab in Unity

Now that you have both PlayFab and Unity set up, let them "know" each other. You need to add your actual title and the developer secret key to the PlayFab configuration in Unity.

For that, go to Assets ➤ PlayFabSDK ➤ Shared ➤ Public ➤ Resources and double-click `PlayFabSharedSettings.asset` in the Project window. In the Inspector window, set the following configurations:

- **Title ID:** Go to the Game Manager in PlayFab and click the gearwheel next to your title's name. Then select My Studios and Titles. You can see the ID of your game (don't mix it with the Title name).
- **Developer Secret Key:** You can find this in the Game Manager. Choose the gearwheel next to your title's name then choose Title Settings ➤ Secret Keys. Copy and paste the secret key into the respective field in the Inspector.

You can leave the other settings at their defaults. You should now be able to communicate with PlayFab from Unity.

Signing Up for Azure

You can use a Microsoft account to sign up for Azure:

https://azure.microsoft.com/

If this is the first time you've used Azure, you will get services free for 12 months and an additional $200 credit to consume any services in the first month. Still, always be careful when using services and check their prices. Always remove unused resources, and if possible, use free tier resources for development and only pay for productive resources.

Once you have signed up, you can look around in the Azure Portal.

https://portal.azure.com/

Note You can check your expenses in the Azure Portal (https://portal.azure.com), under Cost Management + Billing. Azure gives you also a forecast about your spending. You can set up cost alerts, so that Azure informs you when you reach a certain threshold.

After you have access to the Azure Portal, download and install the Azure Command-Line Interface (CLI):

https://docs.microsoft.com/en-us/cli/azure/install-azure-cli

Then, try it from the command prompt:

```
az login
```

Later on, you will use Azure Kubernetes Service (AKS). For that, you should also install the CLI, as follows:

```
az aks install-cli
```

Creating Your First Online Multiplayer Server

In this section, you further extend the earlier local multiplayer game. As shown in Figure 1-6, you will put the server in the cloud, allowing players to play together even when they are not in the same location.

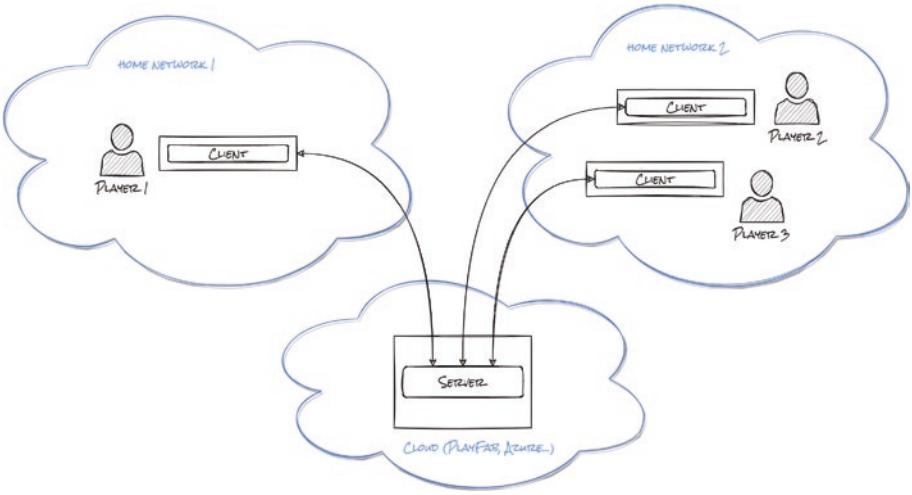

Figure 1-6. *Multiplayer server in the cloud*

Practically, you have to deploy a virtual machine in Azure and install a server version of the game on it. This server will listen on a port, waiting for clients to connect.

Building Your Infrastructure

Create a file (such as `provider.tf`) where you define the provider. Terraform officially supports multiple cloud providers; in this case, let's add Azure:

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=3.13.0"
    }
  }
}

provider "azurerm" {
  features {}
}
```

First create a resource group in Azure that allows the logical grouping of resources. The resource blocks are the core elements in Terraform. You define each resource that Terraform deploys in the cloud. Create a new file called `resourcegroup.tf` and add the following code to it:

```
resource "azurerm_resource_group" "this" {
  name     = "mygame-rg"
  location = "West Europe"
}
```

The resource block declares the following:

- **Resource type:** In this case `azurerm_resource_group` refers to a resource group in Azure.
- **Resource name:** In this case `this` is a naming convention of Terraform. If no other resource exists with this type, use `this`. The name allows you to refer to this resource within the module.
- **Arguments:** Within the { } block, you can define the parameters of the resource. Here, it's the name and the location of the resource group.

Note The location of the resource group does not imply the location of the contained resources in that resource group. It is only an indicator that resources in this resource group are in that specific region.

Now you'll define the network-related resources in another file. Create a file called `network.tf` and add the code that describes a virtual network. We refer back to the name and location of the resource group that we described earlier:

```
resource "azurerm_virtual_network" "this" {
  name                = "mygame-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.this.location
  resource_group_name = azurerm_resource_group.this.name
}
```

Within that virtual network, add a subnet:

```
resource "azurerm_subnet" "internal" {
  name                 = "internal"
  resource_group_name  = azurerm_resource_group.this.name
  virtual_network_name = azurerm_virtual_network.this.name
  address_prefixes     = ["10.0.2.0/24"]
}
```

You also need a public IP to access the server through the Internet. Create a network interface and assign it to this public IP:

```
resource "azurerm_public_ip" "this" {
  name                = "mygame-pip"
  resource_group_name = azurerm_resource_group.this.name
```

```
  location            = azurerm_resource_group.this.location
  allocation_method   = "Dynamic"
}

resource "azurerm_network_interface" "this" {
  name                = "mygametitle-nic"
  location            = azurerm_resource_group.this.location
  resource_group_name = azurerm_resource_group.this.name

  ip_configuration {
    name                          = "mygame-publicip"
    subnet_id                     = azurerm_subnet.this.id
    public_ip_address_id          = azurerm_public_ip.this.id
    private_ip_address_allocation = "Dynamic"
  }
}
```

And finally, the virtual machine. You definitely don't want to have the admin password directly in your Terraform code, but to demonstrate the workings, leave it like that for now. Note that the B1s virtual machine instance is free for the first 12 months, so it is optimal for experimentation. Create a new file called `vm.tf` and copy the following code:

```
resource "azurerm_linux_virtual_machine" "this" {
  name                = "mygame-vm"
  resource_group_name = azurerm_resource_group.this.name
  location            = azurerm_resource_group.this.location
  size                = "Standard_B1s"
  admin_username      = "mygamevmuser"
  admin_password      = "njkwer43GFBS@#"
  disable_password_authentication = false
```

```
  network_interface_ids = [
    azurerm_network_interface.this.id,
  ]

  os_disk {
    caching              = "ReadWrite"
    storage_account_type = "Standard_LRS"
  }

  source_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }
}
```

These Terraform codes will deploy the defined infrastructure components. By default, the state file is on the local machine. If you work in teams on the Terraform code, it is better to store this state file in the cloud, in this case, in Azure Storage. It is also a safe way to store the state file of your infrastructure, even if it has some cost.

The state file always contains the actual resources and their parameters in the cloud. If you delete the virtual machine manually through the Azure Portal, the next "Terraform apply" will deploy it again, exactly the same way it was earlier. If you remove the virtual machine resource from the Terraform script, the next "Terraform apply" will remove the virtual machine from Azure.

When you finish your script and start Terraform, initialize the Terraform configuration in the current working directory:

```
terraform init
```

Then run your script with the following command and make sure it executes without any error messages:

```
terraform apply
```

You may notice that Terraform hangs and never finishes the execution. You can enable logging for troubleshooting with the following commands:

```
set TF_LOG_PATH=terraform.log
set TF_LOG=TRACE
```

Note On macOS, use the EXPORT command to set environment variables.

Later, if you want to remove the virtual machine and all other related resources you created with Terraform, simply execute the following:

```
terraform destroy
```

Creating the Game Server Instance

In Unity, go to File ➤ Build Settings… Select Linux as the target platform and then select the Server Build checkbox. Click the Build button, create a folder, and let Unity generate the server code for you.

Retrieve the public IP address of your virtual machine:

```
az network public-ip show -g singlevm-resources -n
mygame-pip --query "ipAddress"
```

Upload the server files to the virtual machine:

```
scp -r <server-build-folder> mygamevmuser@<publicIP>:/home/
mygamevmuser
```

Execute the server:

```
ssh mygamevmuser@<publicIP>
chmod 755 mygame-server.x86_64
./mygame-server.x86_64
```

Testing and Next Steps

Going back to Unity, start the game and, on the heads-up display, change the IP address from the localhost to the public IP of your virtual machine in the cloud. Your client should connect to the server, and Mirror should spawn the character on the screen. You can try starting your second Unity Editor with the cloned project and see if both can join the server. All should work fine.

With that, you have created the simplest online multiplayer game. It uses a dedicated server hosted in the cloud. This single virtual machine is quite limited and not really usable in a real environment. It does not scale at all, so after more players join, the server will run out of resources. Also, it does not support multiple sessions. We address these issues in the following chapters.

Summary

In this chapter, you learned about the tools and technologies you will use in this book. You created a simple game to showcase the backend features. You learned about the basic functionality of Mirror Networking to create local multiplayer games. With the help of a dedicated server deployed in the cloud, you learned the basic idea of how to implement an online multiplayer game that allows players to play together from any distance. The next chapters extend this concept further and show you how to build a more robust and scalable solution.

Review Questions

1. How do you implement and test a multiplayer game with the help of Unity?
2. What is Infrastructure-as-Code (IaC) and which tools can you use?
3. How do you configure PlayFab in Unity?
4. What is the difference between a local and an online (non-local) multiplayer game?
5. How does Mirror Networking extend the Unity Engine?
6. What are the main components of Mirror and what are their functions?
7. How do you distinguish between local and remote players in Mirror? Why is this necessary?
8. Which resources are required to have a virtual machine in Azure?
9. Which command do you use to execute your Terraform scripts?
10. How do you make a server build of a game in Unity?