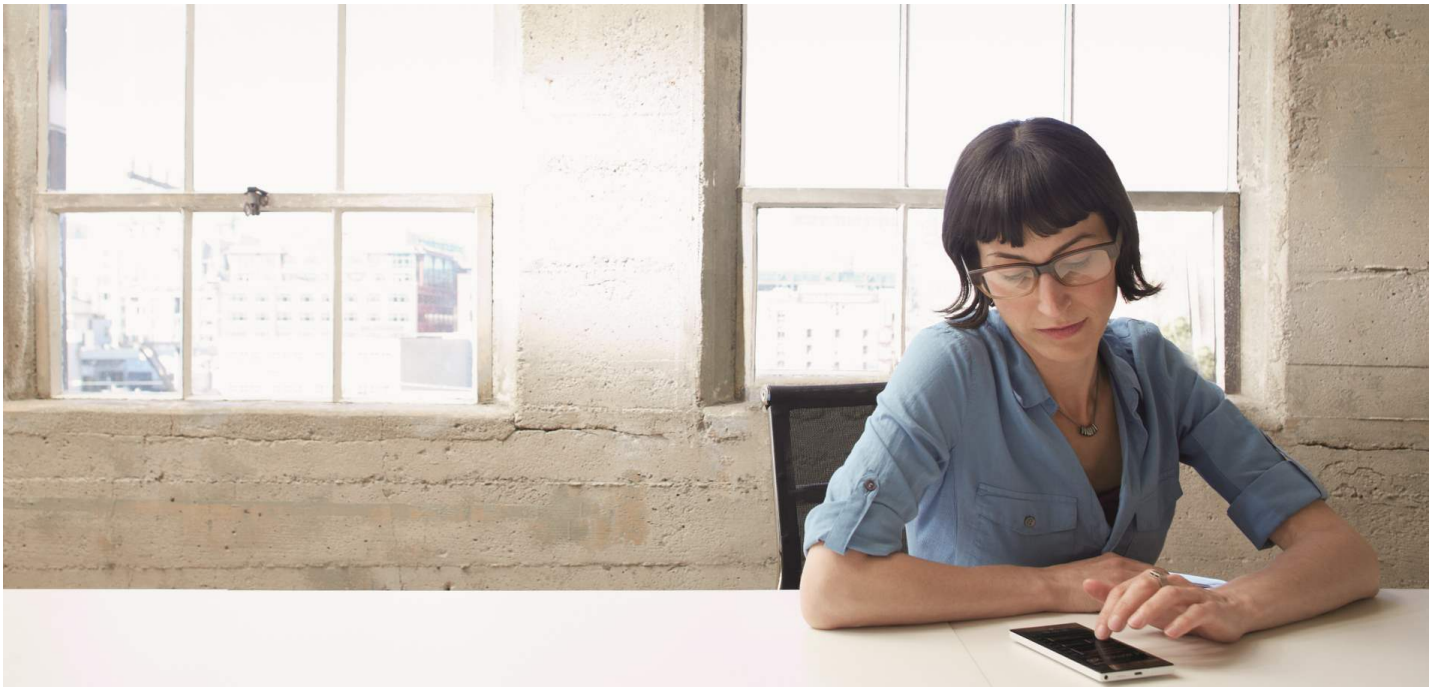
Digamore Entertainment scores with a new gaming platform based on Azure Service Fabric

April 26, 2018



Authored by Marko Flod from Digamore, in conjunction with Mark Fussell and Ed Price from Microsoft.

This is part of a series (https://blogs.msdn.microsoft.com/azureservicefabric/tag/case-study/) on customers where we look at why they chose Azure Service Fabric (https://azure.microsoft.com/en-us/services/service-fabric/) , and dive deeper into the design of their application, particularly from a microservices perspective. In this post, we profile Digamore, their Football Empire mobile game running on Azure, and how they designed the architecture using Service Fabric.

Digamore Entertainment GmbH (http://www.digamore.com/) is a game-development company that took on the goal of developing *Football Empire* (http://www.footballempire.com/) , a free-to-play massively multiplayer online game.

Football Empire, which is being developed for Android and iOS mobile platforms, combines the building game genre with strategy elements. Users manage a team of football players, as well as their club grounds

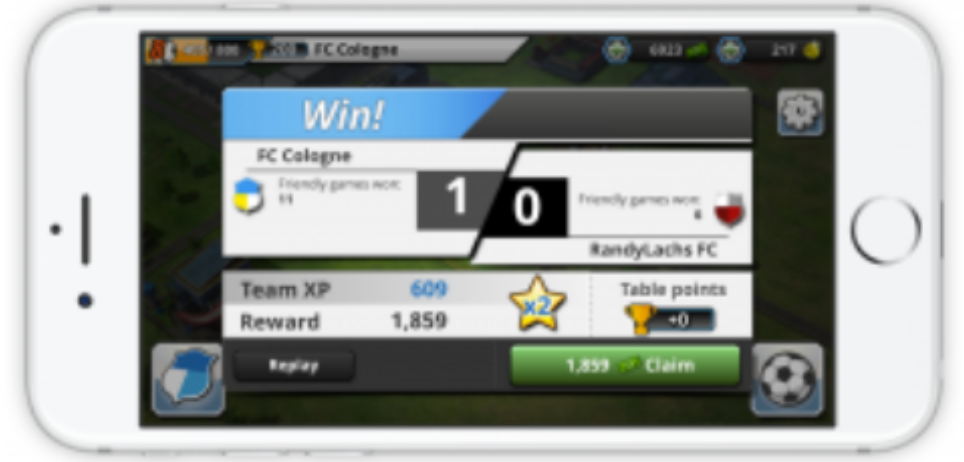
and facilities. It features a small economy of resources and items that can be acquired through micro-transactions. The game's objective is to progress by beating more advanced opponents during football matches. This is accomplished by expanding the club grounds and improving the abilities of the players on your team. Below are some screenshots from the game to give you an idea of how it is played.

> "Though we are a small team and have no one dedicated to DevOps, we were able to develop a system that can handle the requirements of a massively multiplayer online game."

—Marko Flod: Software Engineer
Digamore Entertainment

Starting Football Empire

The club grounds, where you can build and upgrade your facilities



The results screen, after winning a game

New approach to gaming using Service Fabric and microservices

When the project was initially started, the question was raised, what kind of backend should we develop? Fortunately, we had experience to build on, having previously developed a backend for a football manager game that was designed for web browsers. Our investigation revealed that architecture had been developed vertically with a monolithic structure. It was not possible to scale specific parts of the application, which created a bottleneck to the database. Thus, costs would grow much higher as more players accessed the game. Since we wanted to create an application that would attract 100K to 1M users, our application had to be scalable with the goal of being cost effective.

We also had to come to terms with the rapidly growing mobile gaming market that had lifted user expectations. A slow, unresponsive or unstable backend architecture would likely lead to a huge user loss. The service therefore had to be cheap to run, deal with unpredictable scale, be highly resilient, and very responsive in terms of latency in the requests.

We decided to develop with microservices even though we were aware that it increases the complexity of the software and requires more discipline during development. In the Service Fabric Reliable Actor programming model, we finally found our solution. We saw the benefit that each actor can hold its own data and its operations can be spread out. We found a couple of frameworks that support the Actor model, but after our comparison, the combination of Service Fabric and Azure convinced us this was the best choice. In this article, we'll describe which key features that we use and why we chose Service Fabric.

Advantages of using Service Fabric

After reviewing Service Fabric, with our game design requirements in mind, we identified the following advantages:

- **Self-Healing:** Since we're a small team and don't have the resources for a dedicated DevOps strategy yet, we wanted a system that could repair and heal itself. If any node failed, another one would just take its

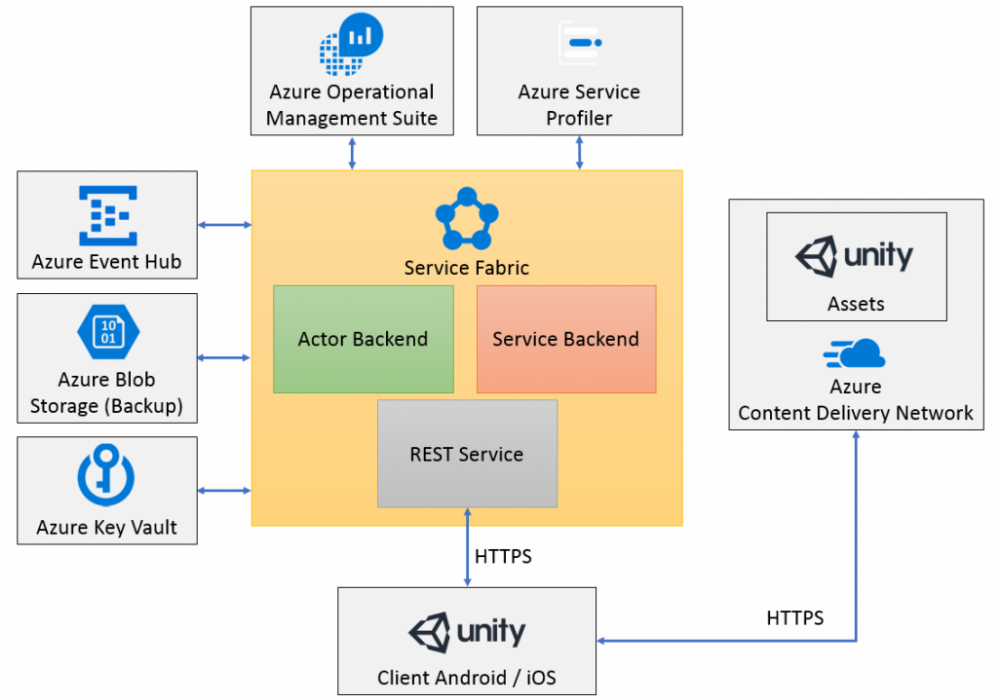
place, and all the services running on the failed node would be redistributed to the remaining nodes. This is crucial to the connection stability of the game.

- **Horizontal Scaling:** Instead of adding more power to single machines (vertical scaling), we can now add more machines to our cluster to keep our system reliable. With vertical scaling, we would reach a limit with a specific amount of users, which is a limit that we're much less likely to run into with horizontal scaling.
- **Automatic Scaling:** The system's scale is configurable, if established metrics help you identify that the current load is too large. A high peak of users could occur, for example, during a commercial break, or at the end of a real football match, and even at these peak load times the system should remain available.
- **Live Cluster Status:** The Service Fabric Explorer is a useful tool to see the status of the cluster. We could monitor the upgrade process while maintaining the possibility of reverting an upgrade, if we noticed any unexpected behavior.
- **Upgrade Management:** Multiplayer games constantly change and grow their content to keep up with the market, this may result in longer wait times for users during upgrades on the backend. Service Fabric offers a good solution, as the backend remains available for the players through the upgrade domain. The upgrade process is customizable and fully automated. A mechanism also distinguishes between code and configuration file changes. It is a feature we really appreciate. It results in faster updates of the backend without the need to recompile because configuration is separate. It is a huge advantage for game-balancing changes, since they don't affect the code.
- **Actor Programming Model:** The logic and design of the game works very well with the actor model, as game entities and logic operations can be easily represented by it. The actor programming model supports the constraints of the project, and it greatly increases the effects of the microservices approach.
- **Profiling:** Analyzing a cluster environment can be more difficult than a desktop application. With the help of Azure Service Profiler (https://www.azureserviceprofiler.com/) and Azure Operations Management Suite (https://www.microsoft.com/en-us/cloud-platform/operations-management-suite) (OMS), we can investigate our application performance while receiving real time alerts if anything unusual should happen. Furthermore, the integration of Service Profiler and OMS was very easy and user friendly.
- **Testing and Bug fixing**: Thanks to Service Fabric's ability to debug code on local development machines, we can easily reproduce and fix bugs. We saw the opportunity to include unit and integration testing, which are essential for stable products. Cheating has always been an issue in multiplayer gaming and should be treated with care. We wrote a lot of tests to ensure that any hacking or exploit possibilities weren't exposed for users.

Architecture (logical) of Azure Services used

Using only a few Azure services, we were able to get a system for a massively multiplayer game up and running. Many of the Azure services required for Service Fabric are automatically created with its default template: The diagram below shows the Azure services used for the Football Empire game.

> "Upgrade management of Service Fabric provides us a fast and user-friendly way to update small game-balancing changes."
>
> —Marko Flod: Software Engineer
> Digamore Entertainment



Overview of the Azure services

- **Azure Operational Management Suite:** We use OMS to analyze and visualize our logs. The alert function is very important for us, as we can receive instant messages for issues (for example, an actor throwing exceptions) and react immediately.
- **Azure Service Profiler:** With the profiler, we can evaluate the communication between actor operations and identify high call wait times.
- **Azure Content Delivery Network** (https://azure.microsoft.com/en-us/services/cdn/) : (https://azure.microsoft.com/en-us/services/cdn/) Since Google Play and the App Store don't allow big install files, the client has to download the content during the application start up. The Azure Content Delivery Network also enables

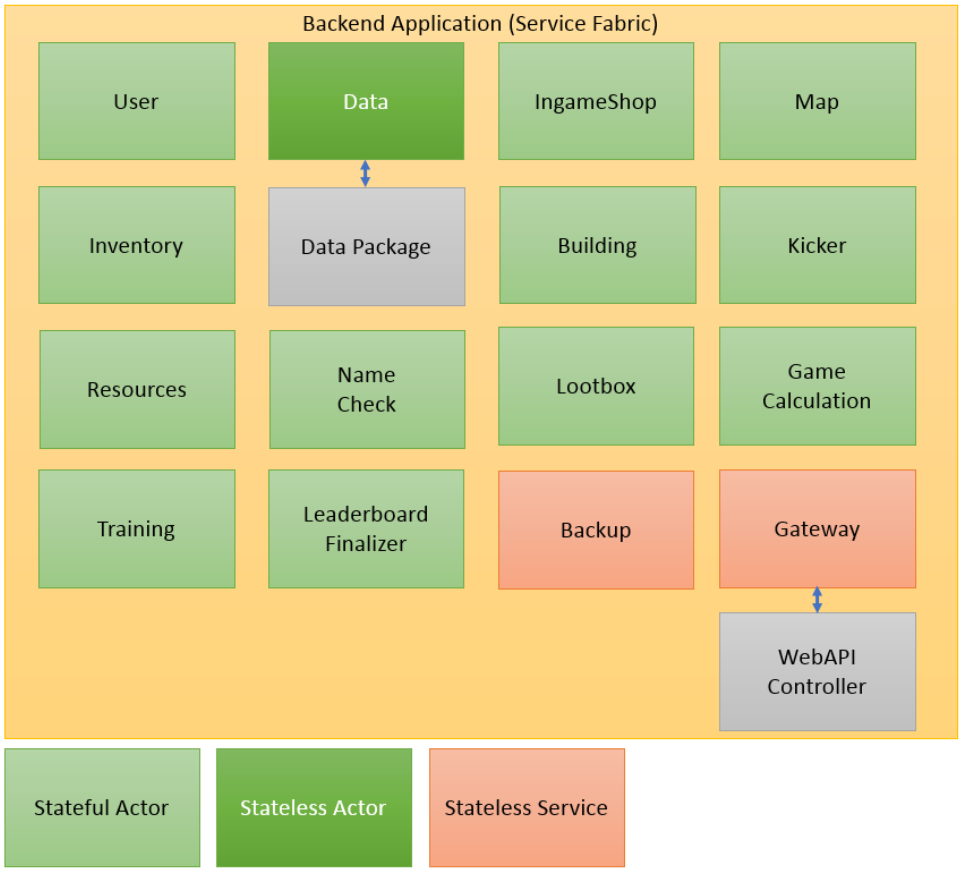
us to update the content of the application without having to resubmit it, and without forcing the user to go to the store to update it.

- **Azure Event Hub** (https://azure.microsoft.com/en-us/services/event-hubs/) : (https://azure.microsoft.com/en-us/services/event-hubs/) When backing up the state of an actor, an event is sent to the event hub. Service Fabric provides the ability to write backup data to Event Hub for an actor and save the state of the actor to blob storage.
- **Azure Blob Storage:** (https://azure.microsoft.com/en-us/services/storage/blobs/) We use the blob storage to save the actor states for backup and restore capabilities.
- **Azure Key Vault:** (https://azure.microsoft.com/en-us/services/key-vault/) The vault is used for storing connection strings and other secret information, such as certificate credentials.

Architecture using Service Fabric actors and reliable services

Currently, our needs are satisfied with a single Service Fabric application. The core of the Service Fabric application contains the stateful and stateless actors, stateless services, and stateful reliable services. If you look at the architecture overview below, you can immediately recognize the game's design logic. It is a huge benefit for us to have the game logic portrayed in the overview of our application, as it makes understanding the code much easier, which leads to easier maintenance and flexibility. This was also one of the major reasons we decided to use the actor model.

> " The documentation, video talks, and Yammer groups provide our developers a fast way to understand the Service Fabric framework in a practical way."
>
> —Marko Flod: Software Engineer
> Digamore Entertainment



Overview of the actors and services in the application

The yellow box (in the diagram above) is the application type, which has an instance running in Service Fabric.

Besides the benefits already mentioned, we also gained considerable scalability. Most of the stateful actors are coupled to a specific user. This gives us the capability where the partitions of the different actors can be easily spread out on the different clusters, if we add more nodes to our cluster. Another advantage is scaling out explicit actor operations, which can be done simultaneously. After a match, for example, we want to change the attributes of all the participating football players, which can be done simultaneously instead of handling each football player individually.

Key Service Fabric services

The following is a list of the services and actors created in the game:

- User Actor: This is the key actor for every user. It contains basic information about the current user state and which football players they have in their team.
- Data Actor: In our game, we have certain simple values stored in a JSON file that controls the game configuration (the of the game). We can control how much money a player receives for a match or how many games a football player has to play in to get stronger. The game logic use these values. Therefore, almost every actor needs those values. With the help of the Data actor, we can access the JSON file that is contained in the and provide the values for the other actors. This actor doesn't save its state, as the JSON file is taken from the data package.
- IngameShop Actor: Some items can only be bought in limited amounts. Thus, transactions have to be saved for the actors to check.
- Map Actor: The player has a club ground, where he can build facilities and move them around. Its representation is done in the Map actor that checks the validity of a specific location to place a building on.
- Inventory Actor: All of the user's items are stored in this actor, and provide operations to check if the amount of a specific item type is available.
- Building Actor: Each building placed on the club ground has a unique identifier stored in this actor. Buildings are upgradable, so we had to trigger the logic on a requested upgrade by the player.
- Kicker Actor: This is a representation of a single football player. It contains all relevant information about the visual appearance, current strength, and abilities. Instead of letting the user actor handle the information on the football players, . Previously, the creation of a new football player could take up to one second for each player. Distributing the generation relay helped greatly with our response times!
- Resources Actor: Resources are the trade goods of the game (soft and hard currency). Information regarding amounts is stored in this actor.
- Name Check Actor: As users can freely choose a name for their football club, a check of existing user names has to be conducted, which can be a crucial operation. Each existing name has its own representative actor. This way we can easily check if a name is available or not.
- Lootbox Actor: As Lootbox features have grown popular in the mobile market, it goes without saying that we wanted to implement one. A Lootbox is like a random gift box for the player. It can contain several items, including soft or hard currency. The amount and variety of items received from the Lootbox depends on the user's state. Emptied Lootboxes are charged after a certain waiting time, thus its state must be persisted.
- Game Calculation Actor: This actor contains the core calculation of a football match within the game, and is created each time two players hold a match against each other. It also contains all information of the game itself, so players can (re-) watch the match on their devices.
- Training Actor: Each football player can undergo training to improve his abilities. But there are some limitations on the training, such as only being able to participate in one training session at a time. The user gets a notification when the training is completed and is then able to pick up the rewards.
- Leaderboard Finalizer Actor: Each player has an internal value (from the Elo rating system) that increases with his wins against other players. (The Elo rating system was created by Arpad Elo.) With the Elo system,

we can determine the strength of the player to find a corresponding opponent. After an amount of time, the players with the highest Elo rating are placed in a Top-3 table.

- Backup Service: We have created a backup service that pulls events from the Azure Event Hub. Each event contains information about which actor should be saved to Azure Blob Storage. Whenever an actor executes a crucial operation, like buying hard currency, we send an event to the Event Hub to request a backup of the current state.

  The backup is the data of the actor serialized into a JSON file, which is written to external Blob Storage. If the Blob Storage of the virtual machines goes into an unrepairable state and the data of the actors is no longer accessible, we can recover the data from the external Blob Storage.

- Gateway Service: We decided to implement a REST service for the client. Each request is routed to a specific WebAPI controller that accesses the different actors and services.

Why we chose Service Fabric

The integration of Service Fabric in Azure and on Windows machines was very intuitive. Each developer can easily develop and test new features on their own machines. We also decided to set up a test server with Service Fabric in our local environment for our testers, which succeeded very quickly. We embraced the simplicity of setting up the cluster and deploying our application. There is an incredible harmony between the Azure features and services (such as Event Hubs, Storage, Key Vault, OMS, Service Profiler, and so on) and Service Fabric, making them very easy to integrate together. By leveraging Service Fabric in the cloud, this led to less operational overhead for the development team, which validated our decision to use Azure Service Fabric.

The Service Fabric deployment tools provide us with an easy way to upgrade or downgrade our application on the cluster or locally. With only a few steps, we are able to roll out an upgrade on our live system or to just reset our local server when we want to reset the user data.

We also received a lot of support from Microsoft concerning Service Fabric and Azure. Microsoft provides a lot of documentation and video talks. Also, any issues reported in GitHub immediately receive responses. We also benefited from directly talking to Microsoft concerning specific technical questions.

With the help of the actor model, we could depict the state of a single user on a high level and allow its interaction with other users. By implementing game logic into single actors, we spread our operations and data on different nodes to make our application scalable. Another benefit is being able to only keep active users in memory—there are many users who may start the application once and never return—due to the virtual actor model. In these cases, we are glad that those players don't use any CPU cycles, just some local stored state in each actor.

We used actor reminders in Service Fabric to keep the leaderboard up to date. This was a very agreeable way to identify tasks that should be repeated after a specific time range.

Summary

With Service Fabric, we were able to create a very scalable backend architecture in a short period of time. Service Fabric gave us developers a straightforward way to implement the different game design requirements, without being too concerned about the infrastructure or spending a lot of time learning a framework.

Because we saved time with the help of Service Fabric, we could write more tests. That made every team member happier and gave us a better night of sleep. But this is only the beginning of our journey with Service Fabric, as we are provided with regular updates and features from the Service Fabric team. We get to benefit from the new features of Service Fabric on a monthly basis.

> "With Service Fabric, we can focus on implementing game design features instead of investing a lot of time setting up the infrastructure."

—Marko Flod: Software Engineer
Digamore Entertainment