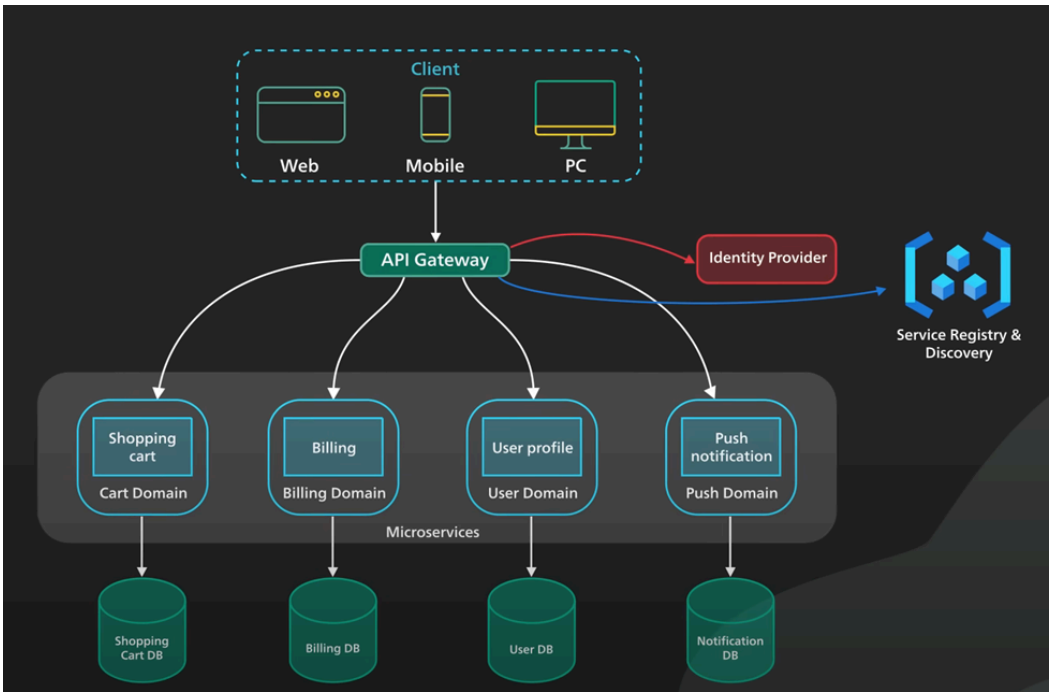
Cooked API Gateway Research Document
Authors: Curtis Reece, Tyler Campbell

What is an API Gateway:
An API (Application Programming Interface) Gateway is a midpoint between a software application and other applications or microservices that handles all communication between them. The API Gateway handles this communication by receiving an API call or request from the application. The API Gateway then routes the API call to the correct microservice(s) to gather the requested data. The API Gateway then handles the logic for combining all of the data that it gathered from the microservice(s) into a single package that it can then pass to the application that originated the API call.

API Gateways come with a lot of benefits related to security, scalability, and efficiency. For security, an API Gateway can be used to monitor requests, responses, and errors, can provide authentication and authorization, and is able to help with DDoS attacks through techniques like rate limiting. With scalability, they can help with load balancing by routing their requests to different microservices to ensure that microservices do not end up being overwhelmed with traffic. With efficiency, an API Gateway can cache responses to common API calls and can reduce the complexity of a workflow to create a simpler and more efficient procedure.

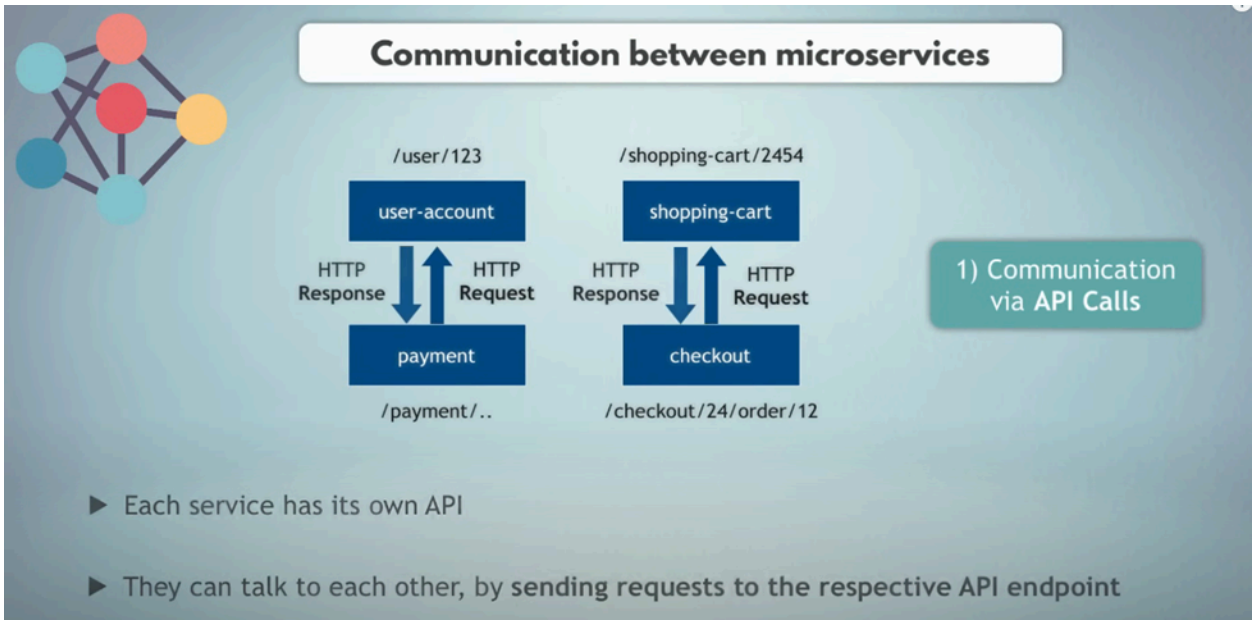
They do present a single point of failure and have some scalability issues as when given too many requests to handle they can result in some higher latency.

Here is a visual from one of our resources. In this example, there is additional abstraction by using an identity provider and service registry that will likely not be needed for our project.

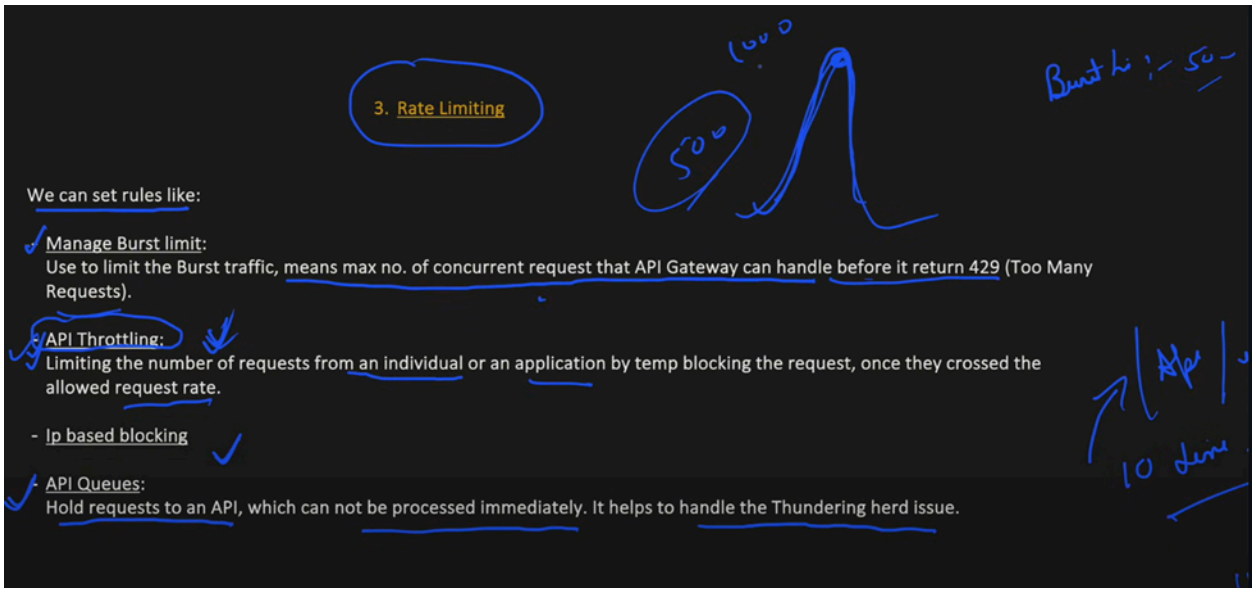


What should an API Gateway do:

An API Gateway should, as mentioned above, handle requests from an application for data from other microservices. This means that the application never actually interacts with the microservices directly, instead interacting with the API Gateway which then interacts with the microservices to gather the data requested, combine it into a usable format for the application, then return it to the application.



The API Gateway can handle traffic in various ways, such as burst limits, throttling for an application or individual user, and Queues of API messages that will be handled over time.



What does our API Gateway do and not do to facilitate this workflow:

Our API Gateway Solution has a GameInfo.cs file that appears to be a class for making objects that would store collected data from each microservice. These objects would be able to pass the requested information to the application, following the workflow of an API Gateway.

Our API Gateway Solution also has a GatewayController.cs file that appears to handle the logic for communicating with the different microservices to retrieve a GameInfo object and add these objects to a list that it would return to the application. It is set up to get the game information from every microservice that is currently present (Snake, Tetris, and Pong).

Our API Gateway Solution also has a Program.cs file that appears to set up the pipeline for HTTP requests using Swagger/OpenAPI. This should allow the communication between the application and the API Gateway and the communication between the microservices and the API Gateway to function.

However, our API Gateway does not actually communicate between services. We have hardcoded values for the game info objects within the games.json file in the BucStop Repo. The files above mimic parts of the actual workflow, but are not being used so we do not know if they are functional.

So, our API Gateway has some sample code for the Application making an initial request for information on what games are available and the information related to them in order to set up a thumbnail for the carousel and have their information displayed within the app. This sample code mocks the workflow for being able to contact the microservices to gather the GameInfo objects which contain the information the application needs, combines them together, then sends this to the application.

In addition, our solution is not actually using the API Gateway for running the game as the Javascript files for the games are not stored in their own microservices but are instead stored in the BucStop Repo. The application just runs whichever game the user has selected without contacting the API Gateway. The workflow that should be followed instead would involve the Javascript files for the games being stored within their own microservices. The user would select a game which would send a request to the API Gateway to either run the game or pass the Javascript file for the game back to the application so that the application can run the game depending on how we choose to design our solution.

For our current API Gateway, there are 3 endpoints seen below to get the game data for the 3 games. As mentioned above, these are not being utilized yet.

```
/// <summary>
/// Handles GET requests to retrieve game information from microservices.
/// </summary>
/// <returns>A collection of GameInfo objects.</returns>
[HttpGet]
public async Task<IEnumerable<GameInfo>> Get()
{
    try
    {
        var SnakeTask = AddGameInfo("https://localhost:1948", "/Snake" ); //Snake
        var Tetristask = AddGameInfo("https://localhost:2626", "/Tetris"); //Tetris
        var PongTask = AddGameInfo("https://localhost:1941", "Pong"); //Pong
        await Task.WhenAll(SnakeTask, Tetristask, PongTask);
        return TheInfo;
    }
```

```
public async Task AddGameInfo(string baseUrl, string endpoint)
```

Resources:

https://www.ibm.com/think/topics/api-gateway

https://www.youtube.com/watch?v=ITAcCNbJ7KE

https://youtu.be/rv4LlmLmVWk?si=-G_E3UOgvdE7EN0