CHAPTER 5

Application Services

In this chapter, we're going to focus on the pertinent issues in achieving scalability for the services tier in an application. We'll explore API and service design and describe the salient features of application servers that provide the execution environment for services. We'll also elaborate on topics such as horizontal scaling, load balancing and state management that we introduced in Chapter 2.

Service Design

In the simplest case, an application comprises one Internet facing service that persists data to a local data store, as shown in Figure 1. Clients interact with the service through its published API, which is accessible across the Internet.

Let's look at the API and service implementation in more detail.

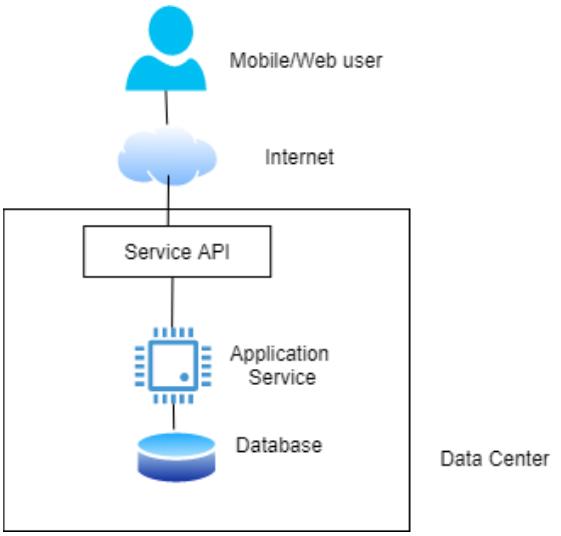

Figure 1 A Simple Service

Application Programming Interface (API)

An API defines a contract between the client and server. The API specifies the types of requests that are possible, the data that is needed to accompany the requests, and the results that will be obtained. APIs have many different variations, as we explored in RPC/RMI discussions in Chapter 4. While there remains some API diversity in modern applications, the predominant style relies on HTTP APIs. These are typically, although not particularly accurately, classified as RESTful.

REST is actually an architectural style that was defined by Roy Fielding in his PhD thesis[1]. A great source of knowledge on RESTful APIs and the various degrees to which Web technologies can be exploited is *REST in Practice* by *Webber, Parastatidis, and Robinson*. Here we'll just briefly touch on the HTTP *CRUD* API pattern. This pattern does not fully implement the principles of REST, but it is widely adopted in Internet systems today.

CRUD stands for *Create, Read, Update, Delete*. A CRUD API specifies how clients perform these operations in a specific business context. For example a user might *create* a profile, *read* catalog items, *update* their shopping cart and *delete* items from their order. A HTTP CRUD API makes these operations possible using four core HTTP verbs, as shown in Table 1.

Verb	**Uniform Resource Identifier Example**	**Purpose**
POST	/skico.com/skiers/{skierID}/{date}	Create a new ski day record for a skier
GET	/skico.com/skiers/(skierID)	Get the profile information for a skier, returned in a JSON response payload
PUT	/skico.com/skiers/{skierID}	Update skier profile
DELETE	/skico.com/skiers/{skierID}	Delete a skier's profile as they didn't renew their pass!

Table 1 HTTP CRUD Verbs

A HTTP CRUD API applies HTTP verbs on *resources* identified by Uniform Resource Identifiers (URIs). In Table 1 for example, a URI that identifies skier 768934 would be:

```
/skico.com/skiers/768934
```

A HTTP GET request to this resource would return the complete profile information for a skier in the response payload, such as name, address, number of days visited, and so on. If a client subsequently sends a HTTP PUT request to this URI, we are expressing the intent to update the resource for skier 768934 – in this example it would be the skier's profile. The PUT request would provide the complete representation for the skier's profile as returned by the GET request. Again, this would be as a payload with the request. Payloads are typically formatted as JSON, although XML and other formats are also possible. If a client sends a DELETE request to the same URI, then the skier's profile will be deleted.

Hence the combination of the HTTP verb and URI define the semantics of the API operation. Resources, represented by URIs, are conceptually like objects in Object Oriented Design (OOD) or entities in Entity-Relationship (ER) model. Resource identification and modeling hence follows similar methods to OOD and ER modeling. The focus however is on resources that need to be exposed to clients in the API. The Further Reading section at the end of this chapter points to

[1] https://en.wikipedia.org/wiki/Roy_Fielding

useful sources of information for resource design.

HTTP APIs can be specified using a notation called OpenAPI[2]. At the time of writing the latest version is 3.0. A tool called SwaggerHub[3] is the de facto standard to specify APIs in OpenAPI. The specification is defined in YAML, and an example is show in Figure 2. It defines the `GET` operation on the URI `/resorts`. If the operation is successful, a 200 response code is returned along with a list of resorts in a format defined by a JSON schema that appears later in the specification. If for some weird reason, the query to get a list of resorts operated by skico.com returns no entries, a 404 response code is returned along with an error message that is also defined by a JSON schema.

```
1.  paths:
2.    /resorts:
3.      get:
4.        tags:
5.          - resorts
6.        summary: get a list of ski resorts in the database
7.        operationId: getResorts
8.        responses:
9.          '200':
10.           description: successful operation
11.           content:
12.             application/json:
13.               schema:
14.                 $ref: '#/components/schemas/ResortsList'
15.         '404':
16.           description: Resorts not found. Unlikely unless we go broke
17.           content:
18.             application/json:
19.               schema:
20.                 $ref: '#/components/schemas/responseMsg'
21.         ‘500’:
22.             $ref: "#/responses/Standard500ErrorResponse"
```

Figure 2 OpenAPI Example

It is also possible for the client to see a `500 Internal Error` response code. This indicates that the server encountered an unexpected condition that prevented it from fulfilling the request. This error code is a generic "catch-all" response. It may be thrown if the server has crashed, or if the server is temporarily unavailable due to a transient network error.

Designing Services

An application server container receives requests and routes them to the appropriate handler function to process the request. The handler is defined by the application service and implements the business logic required to generate results from the request. As multiple simultaneous requests arrive at a service instance, each is typically[4] allocated a thread context to execute the request.

The sophistication of the routing functionality varies widely by technology platform and

[2] https://app.swaggerhub.com/help/tutorials/openapi-3-tutorial

[3] https://app.swaggerhub.com

[4] Node.js is a notable exception here as it is single threaded. However, it employs an asynchronous programming model for blocking I-Os that supports handling many simultaneous requests.

language. For example, in Express.js, the container calls a specified function for requests that match an API signature – known as a route path - and HTTP method. Figure 3 illustrates this with a method that will be called when the client sends a GET request for a specific skier's profile, as identified by the value of `:skierID`.

```
1. app.get('/skiers/:skierID', function (req, res) {
2.   // process the GET request
3.   ProcessRequest(req.params)
4. }
```

Figure 3 Express.js request routing example

In Java, the Spring framework provides an equally sophisticated method routing technique. It leverages a set of annotations that define dependencies and implement dependency injection to simplify the service code. Figure 4 shows an example of annotations usage, namely:

- `@RestController` – identifies the class as a controller that implements an API and automatically serializes the return object into the `HttpResponse` returned from the API.
- `@GetMapping` – maps the API signature to the specific method, and defines the format of the response body
- `@PathVariable` – identifies the parameter as value that originates in the path for URI that maps to this method

```
1. @RestController
2. public class SkierController {
3.
4.     @GetMapping("/skiers/{skierID}",
5.                 produces = "application/json")
6.     public Profile GetSkierProfile(
7.                         @PathVariable String skierID,
8.                         ) {
9.           // DB query method omitted for brevity
10.          return GetProfileFromDB(skierID);
11.     }
12. }
```

Figure 4 Spring method routing example

Another Java technology, JEE servlets, also provide annotations, as shown in Figure 5, but these are simplistic compared to Spring and other higher-level frameworks. The @WebServlet annotation identifies the base pattern for the URI which should cause a particular servlet to be invoked. This is `/skiers` in our example. The class that implements the API method must extend the `HttpServlet` abstract class from the `javax.servlet.http` package and override at least one method that implements a HTTP request. The HTTP verbs map to methods as follows:

- `doGet`: HTTP GET requests
- `doPost`: HTTP POST requests
- `doPut`: HTTP PUT requests
- `doDelete`: for HTTP DELETE requests

Each method is passed as parameters a `HttpServletRequest` and `HttpServletResponse` object. The servlet container creates the `HttpServletRequest` object, which contains members that represent the components of the incoming HTTP request. This object contains the complete URI path for the call, and it is the servlet's responsibility to explicitly parse and validate this, and extract path and query parameters if valid. Likewise, the servlet must explicitly set the properties of the response using the `HttpServletResponse` object.

Servlets therefore require more code from the application service programmer to implement. However, they are likely to provide a more efficient implementation as there is less 'plumbing' involved compared to the more powerful annotation approaches of Spring et al. A good rule of thumb is that if a framework is easier to use, it is likely to be less efficient. This is a classic performance versus ease-of-use trade-off. We'll see lots of these in this book

```
1.  import javax.servlet.http.*;
2.  @WebServlet(
3.      name = "SkiersServlet",
4.      urlPatterns = "/skiers"
5.  )
6.  public class SkierServlet extends HttpServlet (
7.
8.  protected void doGet(HttpServletRequest request,
9.                      HttpServletResponse response) {
10.   // handles requests to /skiers/{skierID}
11.   try {
12.      // extract skierID from the request URI (not shown for brevity)
13.      String skierID  = getSkierIDFromRequest(request);
14.      if(skierID == null) {
15.         // request was poorly formatted, return error code
16.         response.setStatus(HttpServletResponse.SC_BAD_REQUEST);     }
17.      else {
18.         // read the skier profile from the database
19.         Profile profile = GetSkierProfile (skierID);
20.         // add skier profile as JSON to HTTP response and return 200
21.         response.setContentType("application/json");
22.         response.getWriter().write(gson.toJson(Profile);
23.         response.setStatus(HttpServletResponse.SC_OK);
24.      } catch(Exception ex) {
25.          response.setStatus
26.            (HttpServletResponse.SC_INTERNAL_SERVER_ERROR);     }
27.
28.        }
29. } }
```

Figure 5 JEE Servlet Example

State Management

State management is a tricky, nuanced topic. The bottom line is that service implementations that need to scale should avoid storing conversational state. What on earth does that mean? Let's start by examining the topic of state management with HTTP.

HTTP is known as stateless protocol. This means each request is executed independently, without any knowledge of the requests that were executed before it from the same client. Statelessness implies that every request needs to be self-contained, with sufficient information

provided by the client for the Web server to satisfy the request regardless of previous activity from that client.

The picture is a little more complicated that this simple description portrays, however. For example:

- The underlying socket connection between a client and server is kept open so that the overheads of connection creation are amortized across multiple requests from a client. This is the default behavior for versions HTTP/1 and above.
- HTTP supports cookies, which are known as the HTTP State Management Mechanism[5]. Gives it away really!
- HTTP/2 supports streams, compression, and encryption, all of which require state management

So, originally HTTP was stateless, but perhaps not anymore? Armed with this confusion (!), let's look at application services APIs built on top of HTTP.

When a user or application connects to a service, it will typically send a series of requests to retrieve and update information. Conversational state represents any information that is retained between requests such that the subsequent request can assume the service has retained knowledge about the previous interactions. Let's explore a simple example.

In our skier service API, a user may request their profile by submitting a GET request to the following URI:

```
/skico.com/skiers/768934
```

They may then use their app to modify their phone number and send a PUT request to a URI designed for updating this field:

```
/skico.com/skiers/phoneno/4123131169
```

As this URI does not identify the skier, the service must know the unique identifier of the skier, namely 768934. Hence for this PUT operation to succeed, the service must have retained conversational state from the previous GET request.

Implementing this approach is relatively straightforward. When the service receives the initial GET request, it creates a session state object that uniquely identifies the client connection. In reality, this is often performed when a user first connects to or logs in to a service. The service can then read the skier profile from the database and utilize the session state object to store conversational state – in our example this would be `skierID`. When the subsequent PUT request arrives from the client it uses the session state object to look up the `skierID` associated with this session and uses that to update the skier's phone number.

Services that maintain conversational state are known as stateful services. Stateful services are attractive from a design perspective as they can minimize the number of times a service retrieves data (state) from the database and reduce the amount of data that is passed between clients and the services. For services with light request loads they can make eminent sense and are promoted by many frameworks to make services easy to build and deploy. For example, JEE servlets support session management using the `HttpSession` object, and similar capabilities are offered by the `Session` object in ASP.NET.

As we scale our service implementations however, the stateful approach becomes problematic. For a single service instance, we have two problems to consider:

[5] https://tools.ietf.org/html/rfc6265

1. If we have multiple client sessions all maintaining session state, this will utilize available service memory. The amount of memory utilized will be proportional the number of clients we are maintaining state for. If a sudden spike of requests arrive, how can we be certain we will not exhaust available memory and cause the service to fail?
2. We also must be mindful about how long to keep session state available. A client may stop sending requests but not cleanly close their connection to allow the state to be reclaimed. All session management approaches support a default session time out. If we set this to a short time interval, clients may see their state disappear unexpectedly. If we set the session time out period to be too long, we may degrade service perform as it runs low on resources.

In contrast, stateless services do not assume that any conversational state from previous calls has been preserved. The service should not maintain any knowledge from earlier requests, so that each request can be processed individually. This requires the client to provide all the necessary information for the service to process the request and provide a response.

If we consider our example above, we could transform the PUT request to be stateless by incorporating the `skierID` in the URI:

```
/skico.com/skiers/768934/phoneno/4123131169
```

In reality, this is a pretty dumb API design. As we discussed in the service API section, the API should in fact transfer the complete resource – the skier profile – with the GET response and PUT request. This means the GET request for the skier should retrieve and return the complete skier profile so that the client app can display and modify any data items in the profile as the user wishes. The complete updated profile is then sent as the payload in the subsequent PUT operation, and used by the service to update the resource, that is persisted in a database. This is shown in Figure 6.

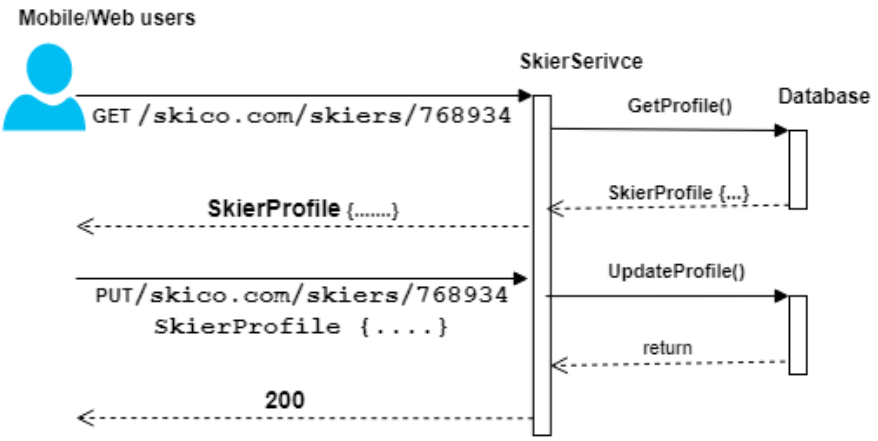

Figure 6 Stateless API Example

Any scalable service will need stateless APIs. If a service needs to retain state pertaining to client sessions – the classic shopping cart example - it must be stored externally to the service. This invariably means an external data store.

We'll revisit this topic later in this chapter during the discussion of horizontal scaling. That's when stateless services really come to the fore.

Applications Servers

Application servers are the heart of a scalable application, hosting the business services that comprise an application. Their basic role is to accept requests from clients, apply application logic to the requests, and reply to the client with the request results. Clients may be external or internal, as in other services in the application that require to use the functionality of a specific service.

The technological landscape of application servers is broad and complex, depending on the language you want to use and the specific capabilities that each offer. In Java, the Java Enterprise Edition (JEE)[6] defines a comprehensive, feature rich standards-based platform for application servers, with multiple different vendor and open source implementations.

In other languages, the Express.js[7] server supports Node, Flask supports Python[8], and in GoLang a service can be created by incorporating the `net/http` package. These implementations are much more minimal and lightweight than JEE and are typically classified are Web application frameworks. In Java, the Apache Tomcat server[9] is a somewhat equivalent technology. Tomcat is open source implementation of a subset of the JEE platform, namely the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies.

Figure 7 depicts a simplified view of the anatomy of Tomcat. Tomcat implements a *servlet container*, which is an execution environment for application-defined servlets. Application defined Servlets are loaded into this container, which provides lifecycle management and a multithreaded runtime environment.

Request arrive at the IP address of the server, which is listening for traffic on specific ports. For example, by default Tomcat listens on port 8080 for HTTP requests and 8443 for HTTPS requests. Incoming requests are processed by one or more listener threads. These create a TCP/IP socket connection between the client and server. If network requests arrive at a frequency that cannot be processed by the TCP listener, pending requests are queued up in the *Sockets Backlog*. The size of the backlog is operating system dependent. In most Linux versions the default is 100.

Once a connection is established, the TCP requests are marshalled by, in this example, a *HTTP Connector* which generates the HTTP request that the application can process. The HTTP request is then dispatched to an application service thread to process. Application container threads are managed in a thread pool, essentially a Java `Executor`, which by default in Tomcat is a minimum size of 25 threads and a maximum of 200. If there are no available threads to handle a request, the container maintains them in a queue of runnable tasks and dispatches these as soon as a thread becomes available. This queue by default is size `Integer.MAX_VALUE` – that is, essentially unbounded[10]. If a thread remains idle for by default, 60 seconds, it is killed to free up resources in the JVM.

---

[6] https://www.oracle.com/java/technologies/java-ee-glance.html

[7] https://expressjs.com/

[8] https://palletsprojects.com/p/flask/

[9] http://tomcat.apache.org/

[10] See https://tomcat.apache.org/tomcat-9.0-doc/config/executor.html for default Tomcat Executor configuration settings

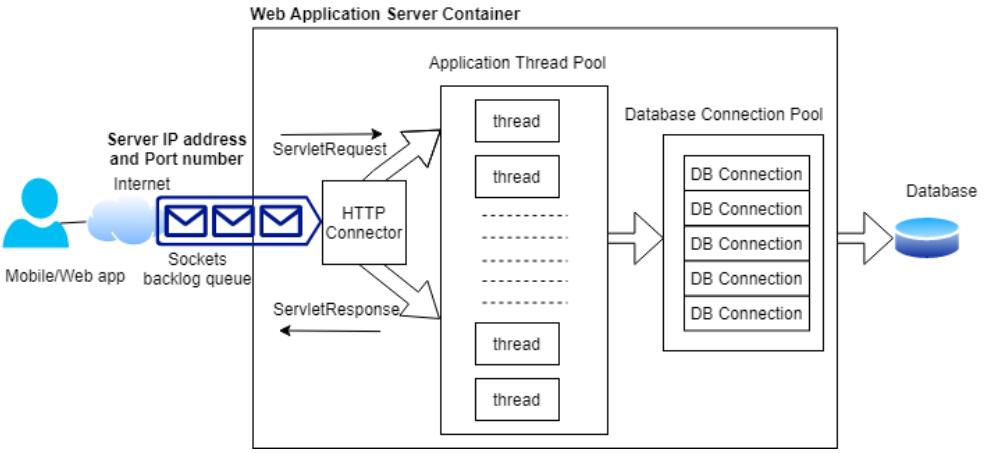

Figure 7 Anatomy of a Web application server

For each request, the method that corresponds with the HTTP request is invoked in a thread. The servlet method processes the HTTP request headers, executes the business logic, and constructs a response that is marshalled by the container back to a TCP/IP packet and sent over the network to the client.

In processing the business logic, servlets often need to query an external database. This requires each thread executing the servlet methods to obtain a database connection and execute database queries. As database connections are limited resources and consume resources in both the client and database server, a fixed size connection pool is typically utilized. The pool hands out open connection to requesting threads on demand.

When a servlet wishes to submit a query to the database, it therefore requests an open connection from the pool. If one is available, access to the connection is granted to the servlet until it indicates it has completed its work. At that stage the connection is returned to the pool and made available for another servlet to utilize. As the thread pool is typically larger than the connection pool, a servlet may request a connection when none are available. The connection pool maintains a request queue and hands out open connections on a FIFO basis, and threads in the queue are blocked until there is availability.

An application server framework such as Tomcat is hence highly configurable to different handle different workloads. For example, the size of the thread and database connection pools can be specified in configuration files that are read at startup.

The complete Tomcat container environment runs within a single JVM, and hence processing capacity is limited by the number of vCPUs available and the amount of memory allocated as heap size. Each allocated thread consumes memory, and the various queues in the request processing pipeline consume resources while requests are waiting. This means that request latency will be governed by both the request processing time in the servlet and the time spent waiting in queues for threads and connections to become available.

In a heavily loaded server with many threads, context switching may start to degrade perform, and available memory may be become limited. If perform degrades, queues grow as requests wait for resources. This consumes more memory. If more requests are received than can be queued up and processed by the server, then new TCP/IP connections will be refused, and clients will see errors. Eventually, an overloaded server will run out of resources and start throwing exceptions and crash.

Time spent tuning configuration parameters to efficiently handle anticipated loads is rarely wasted. A rule-of-thumb is that CPU utilization that consistently exceeds the 70-80% range is a

signal of overload. Similar insights exist for memory usage. Once any resource gets close to full utilization, systems tend to exhibit less predictable performance, as more time is spent, for example thread context switching and garbage collecting. This inevitably effects latencies and throughput.

Monitoring tools available with Web application frameworks enable engineers to gather a range of important metrics, including latencies, active requests, queue sizes and so on. These are invaluable for carrying out data-driven experiments that lead to performance optimization.

Java-based application frameworks such as Tomcat will invariably support the JMX[11] (Java Management Extensions) framework, which is a standard part of the Java Standard Edition platform. JMX enables frameworks to expose monitoring information based on the capabilities of MBeans (Managed Beans), which represent a resource of interest (e.g. thread, database connections usage). This enables an eco-system of tools to offer capabilities for monitoring JMX-supported. These range from JConsole[12] which is available in the JDK by default, to powerful open source technologies such as JavaMelody[13] and many expensive commercial offerings.

Horizontal Scaling

A core principle of scaling a system is being able to easily add new processing capacity to handle increased load. For most systems, a simple and effective approach is deploying multiple instances of stateless server resources and using a load balancer to distribute the requests across these instances. This is known as horizontal scaling and illustrated in Figure 8.

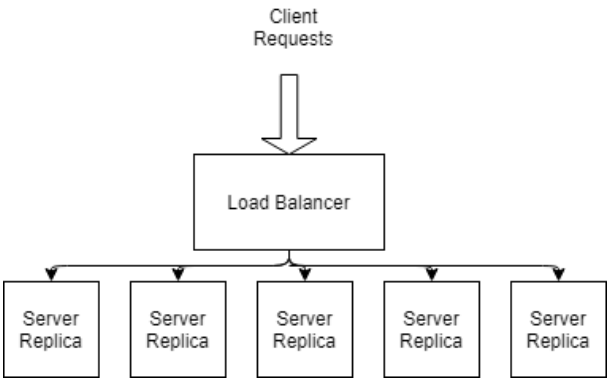

Figure 8 Simple Load Balancing Example

These two ingredients, namely stateless service replicas and a load balancer, are both necessary. Let's explain why.

Service replicas are deployed on their own (virtual) hardware. Hence if we have two replicas, we double our processing capacity. If we have ten replicas, we have 10x capacity. This enables our system to handle increased loads. The aim of horizontal scaling is to create a system processing capacity that is the sum of the total resources available

The servers need to be stateless, so that any request can be sent to any service replica to handle. This decision is made by the load balancer, which can use various policies to distribute requests.

[11] https://en.wikipedia.org/wiki/Java_Management_Extensions
[12] https://en.wikipedia.org/wiki/JConsole
[13] https://github.com/javamelody/javamelody/wiki

If the load balancer can keep each service replica equally busy, then we are effectively using the processing capacity provided by the service replicas.

Horizontal scaling also increases availability. With one service instance, if it fails, the service is unavailable. This is known as a single point of failure (SPoF) – a bad thing, and one to avoid in any scalable distributed system. Multiple replicas increase availability. If one replica fails, requests can be directed to any – they are stateless, remember – replica. The system will have reduced capacity until the failed server is replaced, but it will still be available. Which is important. The ability to scale is crucial, but if a system is unavailable, then the most scalable system ever built is still somewhat ineffective!

Load Balancing

Load balancing aims to effectively utilize the capacity of a collection of services to optimize the response time for each request. This is achieved by distributing requests across the available services as evenly as possible and avoiding overloading some services while underutilizing others. Clients send requests to the IP address of the load balancer, which redirects requests to target services, and relays the results back to the client. This means clients never contact the services directly, which is also beneficial for security as the services can live behind a security perimeter and not be exposed to the Internet.

Load balancers may act at the *network level* or the *application level*. These are often called *Layer 4* and *Layer 7* load balancers respectively. These names refer to network transport layer at Layer 4 in the Open Systems Interconnection (OSI) Reference Model[14], and the application layer at Layer 7. The OSI model defines network functions in seven abstract layers. Each layer defines standards for how data is packaged and transported. Let's explore the differences between the two techniques.

Network level load balancers distribute requests at the network connection level, operating on individual TCP or UDP packets. Routing decisions are made on the basis of client IP addresses. Once a target service is chosen, the load balancer uses a technique called Network Address Translation (NAT). This changes the destination IP address in the client request packet from that of the load balancer to that of the chosen target. When a response is received from the target, the load balancer changes the source address recorded in the packet header from the target's IP address to its own. Network load balancers are relatively simple as they operate on the individual packet level. This means they are extremely fast, as they provide few features beyond choosing a target service and performing NAT functionality.

In contrast, application level load balancers reassemble the complete HTTP request and base their routing decisions on the values of the HTTP headers and on the actual contents of the message. For example, a load balancer can be configured to send all POST requests to a subset of available services, or distribute requests based on a query string in the URI. Application load balancers are sophisticated reverse proxies. The richer capabilities they offer means they are slightly slower than network load balancers, but the powerful features they offer can be utilized to more than make up for the overheads incurred.

In general, a load balancer has the following features which are explained in the following sections:

1. Load distribution policies
2. Health monitoring

[14] https://en.wikipedia.org/wiki/OSI_model

3. Elasticity
4. Session affinity

Load Distribution Policies

Load distribution policies dictate how the load balancer chooses a target service to process a request. Any load balancer worth its salt will offer several load distribution policies – HAProxy offers 10 in fact[15]. The following are probably the most commonly supported across all load balancers:

- round-robin: the load balancer distributes requests to available servers in a round-robin fashion
- least connections: the load balancer distributes new requests to the server with the least open connections
- HTTP header field: the load balancer directs requests based on the contents of a specific HTTP header field. For example all requests with the header field `X-Client-Location:US,Seattle` could be routed to a specific set of servers.
- HTTP operation: the load balancer directs requests based on the HTTP verb in the request

Load balancers will also allow services to be allocated weights. For example, standard service instances in the load balancing pool may have 4 vCPUs and each is allocated a weight of 1. If a service with 8 vCPUs is added, it can be assigned a weight of 2 so the load balancer will send twice as many requests its way.

Health Monitoring

A load balancer will periodically sends pings and attempts connections to test the health of each service in the load balancing pool. These tests are called health checks. If a service becomes unresponsive or fails connection attempts, it will be removed from the load balancing pool and no requests will be sent to that host. If the connection to the service has experienced a transient failure, the load balancer will reincorporate the service once it becomes available and healthy. If, however it has failed, the service will be removed from the load balancer target pool.

Elasticity

Spikes in request loads can cause the service capacity available to a load balancer to become saturated, leading to longer response times and eventually request and connection failures. *Elasticity* is the capability of a load balancer to dynamically provision new service capacity to handle an increase in requests. As load increases, the load balancer starts up new resources and directs requests to these, and as load decreases the load balancer stops services that are no longer needed.

An example of elastic load balancing is the Amazon Web Services (AWS) Auto-Scaling groups. An Auto Scaling group is a collection of services available to a load balancer that is defined with a minimum and maximum size. The load balancer will ensure the group always has the minimum numbers of services available, and the group will never exceed the maximum number. The actual number available at any time will depends on the client request load the load balancer is handling for the services in the group. This scheme is illustrated in Figure 9.

[15] http://cbonte.github.io/haproxy-dconv/2.3/intro.html#3.3.5

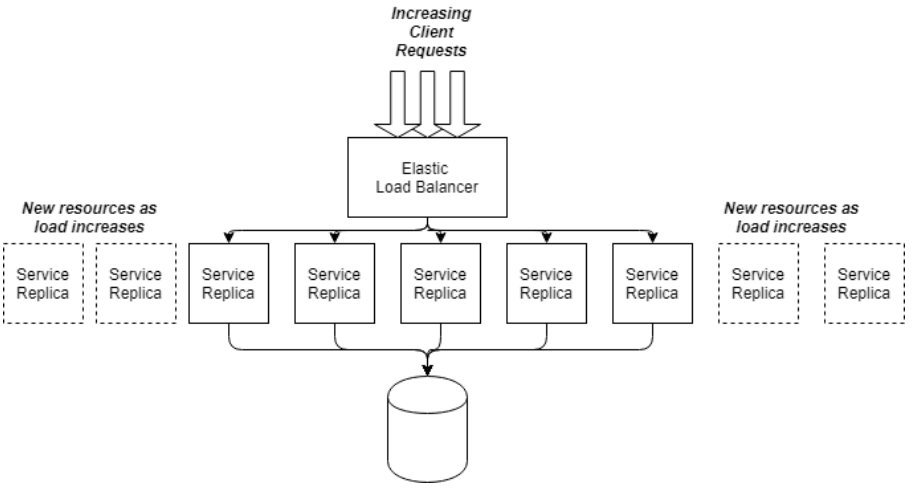

Figure 9 Elastic Load Balancing

Typically, there are two ways to control the number of services in a group. The first is based on a schedule, when the request load increases and decreases are predictable. For example, you may have an online entertainment guide and publish the weekend events for a set of major cities at 6pm on Thursday. This generates a higher load until Sunday at noon. An Auto Scaling group could easily be configured to provision new services at 6pm Thursday and reduce the group size to the minimum at noon Sunday.

If increased load spikes are not predictable, elasticity can be controlled dynamically with a set of configuration parameters. The most commonly used parameter is average CPU utilization across the active services. For example, an Auto Scaling group can be defined to maintain average CPU utilization at 70%. If this value is exceeded, the load balancer will start one or more new services instances until the 70% threshold is reached. Instances need time to start – often a minute or more - and hence a *warmup* period can be defined until the new instance is considered to be contributing to the group's capacity. When the group average CPU utilization drops below 70%, *scale in* or *scale down* will start and instances will be automatically stopped and removed from the pool.

Elasticity is a key feature that allows services to scale dynamically as demand grows. For highly scalable systems, it is a mandatory capability. Like all advanced capabilities however, it brings new issues for us to consider in terms of downstream capacity and costs. We'll discuss these in Chapter XXXX.

Session Affinity

Session affinity, or sticky sessions, are a load balancer feature for stateful services. With sticky sessions, the load balancer sends all requests from the same client to the same service instance. This enables the service to maintain in-memory state about each specific client session.

There are various ways to implement sticky sessions. For example, HAProxy provides a comprehensive set of capabilities to maintain client requests on the same service in the face of service additions, removals and failures[16]. AWS Elastic Load Balancing generates an HTTP cookie that identifies the service a client's session is associated with. This cookie is returned to the client, which must send it in subsequent request to ensure session affinity is maintained.

Sticky sessions can be problematic for highly scalable systems. They lead to a load imbalance

[16] http://cbonte.github.io/haproxy-dconv/2.3/intro.html#3.3.6

problem, in which, over time, clients are not evenly distributed across services. This is illustrated in Figure 10, where two clients are connected to one service while another service remains idle.

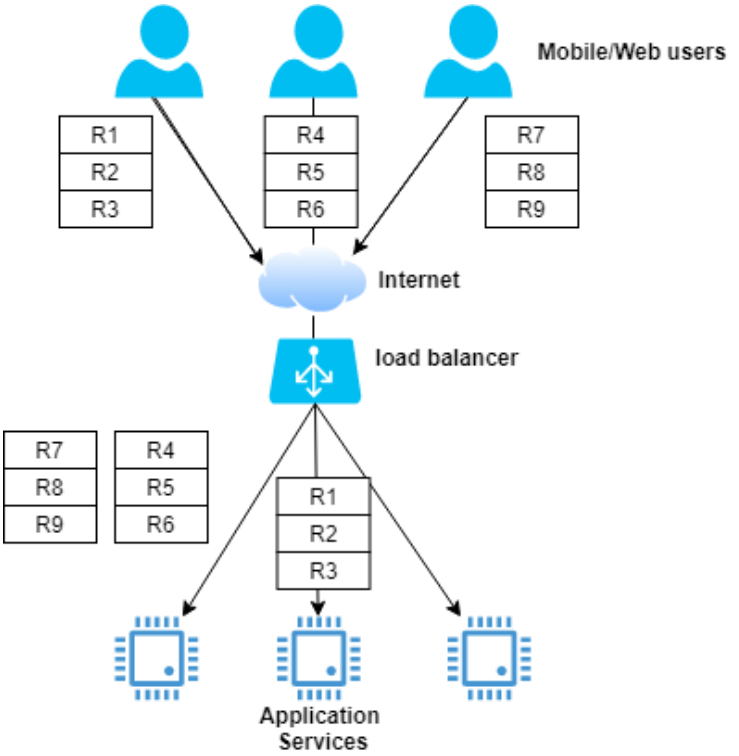

Figure 10 Load Imbalance with Sticky Sessions

Load imbalance occurs because client sessions last for varying amounts of time. Even if sessions are evenly distributed initially, some will terminate quickly while others will persist. In a lightly loaded system, this tends to not be an issue. However, in a system with millions of sessions being created and destroyed constantly, load imbalance is inevitable. This will lead to some services being underutilized, while others are overwhelmed and may potentially fail due to resource exhaustion.

Stateful services have other downsides. When a service inevitably fails, how do the clients connected to that server recover the state that was being managed? If a service instance becomes slow due to high load, how do clients respond? In general stateful servers create problems that in large scale systems are difficult to design around and manage.

Stateless services have none of these downsides. If one fails, clients get an exception and retry, with their request routed to another live service. If a service is slow due to a transient network outage, the load balancer takes it out of the service group until it passes health checks of fails. All state is either externalized or provided by the client in each request, so service failures can be handled easily by the load balancer.

Stateless services enhance scalability, simplify failure scenarios and ease the burden of service management. For scalable applications, these advantages far outweigh the disadvantages, and hence their adoption in most major Internet sites such as Netflix.[17]

[17] https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Summary and Further Reading

Services are the heart of a scalable software system. They define the contract defined as an API that specifies their capabilities to clients. Services are defined and execute in an application server container environment that hosts the service code and routes incoming API requests to the appropriate processing logic. Application servers are highly programming language dependent, but in general provide a multithreaded programming model that allows services to process many requests simultaneously. If the threads in the container thread pool are all utilized, the application server queues up requests until a thread becomes available.

As request loads grow on a service, we can scale it out horizontally using a load balancer to distribute requests across multiple instances. This architecture also provides high availability as the multiple services configuration means the application can tolerate failures of individual instances. The service instances are managed as pool by the load balancer, which utilizes a load distribution policy to choose a target service for each request. Stateless services scale easily and simplify failure scenarios by allowing the load balancer to simply resend requests to responsive targets. Although most load balancers will support stateful services using a feature called sticky sessions, stateful services make load balancing and handling failures more complex. Hence, they are not recommended for highly scalable services.

API design is a topic of great complexity and debate. An excellent overview of basic API design and resource modeling is https://www.thoughtworks.com/insights/blog/rest-api-design-resource-modeling.

The Java Enterprise Edition (JEE) is an established and widely deployed server-side technology. It has a wide range of abstractions for building rich and powerful services. The Oracle tutorial is an excellent starting place for appreciating this platform - https://docs.oracle.com/javaee/7/tutorial/.

Much of the knowledge and information about load balancers is buried in the documentation provided by the technology suppliers. You choose your load balancer and then dive into the manuals. For an excellent, broad perspective on the complete field of load balancing, this is a good resource.

Tony Bourke, Server Load Balancing, O'Reilly Publishing