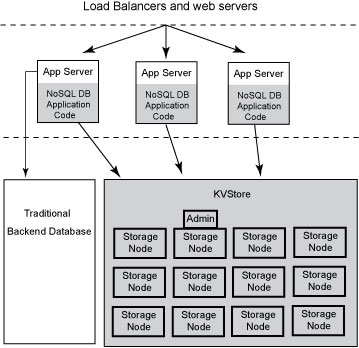
**Used tools:**

**Oracle NoSQL store:**

The KVStore (Key/Value Store), datastore or datacenter is a collection of storage nodes that hosts a set of replication nodes. Data is distributed across the replication nodes. The store contains multiple storage nodes. storage nodes. A storage node is a physical (or virtual) machine with its own local storage space. It should be but does not have to be, identical to all other storage nodes in the store. The following illustration represents the typical architecture used by an application that allows the use of Oracle NoSQL databases.



Every Storage Node hosts one or more Replication Nodes as determined by its *capacity*. The capacity of a Storage Node serves as a rough measure of the hardware resources associated with it. A store can consist of Storage Nodes of different capacities. Oracle NoSQL Database will ensure that a Storage Node is assigned a load that is proportional to its capacity. A Replication Node in turn contains at least one and typically many partitions. Also, each Storage Node contains monitoring software that ensures the Replication Nodes it hosts are running and otherwise healthy.

**Replication Nodes and Shards**

At a very high level, a Replication Node can be thought of as a single database that contains key-value pairs.

Replication Nodes are organized into shards. A shard contains a single Replication Node which is responsible for performing database writes, and which copies those writes to the other Replication Nodes in the shard. This is called the master node. All other Replication Nodes in the shard are used to service read-only operations. These are called replicas. Although there can be only one master node at any given time, any of the members of the shard can become a master node. In other words, each shard uses a single master/multiple replica strategies to improve read throughput and availability.

The following illustration shows how the KVStore is divided up into shards:

**Chart, bubble chart

Description automatically generated**

Note that if the machine hosting the master should fail in any way, then the master automatically fails over to one of the other nodes in the shard. (That is, one of the replica nodes is automatically promoted to master.)

Production KVStores should contain multiple shards. At installation time you provide information that allows Oracle NoSQL Database to automatically decide how many shards the store should contain. The more shards that your store contains, the better your write performance is because the store contains more nodes that are responsible for servicing write requests.

**Replication Factor**

The number of nodes belonging to a shard is called its *Replication Factor*. The larger a shard's Replication Factor, the faster its read throughput (because there are more machines to service the read requests) but the slower its write performance (because there are more machines to which writes must be copied). You set the Replication Factor for the store, and then Oracle NoSQL Database makes sure the appropriate number of Replication Nodes are created for each shard that your store contains.

**Partitions**

Each shard contains one or more *partitions*. Key-value pairs in the store are organized according to the key. Keys, in turn, are assigned to a partition. Once a key is placed in a partition, it cannot be moved to a different partition. Oracle NoSQL Database automatically assigns keys evenly across all the available partitions.

As part of your planning activities, you must decide how many partitions your store should have. Note that this is not configurable after the store has been installed.

It is possible to expand and change the number of Storage Nodes in use by the store. When this happens, the store can be reconfigured to take advantage of the new resources by adding new shards. When this happens, partitions are balanced between new and old shards by redistributing partitions from one shard to another. For this reason, it is desirable to have enough partitions so as to allow fine-grained reconfiguration of the store. Note that there is a minimal performance cost for having many partitions. As a rough rule of thumb, there should be at least 10 to 20 partitions per shard. Since the number of partitions cannot be changed after the initial deployment, you should consider the maximum future size of the store when specifying the number of partitions.

**Topologies**

A *topology* is the collection of storage nodes, replication nodes and administration services that make up an NoSQL DB store. A deployed store has one topology that describes its state at a given time.

After initial deployment, the topology is laid out to minimize the possibility of a single point of failure for any given shard. This means that while a Storage Node might host more than one Replication Node, those Replication Nodes will never be from the same shard. This improves the chances of the shard continuing to be available for read and writes even in the face of a hardware failure that takes down the host machine.

Topologies can be changed to achieve different performance characteristics, or in reaction to changes in the number or characteristics of the Storage Nodes. Changing and deploying a topology is an iterative process.

**Proxmox Virtual Environment:**

Proxmox VE (Virtual Environment) is a complete open-source server virtualization management solution. It tightly integrates the KVM hypervisor and Linux Containers (LXC), software-defined storage and networking functionality, on a single platform. Proxmox VE offers a web interface accessible after installation on your server which makes management easy, typically needing only a few clicks.

Proxmox VE was developed by Proxmox Server Solutions in Austria under the Internet Foundation of Austria and is released under the GNU General Public License. Since it’s an open-source solution it can be customized as per your requirements.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, email

Description automatically generated

**REST API:**

API or Application Programming Interface are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.

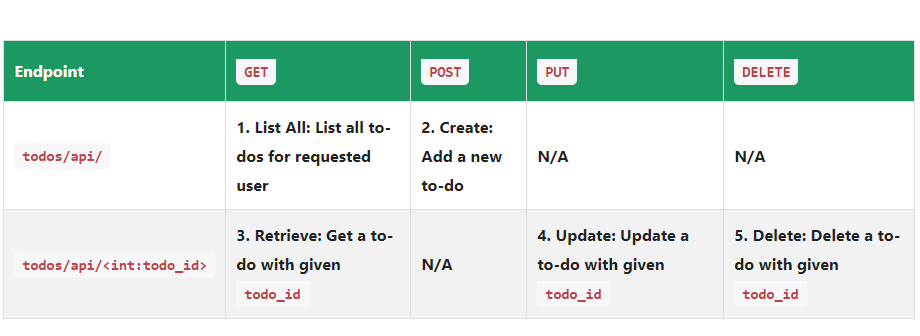
A REST API is used for systems to expose useful functions and data. REST stands for representational state transfer, which can be made up of one or more resources that can be accessed at a given URL and returned in various formats, like JSON, images, HTML, and more.

Building a RESTful API is in most cases complicated, but with the help of Django and Django REST framework, this complexity is handled extremely well.

**Django:**

Django is an open-source and python-based web framework that uses the Model-View-Template (MVT) architectural pattern.

Django has a powerful and flexible toolkit called Django REST framework (DRF) and it’s a useful tool for building Web APIs. It adopts implementations like class-based views, forms, model validators, QuerySet, and more. Its main functionality is [making serialization much easier](https://blog.logrocket.com/understanding-typescript-object-serialization/).

In a RESTful API, endpoints define the structure and usage with the GET, POST, PUT, and DELETE HTTP methods, which must be organized logically.

Models in Django are classes containing the fields we want our database table to contain. After creating the model, you will need to migrate it to the database using the command lines:

**python manage.py makemigrations**

**python manage.py migrate**

**Serialization:**

Serialization is such an important property built into the Django REST framework as it ensures the conversion of the object into a format that can be stored or transmitted. Therefore, after the serialized data has been transmitted or stored, you’ll be able to reconstruct the object and thus obtain the same structure as the object was originally in.

Python uses many available different formats for serialization, one common example that works across many languages is the JSON file format which is human-readable and allows us to store the dictionary and recreate it with the same structure. To convert the Model object to an API-appropriate format like JSON, the Django REST framework uses the ModelSerializer class to convert any model to serialized JSON objects. This makes serialization an important asset to Django REST Framework for web development.

