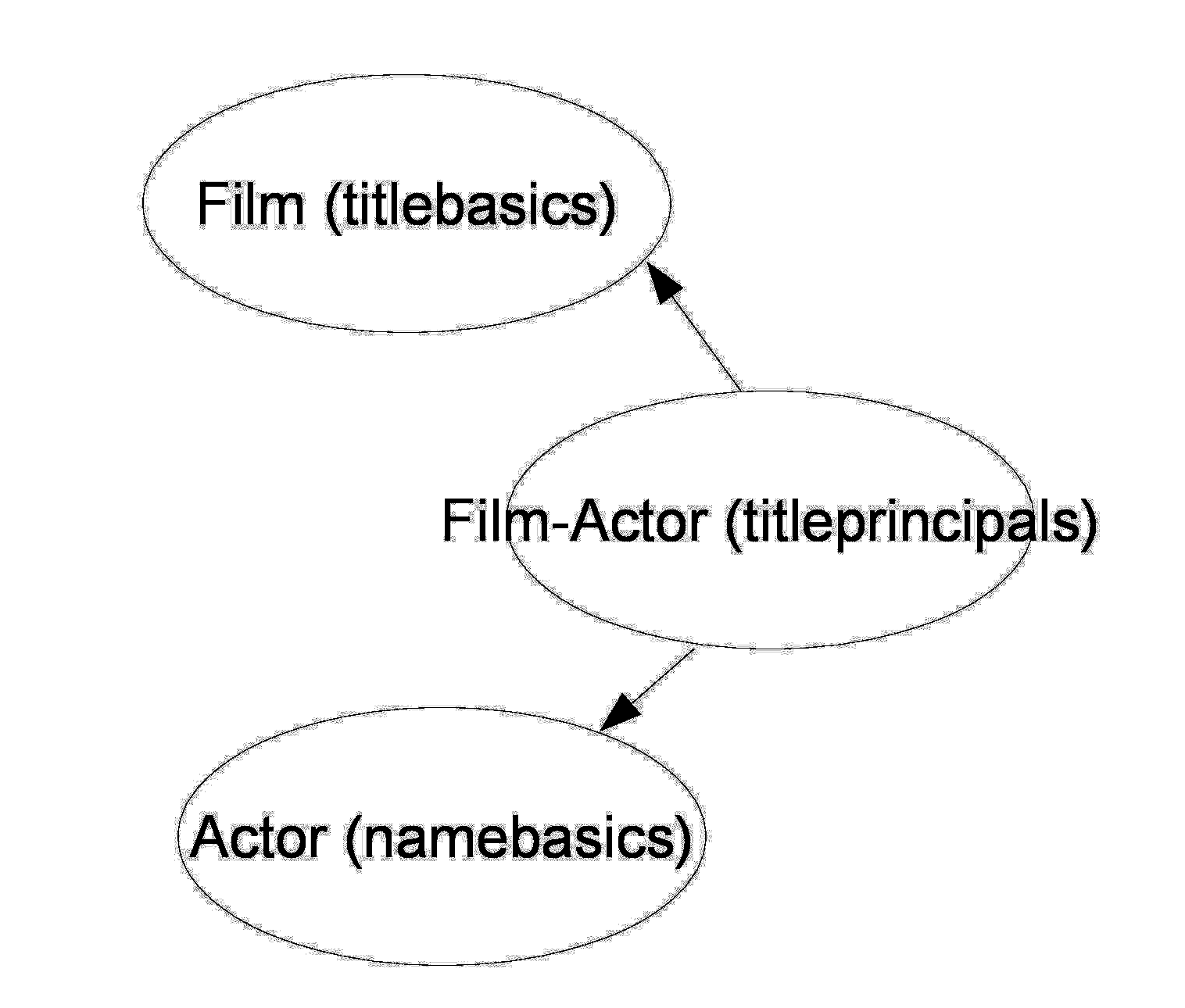
Design for degrees of separation

NB. the term “Film” is used to a “Movie” as the logical entity used in the program code is named Film. Both terms are used interchangeably. A Film is actually a film title, i.e. Several Films may hold the same primaryTitle: some of which may be a video, a documentary with the same name but made in a different year, etcetera.

In case of the Degree of Separation function, it is vital to use filtering extensively. See text below how the filtering has been determined.

Logical relationships:



Filtering of relevant information:

An Actor is a record from the namebasics table for which the date of birth is not empty. This is to filter out relatively unknown actors. Many actors share the same primary name.

In order to limit the number of hits for a film title, a Film is defined to be a record from table titlebasics where the titleType has one of the values ​​'short', 'movie', 'tvshort' and 'tvMovie'.

The Film\_Actor relationship is established via the titleprincipals table based on the reference key tconst (to film) and nconst (to Actor). Only the values ​​from the titleprincipal table are used where the category has one of the values ​​('actor', 'self' or 'archive\_footage'). In this way only the relationships are used with persons who play an actor role in that particular film title.

## Definition:

The actor list of first separation of a certain actor are the actors where a Film\_Actor relationship record exists for a common film title. I.e.: If actors A1 and A2 have degree of separation 1, then a title T1 exists with two Actor-Film records: one for A1 and T1 and one for A1 and T2.

### Algorithm: Construct separation sets for level 1 for actor A1.

Retrieve all Film keys where actor A1 is part of the actor crew

Retrieve all Actor keys for these films

Algorithm: Construct separaton sets level n+1 for actor A1 using previous levels

For all actors in previous separation set, construct the separation set level1 for this actor  
Merge the actors in those separation sets into a set of actors

The separation sets grow exponentially. It is therefore wise to try to avoid to create higher levels of separation.

- It is not necessary to create the separation set completely as soon as there is a hit (second actor is found in the list of actors inside a loop finding actors in a film)

- Only start creating a higher level of separation when the previous level did not succeed in delivering a hit.

# Algorithm

This leads to the following recursive function: (pseudocode)

Actor Check (**actor1**, **actor2**, int **level**)

Found = false

actorfilmlist (<actor>, <film>) is new empty list

loop find find film **film** having actor **actor1** in its actor crew

loop for all actors **actor** in crew for **film** {

if **actor** equals **actor2** {

print “separation level is **level”**

print “**actor2** was in **film** with **actor** with **actor1** and **actor2”**

return **actor**

} else

add (**actor**, **film**) to actorfilmlist

}

}

// not found, otherwise program does not continue here:

if (**level** == maxlevel) {

print “max level reached, no connection”

return null

} else {

for all **actor** in actorfilmlist: // see if any of those actors is connected to actor2 {

foundactor = Check( **actor**, **actor2**, **level** + 1)

if (foundactor != null) { // relation between **actor** and **actor2** via **foundactor**

find any film **film** for **actor** within the actorfilmlist

print “**actor1** was in **film**  with **foundactor”**

return **actor**

}

}

return null;

}

## Microservice architecture for the IMDB application

The challenge was to modernize the IMDB java application using the modern views of application design in Java. This leads us to the following requirements:

1. Use github for version control management
2. Use Docker image(s) to hold the database instead of a preinstalled mySQL database
3. Port the IMDB application to a Spring Boot version
4. Create independence of datasource using JPA and services provided by Spring Boot

In theory, any relational database or even nonrelational database could be used. As a demo, the application is connected to either mySQL in a Docker image as well as connecting to a local Postgres database on the laptop has been created.

1. Development environment and production environment  
   Use different environments with (potentially) different databases. It should be possible to configure the actual environment used by setting an environment variable. Configuration files within the application should determine what to use and how. By the use of the environment variable “MYENV” it is controlled which environment is used (export MYENV = “prod” to use the production environment, export MYENV= “dev” to use the development environment.
2. Use of microservices instead of a monolithic application. Perhaps as separate distributables, at least it should be possible to upgrade them separately, have several instances deployed, monitor them separately, etcetera
3. Create logging, monitoring and regression tests for each of the services.  
   Regression tests should on deploy of course only be run in the development environment.
4. Deploy the services on Heroku and make sure the services each have their own monitoring and maintenance facilities.

The requirements 1..5 are implemented in the IMDB003 database application, currently in Github.  
Requirement 6 is underway: the design and the relations between the microservices will be presented below.

Requirement 7. is partly underway: for logging, LogBack will be used (<https://www.baeldung.com/logback>) for logging and – partly – for monitoring. Junit tests will be used to implement the regression tests. For monitoring of the service performance and the liveliness of the, the off-the-shelf services provided by Heroku will be used.

## Microservice architecture

Looking at the current application, we are able to create categories of queries which should provide the required functionality. The query categories can be stacked.

Logically, functionally, we have the need of four required categories of services returning information via queries on the datasets. The lowest levels are “independent” services running on only one dataset, lower level services are used by higher level services.

NB import files are to be found in

D:\IMDB\_data

#### 1. Actorservice

Given an actor key or an actor name, return the actor(s) records. Perhaps restrict the result set to persons for whom the birthday is not null. Uses only dataset Actor

***2. Film service***

Given a film key or a film name, return the film(s) records. Uses only dataset Film.

#### 3. ActorFilmService

Given a film key, return the actor keys of persons having an actor role in this film.  
Given an actor key, return the film keys of films the person was an actor in.  
Uses dataset Film-Actor only.

#### 4. CoincidenceService

Given two actor keys, return a list of films they were both in as actor. Uses ActorFilm Service (to obtain the film list), Actor Service and Film Service (to obtain the printable characteristics of the actors and the films)

#### 5. Degree of Separation service

Given two actor keys, return a structure holding film keys and actor keys connecting the two actors. The data structure has a practical example of the degree of separation between the two persons.

Uses the Coincidence service (to determine the sample path between the actors), uses Actor Service and Film Service (to obtain the printable characteristics of the films and actors encountered on the sample path).

## Useage of the services to implement the application

How are those requirements used in the application: the requirements for the applications are listed as an example:

1. Typecasting an actor  
Given an actor name, use the ActorService to obtain his key and his details. Use the film service to obtain the list of film records and use the field “genre” to obtain the genres and their count. Find the most frequent occurring genre – this is the typecast of this actor. Show the actor details and his typecast, perhaps the number of films he was in.

More details about the films and crew of these films for this actor:

In a more “Deluxe” version also list the films of this actor and the crew of these films:

- use the Actor service to retrieve the film keys, for every film key use the Film service to retrieve the film details.  
Use the ActorFilm service to retrieve the actor keys having a role in this film, for every actor thus found retrieve the actor details using the Actor service.  
  
  
2. Finding coincident films between actors  
Use the Actor service to obtain the actor key and the actor details for both actors. For both actors, use the actor-film service to obtain a list of films having this person as “actor” (nb. This is a piece of information inside the Actor Film table).  
Make a set of common film keys.   
List the actor details for both actors and the list of films.

Use the film service to obtain the film details for each film.

In the more “Deluxe” version also list the crew for each film using the ActorFilm service and the Actor service.

3. Determining degree of separation

Using the ActorFilm service only, determine the degree of between two actors including a sample path by creating a special purpose data structure having film keys and actor keys.  
Use the datastructure and both the Actor service and the Film service to list the specifics of the sample path.

Load balancing and resource management:

The degree of separation is – in principle – a quite simple recursive algorithm on one dataset only. It is well possible to create more than one instance of this service in order to solve the request. The bulk of the “work” consists of creating the datastructure, not of the simple task of creating the information to list details of the specific films or the specific actors.  
This service may be quite demanding in terms of memory, performance and database useage of just this one table.  
Therefore, it is a perfect candidate to be replicated. Perhaps it is even a good idea to have it query a separate database for just this table.