

Vorwissenschaftliche Arbeit

Monolithic vs. Microservices: a comparison based on the frameworks Flask and Django

Verfasst von: Benedict Armstrong

Klasse: 8A

Schuljahr: 2017/18

Betreuung durch: Mag. Christian Schöbel

Wien, Februar 2018

Abstract

{Text}

Vorwort

{Text}

Table of Contents

[1 Introduction 4](#_Toc502254108)

[2 Monolithic Architecture 5](#_Toc502254109)

[2.1 Definition 5](#_Toc502254110)

[2.2 Historical Perspective 5](#_Toc502254111)

[3 Microservices Architecture 5](#_Toc502254112)

[3.1 Definition 5](#_Toc502254113)

[3.2 Historical Perspective 5](#_Toc502254114)

[4 Monolithic and Microservice Architectures in Practice 5](#_Toc502254115)

[4.1 Cinema Seat Reservation Application 6](#_Toc502254116)

[4.1.1 Requirements 6](#_Toc502254117)

[4.1.2 Structure 6](#_Toc502254118)

[4.1.3 Problems 6](#_Toc502254119)

[4.1.4 Implementation 6](#_Toc502254120)

[4.2 Get showing movies 7](#_Toc502254121)

[4.2.1 Microservices 7](#_Toc502254122)

[4.2.2 Monolithic 8](#_Toc502254123)

[4.3 Microservices 9](#_Toc502254124)

[4.3.1 Database Operations 9](#_Toc502254125)

[4.3.2 Rest Server 9](#_Toc502254126)

[4.3.3 Open API with connexion 9](#_Toc502254127)

[4.3.4 Client-side 9](#_Toc502254128)

[4.3.5 Design 9](#_Toc502254129)

[4.4 Monolithic 9](#_Toc502254130)

[4.4.1 Database Operations with Django Database API 9](#_Toc502254131)

[4.4.2 Integrating Legacy Database in Django 9](#_Toc502254132)

[4.4.3 Views 9](#_Toc502254133)

[4.4.4 Forms and working with Django Framework 9](#_Toc502254134)

[4.4.5 Design 9](#_Toc502254135)

[5 Comparitive Advantages and Disadvantages 9](#_Toc502254136)

[5.1 Monolithic Architecture 9](#_Toc502254137)

[5.1.1 Development concerns 9](#_Toc502254138)

[5.1.2 Production concerns 10](#_Toc502254139)

[5.2 Microservices Architecture 10](#_Toc502254140)

[5.2.1 Development concerns 10](#_Toc502254141)

[5.2.2 Production concerns 10](#_Toc502254142)

[6 Conclusion 10](#_Toc502254143)

[7 Sources 10](#_Toc502254144)

# Introduction

{Text}

# Monolithic Architecture

## Definition [1],[2]

Monolithic architecture in general describes software where the user interface (UI) and the business logic are combined on a single platform. Similarly, in webdevelopment terms it referes to a model for the design of software in which the application is composed all in one piece. The UI is generated as a so called “View” on the server and sent out finished in one piece to the user. Typically, the components are interconnected and all of these components need to be present for code to be executed or compiled. A framework is not required but it greatly eases the development so for this paper I have chosen to uses Django, developed and maintained by the Django Software Foundation.

## Historical Perspective

The first applications to be developed for mainframes in the early 50’s up to the 70’s were monolythic. Only with the standardisation of TCP wide spread networking became possible and with that microservices as we know them.

# Microservices Architecture

## Definition

Microservice architecture describes the logical structure and design of a software application which consists of modular, loosely coupled components. These Services should be lightweight, independent and simple. From a web development point of view the key difference, compared to a monolithic architecture, is the separation of the presentation layer and the business logic, that means that the UI is composed in the browser on the client’s machine instead of the server. Applications like Netflix use this to reduce server load which dramatically reduces costs for the company. As with Monolithic architecture there are many different frameworks to ease the development process and for this paper I have chosen to use Flask which is open source and can be found on Github.

## Historical Perspective

The term Microservice was first used to describe this style of application design in 2011 by a workshop of software architects near Venice and has since gained popularity.

{I need to do further research on the topic as there are a lot of conflicting sources out there}

# Monolithic and Microservice Architectures in Practice

In order to compare Monolithic and Microservice Architectures I have decided to get some first-hand experience and write my own Application. To highlight the differences, I will go through a call for the movies which are currently showing with each of the two methods.

## Cinema Seat Reservation Application

As a practical real-world example, I have created a web app for reserving seats in a cinema using both approaches. Their user interface is almost identical and kept as simple as possible.

### Requirements

The User should be able to pick a Movie from a List and then reserve seats for a specific Time and Date. The System should also support adding and the removal of movies by an administrator. For simplicity, the authentication and authorization were ignored.

### Structure



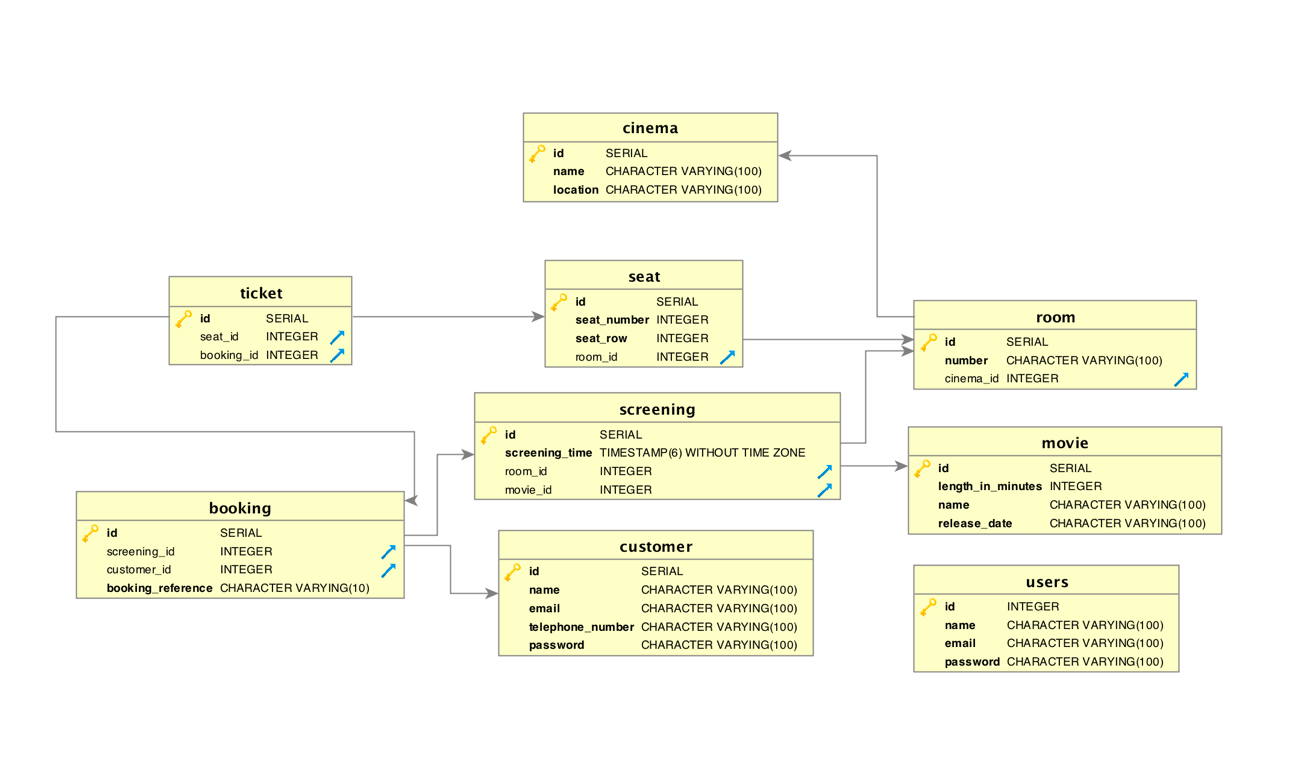
()

### Problems

{Include Section?}

### Implementation

#### Database



For this application, I have chosen to keep the database as simple and straight forward as possible. I am using a Postgres 10 database structured as seen in this Picture.

#### Userstories

{include Section?}

## Get showing movies

I will go through an example request, using both approaches, which gets all movies currently showing in a cinema.

### Microservices

For the client-side I am using the JavaScript and the jQuery library, the server-side coded in Python using the Flask and Connexion Frameworks.

#### Client-side

1. $.ajax({
2. url: "http://127.0.0.1:5000/movies",
3. success: **function**(result) {
4. **for** (**var** i = 0; i < result["showing\_movies"].length;i++) {
5. **var** movie = result["showing\_movies"][i];
6. paint\_movie(movie);
7. };
9. }, error: **function**(xhr) {
10. alert("Error (" + xhr.status + ") :  " + xhr.statusText);
11. }
12. });

The initial request is triggered upon loading of the webpages with the jQuery *ready()* function which executes once the whole document has been loaded in the browser. The *$.ajax()* function shown above is nested within this function. This ajax function performs a http request to *http://127.0.0.1:5000/movies* which is a request to the server, localhost in this case, on port 5000 which it is listening on. The function given as the *success* parameter is executed as the name suggests after a successful request, same with the *error* parameter. In the success function, we process the received *json* data and paint it on the website. The Error function alerts the user of an error should one occur.

#### Server-side

On the server-side the request is first caught by connexion, it filters for the request type and has fixed returns. Connexion then calls the *showing\_movies()* function in controllers/movies.

1. **def** showing\_movies():
2. movies = get\_movies\_showing()
3. **if** (movies **is** None):
4. **return** "No movies found", 404
5. **else**:
6. **return** {'showing\_movies': movies}

This takes the return of the *get\_movies\_showing()* function from the dao and sends it back to Connexion which converts it to json and sends it out.

1. **def** get\_movies\_showing():
2. with get\_db\_cursor() as cursor:
3. cursor.execute("""select \* from movie where id in (select movie\_id from screening);""")
4. **return** cursor.fetchall()

This function performs the actual operation on the database, in this case fetching all the showing movies and returning them.

### Monolithic

Here I am using the Django framework and showing a function from the *views* file. In this case there is no need for routing as there are no requests from outside coming in.

#### Client-side

1. **def** index(request):
2. showing\_movies\_list = (Movie.objects.filter(
3. id\_\_in=[s.movie\_id **for** s **in** Screening.objects.all().distinct()]))
4. context = {'showing\_movies\_list': showing\_movies\_list}
5. **return** render(request, 'cinema/index.html', context)

In the *index()* function, which is one of the views, the showing movies are fetched from the database and passed to the *render()* function, which renders the template, as the context for painting the view.

## Microservice

Overview of the key Microservices in the Cinema Seat Reservation Application.

### Database Operations

These operations are handled by the DAO, short for database operations an example function in the DAO might look like this:

1. **def** get\_movie(movie\_id):
2. with get\_db\_cursor() as cursor:
3. cursor.execute("""select \* from movie where id=%s;""", [movie\_id])
4. **return** cursor.fetchone()

The *get\_movie()* function fetches a movie by its id. This function gets called by the controllers.

### Controllers

The controllers process the data received from the DAO functions.

1. **def** info\_movie(movie\_id):
2. movie = get\_movie(movie\_id)
3. **if** (movie **is** None):
4. **return** "Movie %s not found" % (movie\_id), 404
5. **else**:
6. **return** movie

In this example, the info\_movie() function calls the DAO function from above and checks weather there is a movie returned and decides what response connexion should send back. If there is no movie a http 404 error, the infamous *“Not found”,* is returned if there is no movie with a matching id in the database otherwise it just returns the movie.

### Flask

Flask handles the http requests and responses coming from outside. With Flask it is pretty easy to do routing and processing combining the routing and the controller into one function, like in the example below which returns all of the users in the database.

1. @app.app.route('/users/all', methods=['GET'])
2. **def** get\_users():
3. users = get\_all\_users()
4. **return** jsonify({'users': users})

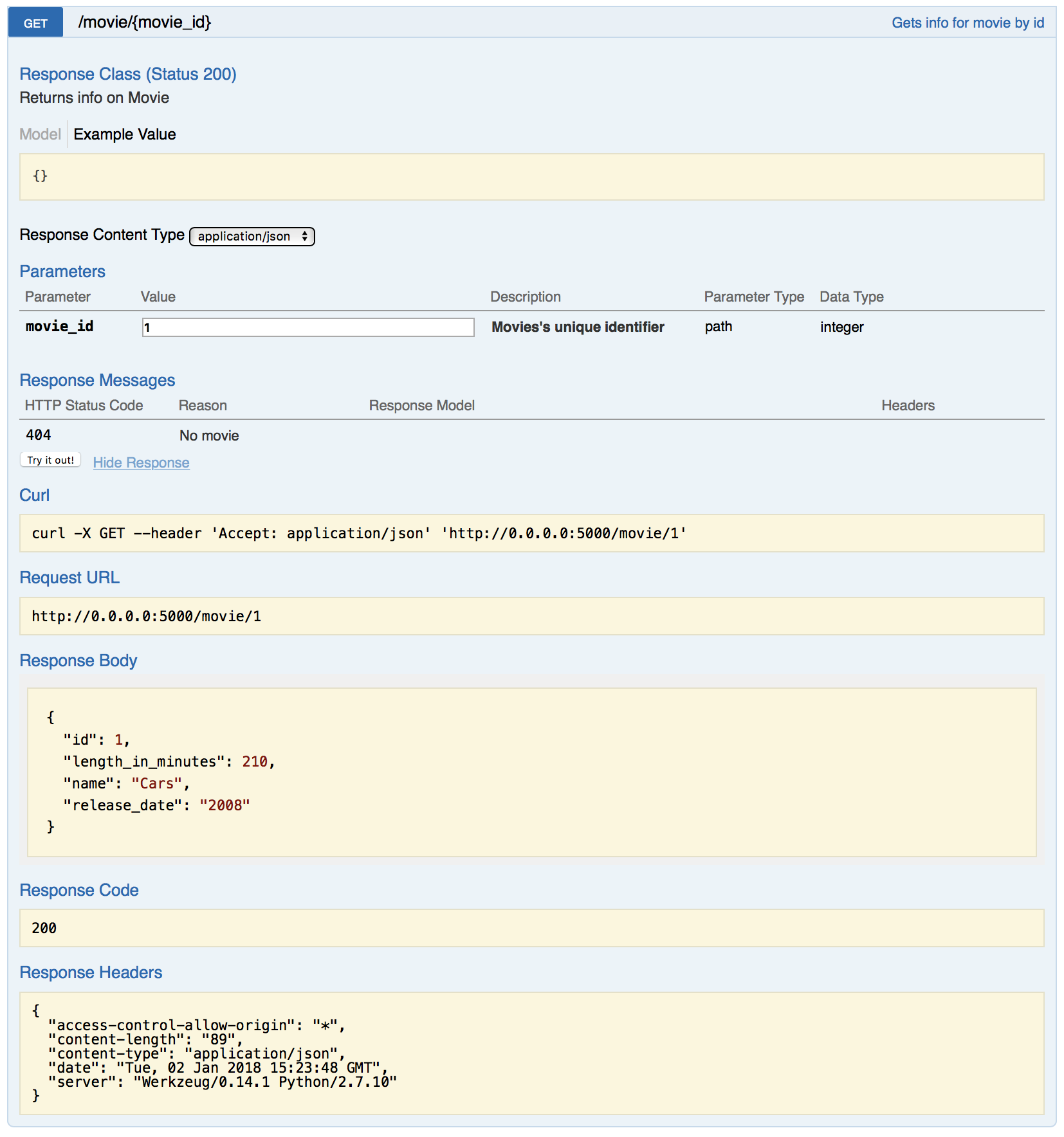
So, with Flask it is possible to leave out the controller or rather compact it but that causes confusion and especially for larger projects I think it is easier to split it up.

### Open API with connexion

That’s why this application uses connexion on top of Flask. Connexion uses a .yaml file which defines the API. In this file, every request is defined with a path, contents and responses. The content of the request can be in the body, in the request header or in the URL or a combination of all. It is good practice to give every request a response, normally a http 200 for a successful request but sometimes something different like a 404 if there is nothing found for a query.

1. /movies:
2. get:
3. tags: [movies]
4. operationId: controllers.movies.showing\_movies
5. summary: Gets all of the showing movies
6. responses:
7. 200:
8. description: Returns showing movies
9. schema:
10. type: array
11. items:
12. $ref: '#/definitions/Movie'
13. 404:
14. description: No movies

Connexion also uses Open API, previously known as swagger, which gives you a visual representation of the API with handy tools for testing and debugging. This makes it very easy for someone without any knowledge of the underlying business logic to write a client for the application as everything you need to know for a request is precisely detailed in the yaml file and neatly visualised, with examples to try it out, by Open API.



Example of a request detailed in Open API

### Client-side

### Design

## Monolithic

### Database Operations with Django Database API

### Integrating Legacy Database in Django

### Views

### Forms and working with Django Framework

### Design

# Comparitive Advantages and Disadvantages

This Chapter will elaborate the Advantages and Disadvantages of either method. They are in no particular order and as subjective as possible.

## Monolithic Architecture

Starting with Monolithic Architecture and the concerns I was faced with during development and future production concerns

### Development concerns

#### Learning curve and Development time

Learning monolithic programming might seem simple in the beginning especially with a framework like Django and you can have a functioning web application running in no time and without prior knowledge of the framework and concept but that can be deceiving. Even though you might be able to write a simple application quickly, you’ll most likely lack an understanding of the processes going on under the hood which will eventually lead to problems down the road when you have to scale up the application or customize it further.

### Production concerns

#### Performance

#### Scalability

#### Security

#### Upgradeability

#### Crossplatform

## Microservices Architecture

### Development concerns

#### Learning Curve

#### Teamwork

### Production concerns

#### Performance

#### Scalability

#### Security

#### Upgradeability

#### Crossplatform

# Conclusion

# Sources

Django Software Foundation, 2017. *Django.* [Online]   
Available at: https://www.djangoproject.com  
[Accessed 15 12 2017].

Margaret, R., n.d. *Definition, monolithic architecture.* [Online]   
Available at: http://whatis.techtarget.com/definition/monolithic-architecture  
[Accessed 15 12 2017].

Opensource, 2017. *Flask.* [Online]   
Available at: http://flask.pocoo.org  
[Accessed 31 12 2017].