

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](#_Toc502921544)

[2 Monolithic Architecture 5](#_Toc502921545)

[2.1 Definition 5](#_Toc502921546)

[2.2 Historical Perspective [1] 5](#_Toc502921547)

[Microservices Architecture 6](#_Toc502921548)

[2.3 Definition Microservices 6](#_Toc502921549)

[2.4 Historical Perspective 7](#_Toc502921550)

[2.4.1 Service-oriented Computing 7](#_Toc502921551)

[2.4.2 Second generation of services 7](#_Toc502921552)

[2.4.3 SOA with Web Services 8](#_Toc502921553)

[3 Monolithic and Microservice Architectures in Practice 8](#_Toc502921554)

[3.1 Cinema Seat Reservation Application 8](#_Toc502921555)

[3.1.1 Requirements 8](#_Toc502921556)

[3.1.2 Structure 8](#_Toc502921557)

[3.1.3 Problems 9](#_Toc502921558)

[3.1.4 Implementation 9](#_Toc502921559)

[3.2 Get showing movies 9](#_Toc502921560)

[3.2.1 Microservices 9](#_Toc502921561)

[3.2.2 Monolithic 10](#_Toc502921562)

[3.3 Microservice 11](#_Toc502921563)

[3.3.1 Database Operations 11](#_Toc502921564)

[3.3.2 Controllers 11](#_Toc502921565)

[3.3.3 Flask 11](#_Toc502921566)

[3.3.4 Open API with connexion 12](#_Toc502921567)

[3.3.5 Client-side 13](#_Toc502921568)

[3.3.6 Design 13](#_Toc502921569)

[3.4 Monolithic 13](#_Toc502921570)

[3.4.1 Database Operations with Django Database API 14](#_Toc502921571)

[3.4.2 Integrating Legacy Database in Django 14](#_Toc502921572)

[3.4.3 Views 14](#_Toc502921573)

[3.4.4 Forms and working with Django Framework 14](#_Toc502921574)

[3.4.5 Design 14](#_Toc502921575)

[4 Comparitive Advantages and Disadvantages 14](#_Toc502921576)

[4.1 Monolithic Architecture 14](#_Toc502921577)

[4.1.1 Development concerns 14](#_Toc502921578)

[4.1.2 Production concerns 15](#_Toc502921579)

[4.2 Microservices Architecture 15](#_Toc502921580)

[4.2.1 Development concerns 15](#_Toc502921581)

[4.2.2 Production concerns 15](#_Toc502921582)

[5 Conclusion 15](#_Toc502921583)

[6 References 15](#_Toc502921584)

# Introduction

https://msdn.microsoft.com/en-us/library/ee658098.aspx

# Terminology

|  |  |
| --- | --- |
| XML | eXtensible Markup Language |
| JSON | **J**ava**S**cript **O**bject **N**otation |
| CRUD | **C**reate **R**etrieve **U**pdate **D**elete |
| SOA | **S**ervice **O**riented **A**rchitecture |
|  |  |
|  |  |
|  |  |
|  |  |

# Monolithic Architecture

Traditional, mainstream serverside development languages like C++, Java and Python offer ways to abstract a complex program into modules, breaking down the complexity. However, these languages are designed for the creation of single executable artefacts, also called monoliths which rely on the sharing of resources of the same machine (memory, databases, files), which in turn means that they cannot be executed independently [1].

## Definition

Monolithic architecture, architecture using monoliths, in general describes software whose modules cannot be executed seperately [1]. As the adjective monolithic, which describes something that is “cast as a single piece” [2] implies it is all one entity. 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 again all of these components need to be present for code to be executed or compiled [3, 4].



Img. 1, Possible structure of a basic monolithic web application [5]

## Historical Perspective [1]

The development of large-scale applications in the early 60’s up to the 70’s lead to the first problems and confrontations with Software design and it’s implications on the development process. The 1970s saw a big increase in references to software design in scientific research and general interest but a solid foundation of the topic was only established by Perry and Wolf in their 1992 book “Foundations for the study of software architecture.”. The advent and diffusion of object-orientation, starting in the 1980s and in particular in the 1990s, brought its own contribution to the field of Software Architecture. A typical example of an architectural design pattern in object-oriented programming is the Model-View-Controller (MVC), which I have used in the development of the cinema seat reservation application [§3.4 ] and is visualized in the image above.

# Microservices Architecture

Microservice architecture is software design philosophy which describes the logical structure of a software application consisting of microservices.

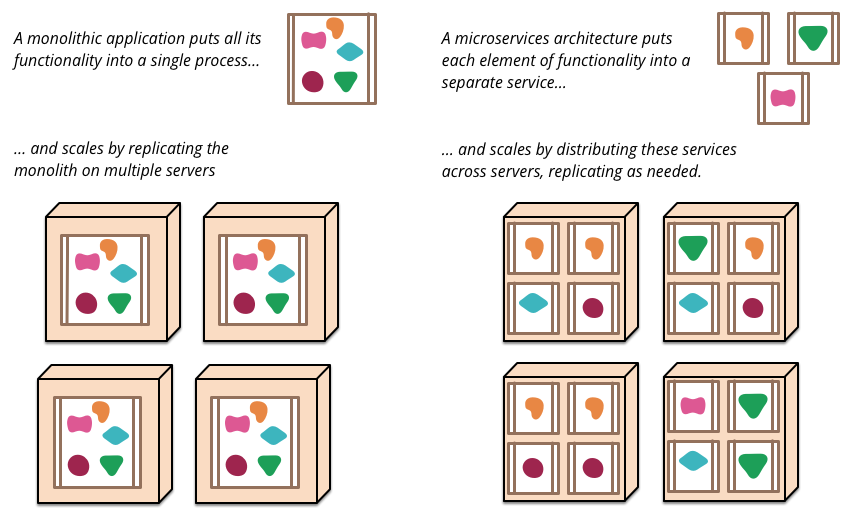
## Definition Microservices

Microservices are modular, loosely coupled components which are lightweight and simple. They should take an input and give back a predictable output. How it calculates the output shouldn’t matter, in other words the programming language, the design and the calculations of the service should be replaceable but with a consistent output. [6]



Img. 2, Example of a simple applications consisting of microservices [1]

Consider as an example a microservice which solves equations, so when it receives an input equation it gives back the answers. It should not however do other things like plot the resulting function. That should again be handled by a different independent service which can call upon the first module for the calculations. From a web development point of view one of the key differences, 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.



Img. 3, Comparison of Monoliths and Microservices [7]

## Historical Perspective

Attention to *separation of concerns* led to Component-based software engineering which has given better control over design, implementation and evolution of software systems. The last decade has seen a further shift towards the concept of services first [8] and the natural evolution to microservices afterwards. [1]

The term “microservices” was first coined at an architectural conference in Venice in 2011 as way to describe the approach of developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [7]. Netflix one of the pioneers of microservice architecture describe its architecture as a “fine grained SOA” [9].

### Service-oriented Computing

Service-oriented computing is a paradigm where a program — called a service — offers functionalities to other components, accessible via message passing.

### Second generation of services

Service-oriented architecture (SOA) is an architectural style for building software applications that use services available in a network such as the web. It promotes loose coupling between software components so that they can be reused. Applications in SOA are built based on services. A service is an implementation of a well-defined business functionality, and such services can then be consumed by clients in different applications or business processes. [10]

In other words: services decouple their interfaces (i.e. how other services access their functionalities) from their implementation [1].

### SOA with Web Services

Web services are software systems designed to support interoperable machine-to-machine interaction over a network [10]. A popular approach to this style is REST which I am using to build the cinema seat reservation application.

{include reference to rest section}

# 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



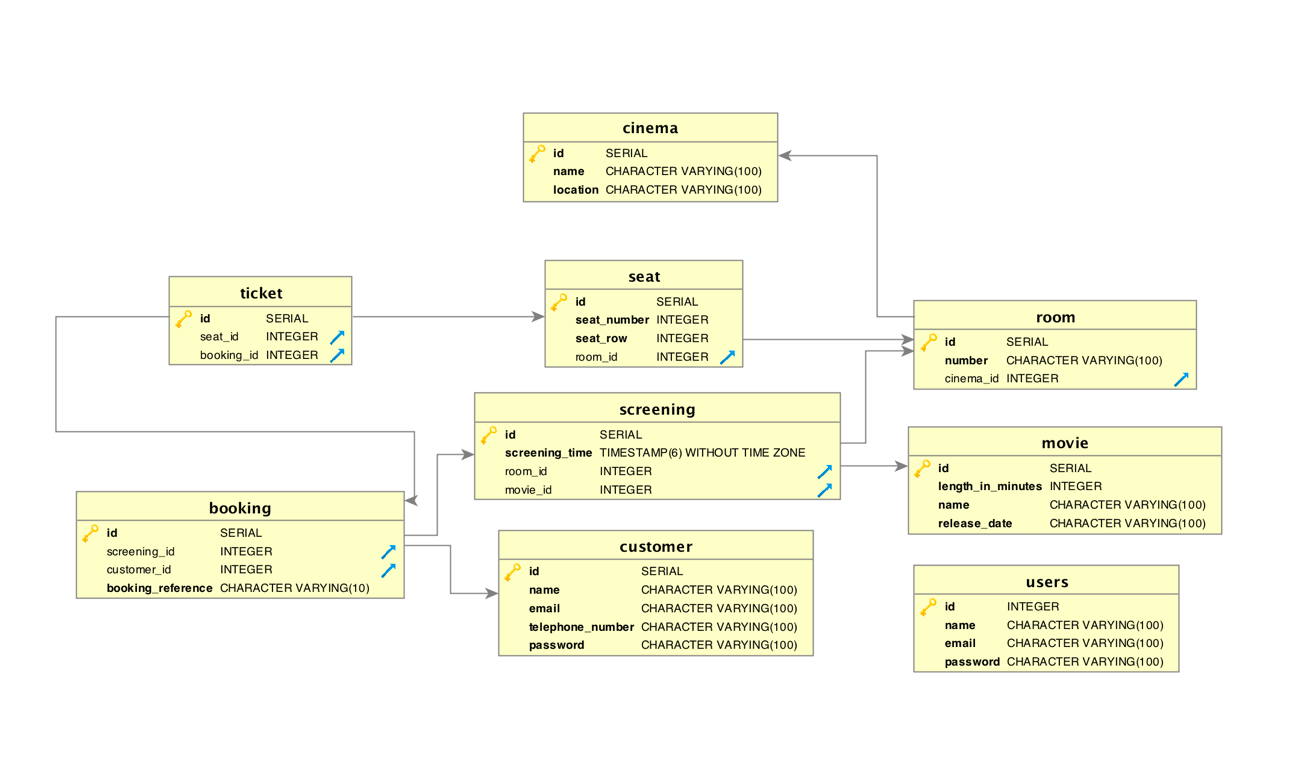
Img. 4, Basic structure of the Cinema Reservation Application [5]

### Problems

{Include Section?}

### Implementation

#### Database



Img. 5, Database Structure

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 the image above.

## Get showing movies

To give a better understanding of the application and the structure I will go through an example request, using both approaches, which gets all movies currently showing in a cinema.

### Microservices

On the client-side the application uses the JavaScript and the jQuery libraries, the server-side is 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 that the API server is listening on for http requests. 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 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

This section will give a brief overview of the technologies used in the application.

### Representational State Transfer (REST) [8, 11]

This Application was written using a REST web service architecture. REST is a software architecture defined by Roy Thomas Fielding in his PhD thesis from the year 2000, “Architectural Styles and the Design of Network-based Software Architectures”. The REST Web is the subset of the WWW (based on HTTP) in which agents provide uniform interface semantics -- essentially create, retrieve, update and delete -- rather than arbitrary or application-specific interfaces, and manipulate resources only by the exchange of representations. Furthermore, the REST interactions are "stateless" in the sense that the meaning of a message does not depend on the state of the conversation. Objects in the system, called resources, with Uniform Resource Identifiers (URIs) can be manipulated with the operations above, commonly known as “CRUD” operations. REST is often used with the HTTP web protocol and JSON (or XML) as the data format.

### 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})

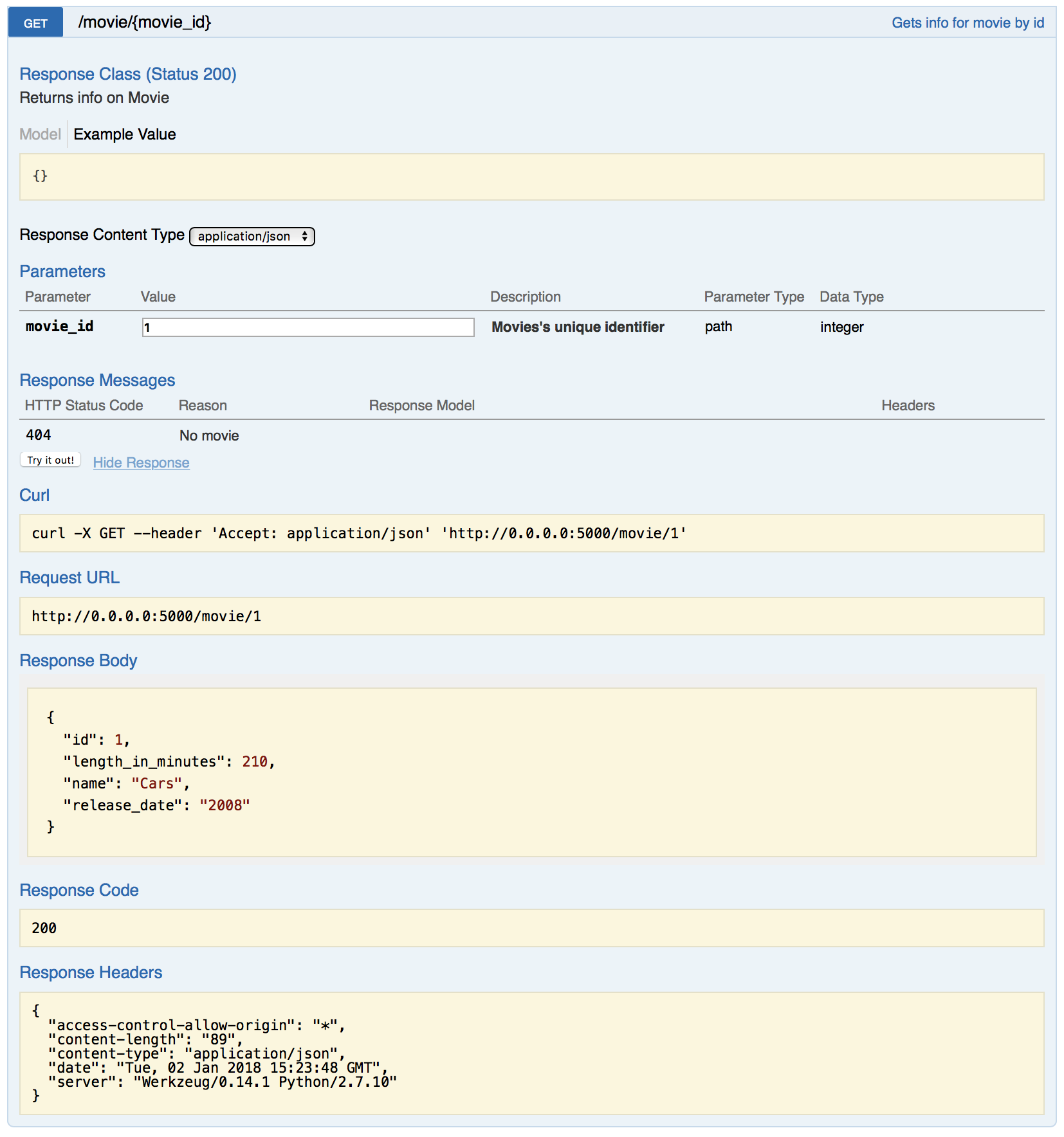
So, with Flask it is possible to leave out the controller or rather compact it but that causes confusion and especially for larger projects I think it is easier to split it up.

### 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.



Img. 6, Example of a request detailed in Open API

### Client-side

On the client-side requests to the API are made with the jQuery *ajax()* function. It gets a json file from the server, converts it to a JavaScript object and passes that to the function in the success parameter.

## Monolithic

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.

### Model-View-Controller

The application is built upon the principles of the Model-View-Controller (MVC) design pattern. “MVC is a pattern used to isolate business logic from the user interface. Using it, the Model represents the information (the data) of the application and the business rules used to manipulate the data, the View corresponds to elements of the user interface such as text, checkbox items, and so forth, and the Controller manages details involving the communication between the model and view. The controller handles user actions such as keystrokes and mouse movements and pipes them into the model or view as required.” [12]

### Database Operations with Django Database API

Django provides a neat and easy to use Database API with which SQL statements like this:

1. **select** \* **from** movie **where** id in (**select** movie\_id **from** screening);

Look like this:

1. showing\_movies\_list = (Movie.objects.filter(id\_\_in=[s.movie\_id **for** s **in** Screening.objects.all().distinct()]))

{Include Section?}

### Integrating Legacy Database in Django

{Include Section?}

### Views

### Forms and working with Django Framework

### Design

# Comparitive Advantages and Disadvantages

This chapter will elaborate on the advantages and disadvantages of either method.

## Monolithic Architecture

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

### Development concerns

Concerns a developer or a team of developers is faced with when developing a monolithic application.

#### Learning curve

Learning monolithic programming might seem simple in the beginning especially with a framework like Django and one can have a functioning web application up and running in no time and without prior knowledge of the framework and concept. This doesn’t come without some negative side-effects though, 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. This may eventually lead to problems down the road when you have to scale up the application or customize it further. Django and comparable frameworks are quite opinionated which can be challenging if you are trying to do things “your own way”. The pre-existing code and structure almost forces you, except if you have an excellent knowledge of the framework to adhere to a very strict, predefined way of doing things. This is especially bad for people used to working with other framework which impose different opinions.

#### Debugging and Development

Django offers handy tools and helpers for debugging but nevertheless it can quickly become tedious and time consuming due to the structure of the application. Bugs can be hidden multiple layers and dependencies down. To make a new addition or changes the whole system has to be brought down, rebuilt and restarted leading to delays in the development time, especially with bigger systems. Changes propagate through the whole system and can break otherwise working code making it hard to change things after they have been implemented.

The more Complex a system gets the harder it is to understand and due to the tight coupling and dependencies it can lead to, what in software development is sometimes referred to as, a “big ball of mud” where no single developer or even a group understands the application as a whole [13]. Due to the lack of a standard interface parts of a monolithic application only offer limited reusability.

Simple small-scale applications evade many of these problems. They are simple to deploy especially with a framework like Django which requires one command to build the code and start a server. Updating the code

### Production concerns

#### Performance

Frameworks usually offer pretty good optimisation and especially for simple applications performance is high.

#### Scalability

#### Security

#### Upgradeability

#### Crossplatform

## Microservices Architecture

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.

### Development concerns

#### Learning Curve

#### Teamwork

### Production concerns

#### Performance

#### Scalability

#### Security

#### Upgradeability

#### Crossplatform

# Conclusion

# References

|  |  |
| --- | --- |
| [1] | S. G. A. L. L. M. M. F. M. R. M. L. S. Nicola Dragoni, “Microservices: yesterday, today, and tomorrow,” 2017. [Online]. Available: https://arxiv.org/pdf/1606.04036.pdf. [Accessed 3 January 2018]. |
| [2] | Merriam Webster, “monolithic,” 2017. [Online]. Available: https://www.merriam-webster.com/dictionary/monolithic. [Accessed 3 January 2018]. |
| [3] | Wrox Press Ltd, “Three-tier Application Model,” 1998. [Online]. Available: https://msdn.microsoft.com/en-us/library/aa480455.aspx. [Accessed 3 Janurary 2018]. |
| [4] | R. Margaret, “Definition: monolithic architecture,” 2016. [Online]. Available: http://whatis.techtarget.com/definition/monolithic-architecture. [Accessed 14 December 2017]. |
| [5] | R. Annett, “What is a Monolith?,” 2014. [Online]. Available: http://www.codingthearchitecture.com/2014/11/19/what\_is\_a\_monolith.html. [Accessed 3 January 2018]. |
| [6] | E. Wolff, “Microservices Primer,” learnpub, 2017. [Online]. Available: https://leanpub.com/microservices-primer/read. [Accessed 3 January 2018]. |
| [7] | M. F. James Lewis, “Microservices, a definition of this new architectural term,” 25 March 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed 5 January 2018]. |
| [8] | H. H. F. M. E. N. M. C. ,. C. F. ,. D. O. David Booth, “Web Services Architecture,” W3C, Febuary 2004. [Online]. Available: https://www.w3.org/TR/ws-arch/#introduction. [Accessed 4 January 2018]. |
| [9] | S. T. Allen Wang, “Announcing Ribbon: Tying the Netflix Mid-Tier Services Together,” 28 January 2013. [Online]. Available: https://medium.com/netflix-techblog/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62. [Accessed 5 January 2018]. |
| [10] | Q. H. Mahmoud, “Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI),” April 2005. [Online]. Available: http://www.oracle.com/technetwork/articles/javase/soa-142870.html. [Accessed 5 January 2018]. |
| [11] | R. T. Fielding, “Representational State Transfer (REST),” 2000. [Online]. Available: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest\_arch\_style.htm. [Accessed 5 January 2018]. |
| [12] | rj45, “Simple Example of MVC (Model View Controller) Design Pattern for Abstraction,” 8 April 2008. [Online]. Available: https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design. [Accessed 5 January 2018]. |
| [13] | J. Y. Brian Foote, “Big Ball of Mud,” 26 June 1999. [Online]. Available: http://www.laputan.org/mud/. [Accessed 5 January 2018]. |
| [14] | Django Software Foundation, “Django,” 2017. [Online]. Available: https://www.djangoproject.com. [Accessed 15 12 2017]. |
| [15] | Opensource, “Flask,” 2017. [Online]. Available: http://flask.pocoo.org. [Accessed 31 12 2017]. |