IMDb (Part 3): From Databases to the Web

Big Data Engineering
(formerly Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

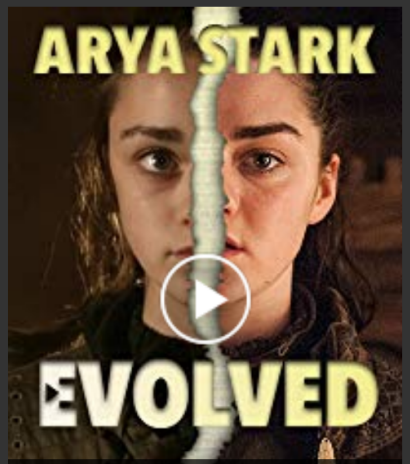
June 29, 2023

IMDb (Part 3): From Databases to the Web

Planned structure for each two-week lecture:

1. Concrete application: IMDb
2. What are the ~~data management and~~ **security and Web development** issues behind this?
3. Basics to be able to solve these problems
   (a) Slides
   (b) Jupyter/Python/SQL Hands-on
4. Transfer of the basics to the concrete application

Shazam! Soars to the Top
Weekend Box Office

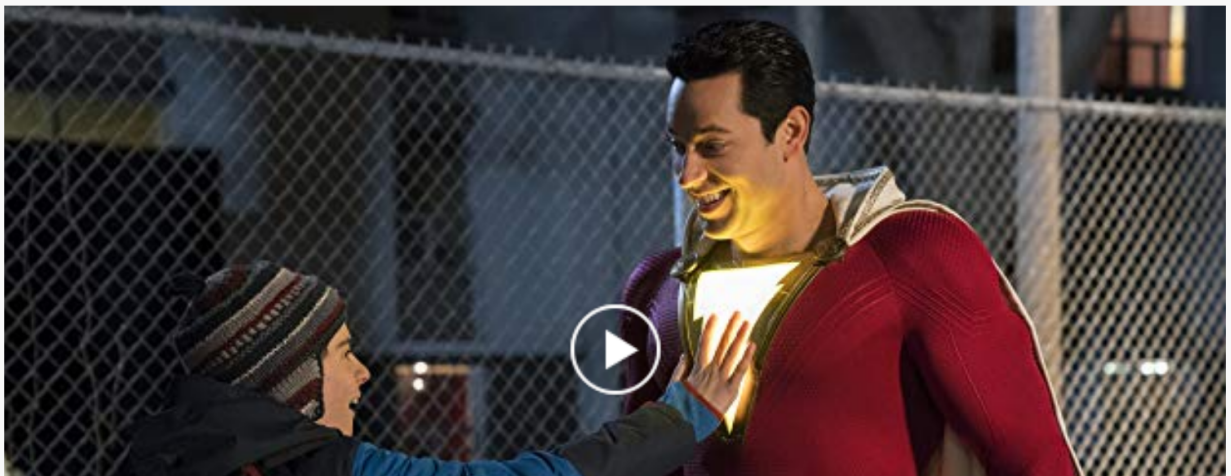


The Evolution of Arya Stark
From Misfit to Assassin

From 'Holby City' to 'Killing Eve'
The Rise of Jodie Comer

Browse trailers »

Zachary Levi Reveals His 'Shazam!' Suit Struggles



Opening This Week

- Hellboy - Call of Darkness
- Mister Link - Ein fellig verrücktes Abenteuer
- After Passion
- Little
- High Life
- Les filles du soleil
- Sauvage — Opens Apr. 10

See more opening this week »

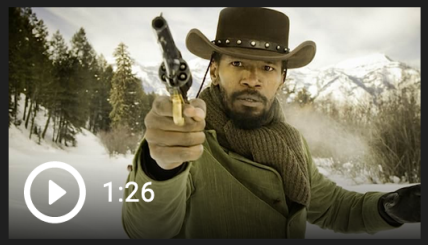
Get Showtimes »

Now Playing (Box Office)

Shazam! $53.5M	Showtimes
Friedhof der Kuscheltiere $25.0M	Showtimes
Dumbo $18.2M	Showtimes

IMDb Menu All django unchained IMDbPro Watchlist Sign In EN

Django Unchained
2012
Jamie Foxx, Christoph Waltz


1:26
Winner: Best Supporting Actor


2:06
International Version #2

Scenes from Django Unchained - UK Winner
2013
Andi Osho, Struan Rodger

Django Unchained: Pai Mei's Talk with Broomhilda
2012
Artem Mishin, Zivia Walker

Remembering J. Michael Riva: The Production D...
2013
Jamie Foxx, Reginald Hudlin

Double Twins Do Django Unchained
2014
Lionel Brugeaud, Paul Philip Clark

Django Unchained. Poetic Version
2022

Django & Django

Up next

1:07
"Silo" Finale Exclusive Clip
Streaming June 30 on Apple TV+

4:01
Heroes or Villains? The "Secret Invasion" Cast Weigh-In
Watch the Interview

0:46
'Priscilla'
Watch the Trailer

Browse trailers ›

Jetflix 1:01

Featured toc

TOM HANKS "ASTEROID CITY"

Django Unchained

2012 · R · 2h 45m

IMDb RATING 8.4/10 1.6M

YOUR RATING Rate

POPULARITY 129 ▼23

28 VIDEOS

99+ PHOTOS

Drama Western

With the help of a German bounty-hunter, a freed slave sets out to rescue his wife from a brutal plantation owner in Mississippi.

Director Quentin Tarantino

Writer Quentin Tarantino

Stars Jamie Foxx · Christoph Waltz · Leonardo DiCaprio ›

Watch on Prime Video
rent/buy from EUR2.99

More watch options

Add to Watchlist
Added by 981K users

1.8K User reviews 663 Critic reviews

Django Unchained (2012)

Edit

Full Cast & Crew

IMDbPro See agents for this cast & crew on IMDbPro

Actually...

Well, nice website, but: so far we did not look at how to get the data from the database and use it to generate a website.

Directed by

Quentin Tarantino

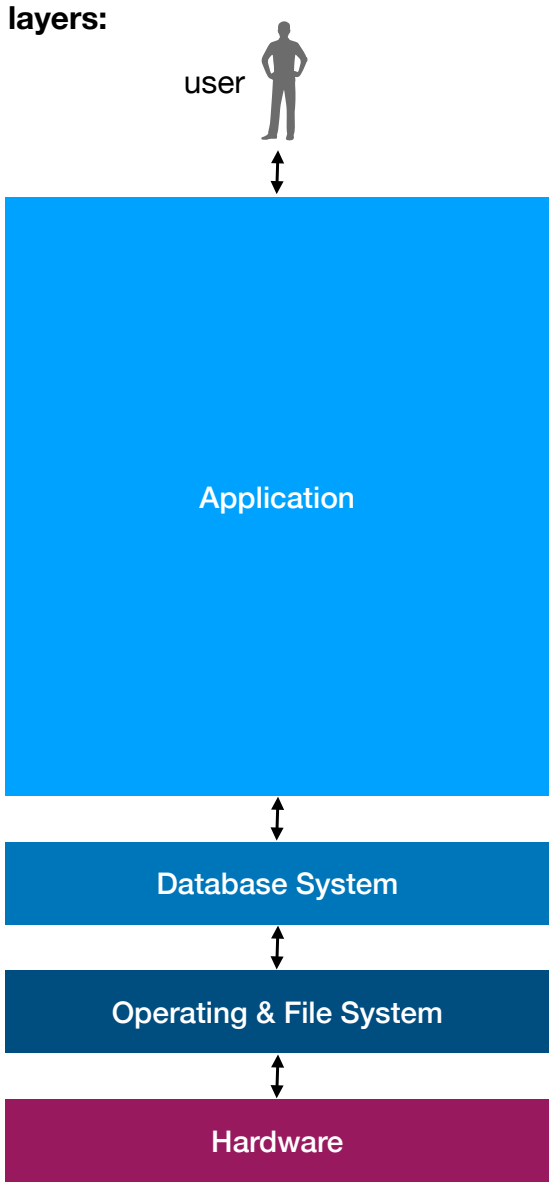
Writing Credits

Quentin Tarantino ... (written by)

Cast (in credits order) verified as complete

	Jamie Foxx	... Django
	Christoph Waltz	... Dr. King Schultz
	Leonardo DiCaprio	... Calvin Candie
	Kerry Washington	... Broomhilda von Shaft

Web Development Stacks: Starting Point



example technologies:

application-code written in Python, Rust, javascript, PHP

PostgreSQL, MySQL, Oracle, SQLite, DuckDB

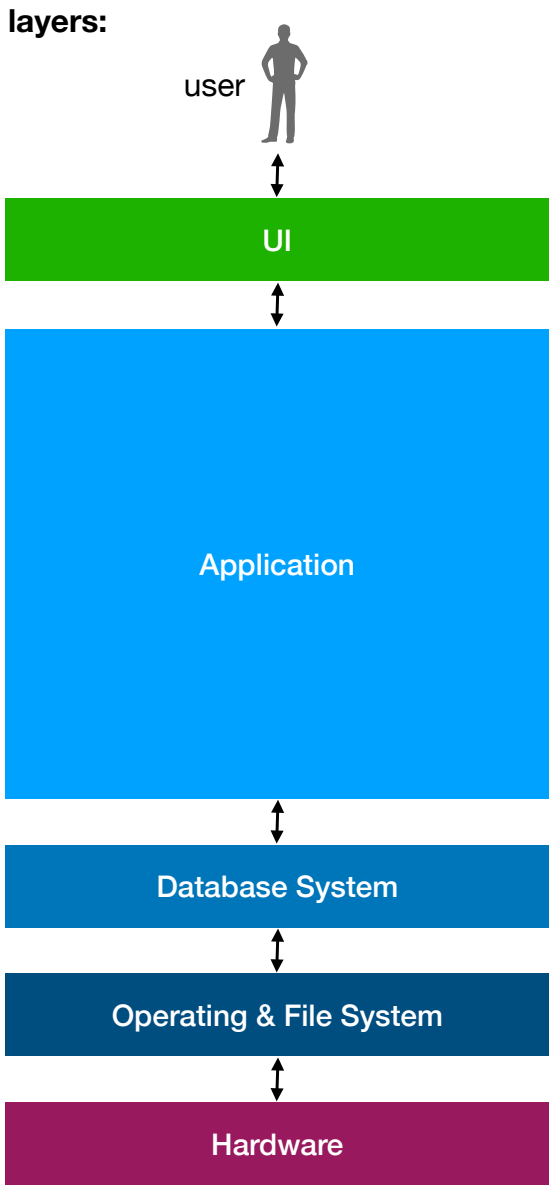
Linux, Windows, OS X, Android, iOS

CPU, DRAM, SSD, hard disk

Monolithic Application

One big application sitting on top of the database system. That application could be layered inside, but does not have to be.

Separate UI-Layer



example technologies:

application-code written in Javascript, PHP

application-code written in Python, Rust, PHP

PostgreSQL, MySQL, Oracle, SQLite, DuckDB

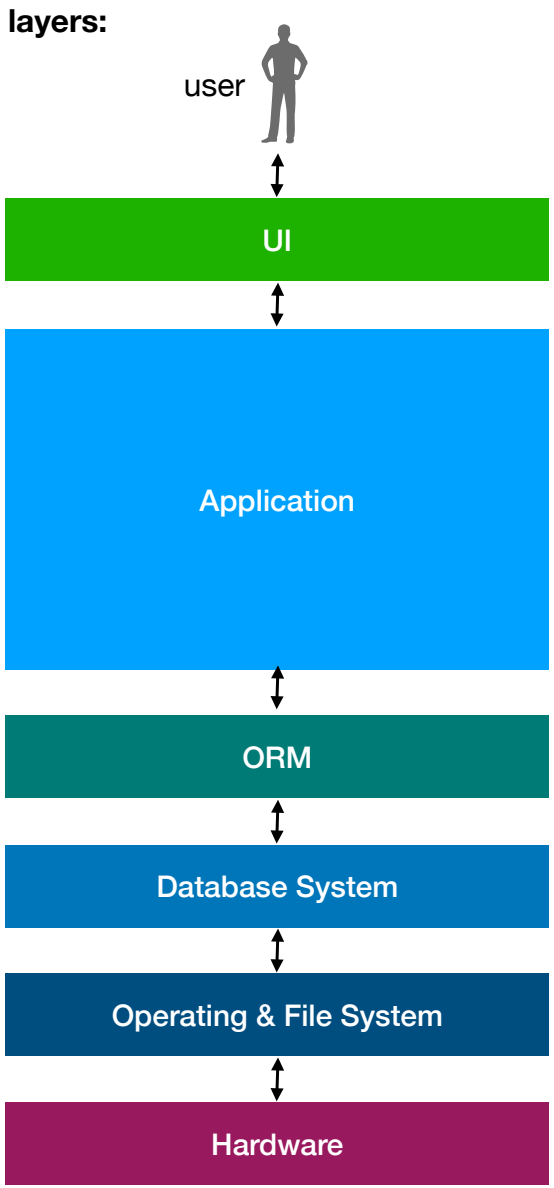
Linux, Windows, OS X, Android, iOS

CPU, DRAM, SSD, hard disk

Split of UI- and Application code

Allows you to easier decouple UI from the actual application

Separate ORM-Layer



example technologies:

application-code written in Javascript, PHP

application-code written in Python, Rust, PHP

object-relational mapper, Django ORM

PostgreSQL, MySQL, Oracle, SQLite, DuckDB

Linux, Windows, OS X, Android, iOS

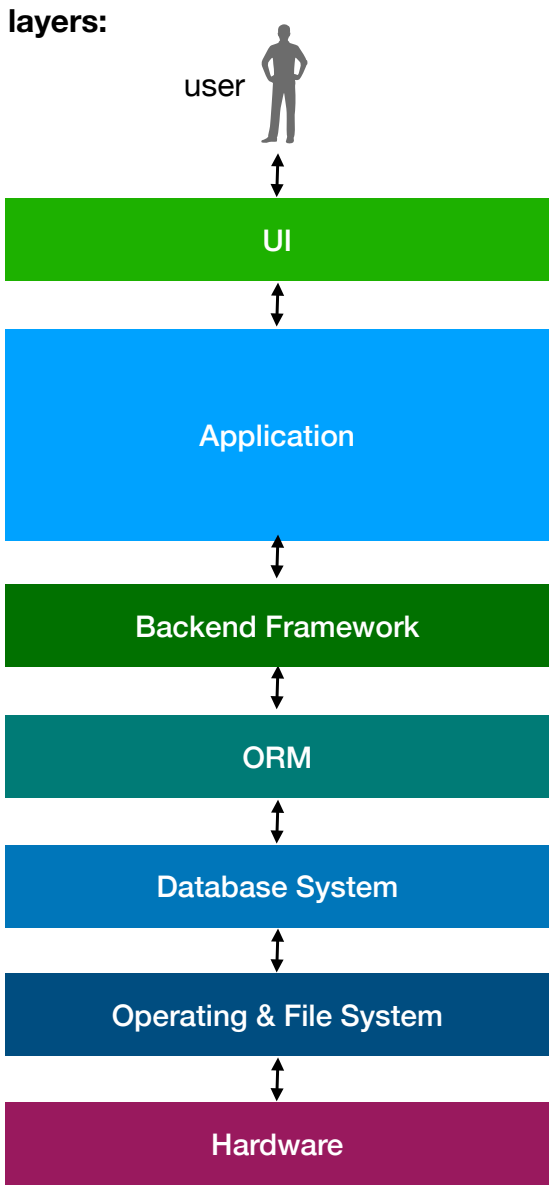
CPU, DRAM, SSD, hard disk

Use an Object-relational-mapper

Fixes the impedance mismatch between SQL and an object-oriented programming language used in the application

Impedance mismatch: Different representations for data: tuples in the relational world vs objects in the OO world.

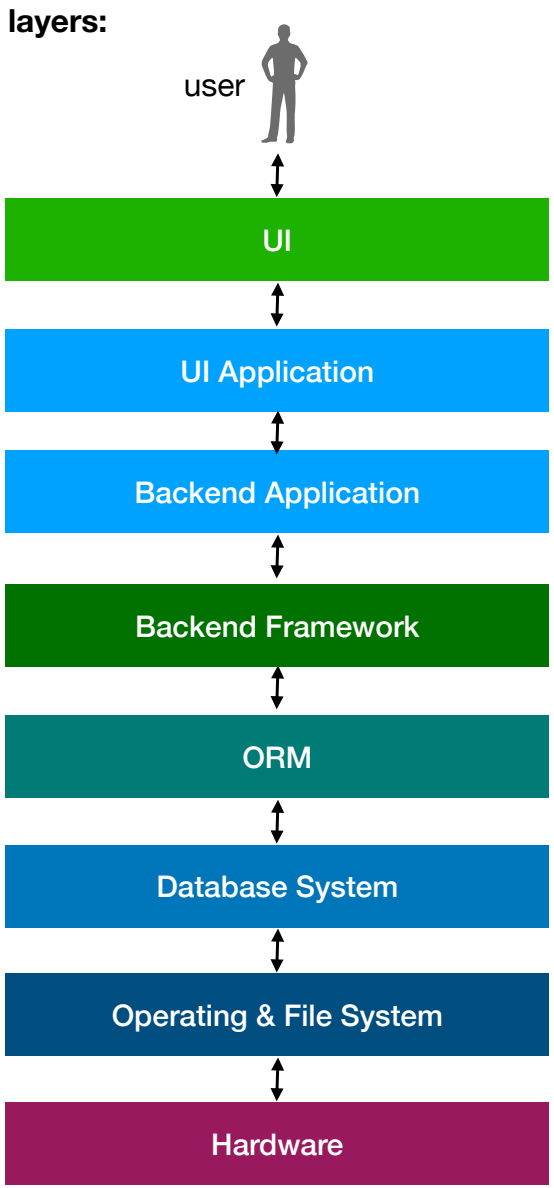
Separate Backend Framework-Layer



Use a Web-Development Framework

Basically a framework/library, allowing you to focus even more on the code that is specific to your application rather than reinventing the wheel

Separate Backend- and UI-Application-Layers



example technologies:

application-code written in Javascript, Vue

UI-code written in javascript, PHP, React

application-code written in Python, Rust

Django

object-relational mapper, Django ORM

PostgreSQL, MySQL, Oracle, SQLite, DuckDB

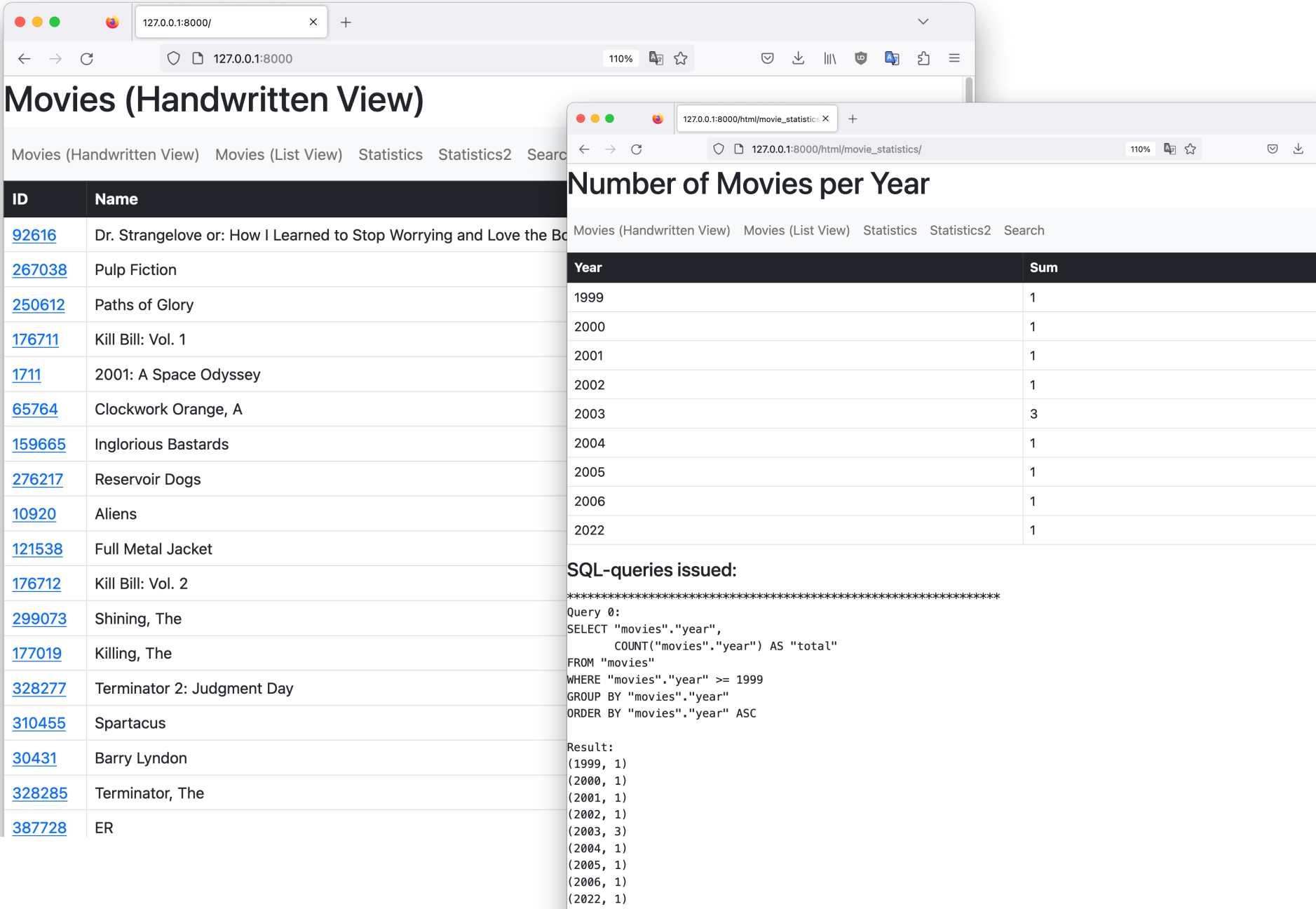
Linux, Windows, OS X, Android, iOS

CPU, DRAM, SSD, hard disk

Decouple UI- and Backend Application

Allows for different frameworks for UI- and backend

Example Project based on IMDb (django_imdb)



ORM (1/2)

Typical Properties of an ORM (Object-Relational Mapper)

An ORM shields the application developer from the database. It allows the application developer to

1. define an object-oriented schema
2. query for objects in that schema
3. insert, update, and delete those objects

Advantages

- much easier application development
- no more messing around with SQL
- automatic protection against SQL injection attacks

Hiding Relational Algebra

Do you recall IMDB (Part 2), Slide 44:
“

Unfortunately, expressions in **relational algebra** are sometimes a bit hard to read. Therefore, another option is to hide **relational algebra** expressions under another language, i.e. design another query language and then translate that language automatically into **relational algebra**.

Fundamental Theorem of Software Engineering (FTSE):

“We can solve any problem by introducing an extra level of indirection . . . except for the problem of too many levels of indirection.”

[David Wheeler]

”

Well, yes, let’s re-use that slide:

Hiding SQL

Unfortunately, expressions in **SQL** are sometimes a bit hard to read. Therefore, another option is to hide **SQL** under another language, i.e. design another query language and then translate that language automatically to **SQL**.

Fundamental Theorem of Software Engineering (FTSE):

"We can solve any problem by introducing an extra level of indirection . . . except for the problem of too many levels of indirection."

[David Wheeler]

In Django ORM that 'other language' is called QuerySets.

Object-Oriented Schema in Django ORM

OO Schema

In Django ORM, an object-oriented schema (OO Schema) is a set of models. Each model is a class definition which corresponds to one relation in the relational model. Models are typically defined in a file 'models.py'.

Example: The relational model:

[movies] : {[id:int, name:varchar(200), year:int, rank:float]}

corresponds to the model:

```
class Movie(models.Model):
    name = models.CharField(max_length=200)
    year = models.IntegerField()
    rank = models.FloatField()
    genre = models.ManyToManyField(Genre)
```

Keys

As we do not specify a (primary) key, Django implicitly creates an attribute 'id' which is marked as the primary key.

N:M-relations

The 'intermediate' tables representing N:M-relations in the relational model, are called 'through'-tables. They do not have to be specified explicitly. They are created implicitly by Django once a ManyToManyField is specified.

Through-Tables with Additional Attributes

If you want to add additional attributes to a through-table you can do this as follows:

```python
class Actor(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    movies = models.ManyToManyField(Movie, through='PlayIn')
```

Here, the attribute 'movies' defines the N:M-relationship to the model 'Movie'. Additionally, the parameter 'through' specifies that there is an explicit definition of the through-table given by the model 'PlayIn'.

```python
2 usages  Joris Nix +1
class PlayIn(models.Model):
    actor = models.ForeignKey(Actor, on_delete=models.CASCADE)
    movie = models.ForeignKey(Movie, on_delete=models.CASCADE)
    role = models.TextField()
```

Here, the model 'PlayIn' defines an extra attribute 'role'.

This corresponds to the relational model:

[PlayIn] : {[id:int, movie_id:(Movie), actors_id:(Actor), role:str]}

rather than:

[PlayIn] : {[movie_id:(Movie), actors_id:(Actor), role:str]}

Also compare our discussion in IMDb (Part 1), slides 45ff.

Keys and pk

Keys

As explained above, if we do not specify a key attribute, Django will implicitly create a key attribute 'id'. However, we can label any attribute as primary key. Then, no artificial 'id'-attribute will be added by Django.

Example:

```
class Grades(models.Model):
    abbreviation = models.CharField(max_length=3, primary_key=True)
```

Here, attribute 'abbreviation' is declared to be the primary key of model 'Grades'.

pk

In Django, whatever was defined to be the primary key for a model, you can simply refer to it using the attribute 'pk'.

Example:

`grade.pk` is a shortcut for `grade.abbreviation`

Unique

Unique

As in SQL, we can constrain groups of columns to be unique. This can also be used as a workaround to 'simulate' composite keys (which are not supported by Django).

Example: (for PlayIn)

```
class PlayIn(models.Model):
    actor = models.ForeignKey(Actor, on_delete=models.CASCADE)
    movie = models.ForeignKey(Movie, on_delete=models.CASCADE)
    role = models.TextField()

    Joris Nix +1
    class Meta:
        # Django does not support composite keys: https://code.djangoproject.
        # avoid that the same person plays the same role multiple times in th
        unique_together = (('actor', 'movie', 'role'),)
        db_table = "play_in"
```

Here, any combination of the attribute values for 'actor', 'movie' and 'role' must be unique.

Again, keep in mind that for this model an artificial key attribute 'id ' will be created.

Managed Database Schema

Managed Database Schema

Django manages the database schema for you. For any changes you want to do, you simply change the OO schema. Django will then change the relational database schema accordingly.

However, you have to trigger this process manually.

`python3 manage.py makemigrations`

This command analyses your changes and creates Python code that reflects those changes, i.e. changes that have to be applied to the relational database schema to make it reflect your changes in the OO schema. That migrations code is stored in the subdirectory `migrations`.

`python3 manage.py migrate`

This command inspects whether there are migrations that still need to be executed against the database. If this is the case, the command will execute these migrations, i.e. change the relational database schema.

Unmanaged Database Schema

Unmanaged Database Schema

You can switch off schema managing for individual models. However, then you have to maintain schema changes in both places yourself.

This can be done by setting 'managed=False' in the inner class Meta of the model.

Example:

```python
class Actor(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    gender = models.CharField(max_length=1, choices=GENDER_CHOICES.choices)
    movies = models.ManyToManyField(Movie, through='PlayIn')

    Joris Nix
    def __str__(self):...

    Joris Nix *
    class Meta:
        managed = False
```

As 'managed=False' in inner class Meta, the model 'Actor' will not be considered when computing migrations.

Meta

Meta Class

The inner class Meta is also useful to pass additional options down to the underlying relational database:

1. constraints: constrain certain domains
2. indexes: define indexes
3. unique_together: define unique constraints
4. ordering: the default ordering used when returning lists of objects
5. db_table: the table name to use in the database
6. abstract: make this model an abstract base class

Example:

```
    class Meta:
        managed = False
        db_table = "actors"
        constraints = [
            models.CheckConstraint(
                name="%(app_label)s_%(class)s_gender_valid",
                check=models.Q(gender__in=GENDER_CHOICES.values),
            )
        ]
```

This defines the underlying database table name to be ‘actors’.
It also constrains the attribute ‘gender’ to only have values which are contained in domain ‘GENDER_CHOICES‘.

OO Schema Bootstrapping

OO Schema Bootstrapping

If you start from an existing relational database schema, you can automatically generate the OO schema as follows:

```
python3 manage.py inspectdb > models.py
```

That's it! No need, to rewrite the entire schema in Django.

This allows you...

1. ...to do good-old relational modelling as learned in this lecture.
2. Refine that model
3. Refine that model even more
4. Refine that model even much more
5. ...
6. `python3 manage.py inspectdb > models.py`
7. set all models to managed
8. for any future schema change: do it in the OO model only

QuerySets: Example Query Translation

QuerySet statement in Python:

```python
movies_group_by_year_count = Movie.objects.filter(
    year__gte=1999
).values('year').annotate(
    total=Count('year')
).order_by('year')
```

Object-oriented Model:

```python
11 usages  Joris Nix +1
class Movie(models.Model):
    name = models.CharField(max_length=200)
    year = models.IntegerField()
    rank = models.FloatField()
    genre = models.ManyToManyField(Genre)
```

Translated SQL statement[1]:

```
SELECT "movies"."year",
       COUNT("movies"."year") AS "total"
FROM "movies"
WHERE "movies"."year" >= 1999
GROUP BY "movies"."year"
ORDER BY "movies"."year" ASC
```

Relational Result of the SQL query:

(1999,1),
(2000,1),
(2001,1),
(2002,1),
(2003,3),
(2004,1),
(2005,1),
(2006,1)

OO-result of the QuerySet:

{'year': 1999, 'total': 1},
{'year': 2000, 'total': 1},
{'year': 2001, 'total': 1},
{'year': 2002, 'total': 1},
{'year': 2003, 'total': 3},
{'year': 2004, 'total': 1},
{'year': 2005, 'total': 1},
{'year': 2006, 'total': 1}

[1] for any QuerySet `q` just call `q.query.__str__()` to obtain the SQL query

Accessing the ORM and running the Django Server

Access the ORM in your Jupyter Notebook

Through Django extensions you can play with the QuerySets in your Jupyter notebook.

Running the Django Server

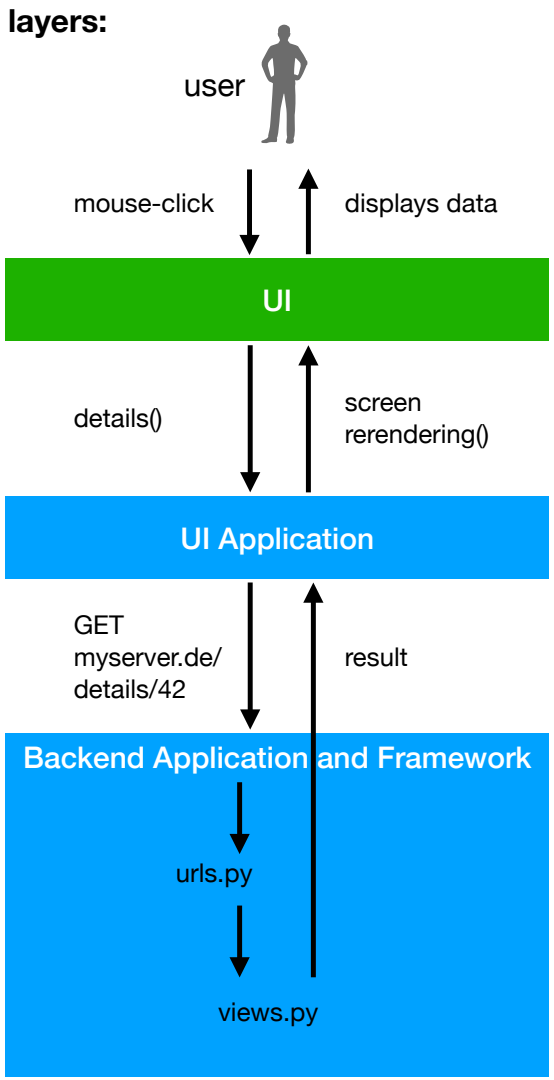
You can start the server by executing:

```
python3 manage.py runserver
```

Obviously, most modern IDEs support Django as well. We recommend PyCharm. To inspect the database we recommend DataGrip. As a student you can use both tools for free.

One cool feature of the Django server is that it listens to changes to your code. If you modify the python code and save, you do **not** have to restart the server, it will restart automatically. This is very useful for development.

How to use the Django Server from Outside: Routing and Views



example technologies:

application-code written in Javascript, Vue

UI-code written in javascript, PHP, React

application-code written in Python, Rust

plus Django

(here depicted as a single combined layer)

From the point of view of the Django server there is an incoming request `GET myserver.de/details/42`.

The URL-pattern is analysed by Django using `urls.py` which then invokes a view-function in `views.py`.

Django returns the result of that function.

Routing: Mapping URLs to Views

Routing

The purpose of routing is to map incoming requests to views. This mapping is done by inspecting the request URL, in particular its suffix. The routing patterns are defined in file `urls.py`.

Example:

```
5   urlpatterns = [
6       path("", views.index, name="movies_index"),
7       path("details/<str:pk>", get_movie_details, name="movie_details"),
8       path("search/", get_movie_search, name="movie_search"),
9       path("movies/", MovieListView.as_view(), name="movie_list_view"),
10      path("movie_statistics/", movie_statistics, name="movie_statistics"),
11      path("movie_statistics2/", movie_statistics2, name="movie_statistics2"),
12  ]
```

Line 8: Any URL with a pattern <hostname>/search will be routed to the user-specified view `get_movie_search`; similar rules for lines 7, 10, and 11

Line 9: Any URL with a pattern <hostname>/movies will be routed to view `MovieListView.as_view()`, i.e. a view created by Django to provide a list of all Movies.

Line 7: Any URL with a pattern <hostname>/details/<str> will be routed to view `get_movie_details`, e.g. <hostname>/details/42 would call that view with $pk = 42$.

Views: Computing Results to Requests

Django computes results to requests through so-called views (not be confused with SQL views[2]). Views are typically defined in a file 'views.py'. Routing will decide which views to invoke. Views can be used in two different ways:

1. HTML-views: Django as an HTML-Web Server

Django serves HTML-files just like any other Web server. In particular, Django uses HTML templates (a standard HTML page with placeholders for data) and fills in data from the database, then it serves the merged HTML-page. For this, Django ships with an inbuilt template-engine.

In our example project: all URLS starting with 'html/'

2. JSON-views: Django as a Restful Server

Django serves JSON-data to the Web application. Rest-'endpoints', i.e. views returning JSON data, can easily be defined using the Django REST framework.

You can freely mix both types of views in the same server.
In our example project: all URLS starting with 'api/' or empty URL suffix.

[2] However, there is a relationship to what database people call logical data independence (LDI): Django views allow you to implement LDI on the application rather than on the database schema-level.

Example HTML-view Definition

```
88  def movie_statistics(request):
89      # Display number of movies per year filtered by year >= 1999
90      movies_group_by_year_count = Movie.objects.filter(
91          year__gte=1999
92      ).values('year').annotate(
93          total=Count('year')
94      ).order_by('year')
95
96      context = {
97          "movies_group_by_year_count": movies_group_by_year_count,
98      }
99      return render_with_queries(request, "movies/movie_statistics.html", context)
```

Line 90: We compute statistics on movies using a QuerySet expression.

Line 96: We assign the result to a context object.

Line 99: We call 'render_with_queries' specifying both the HTML-template ('movies/movies_statistics.html') to be used as well as the data ('context') to be used to fill out that template. That function will eventually call a render()-function which returns an HttpResponse object with the fused HTML-page.

In our example project the call to 'render_with_queries' will additionally add the SQL commands issued as well as the relational results of those queries to the front-end (you will not do this in a real application).

Details: render_with_queries and show_queries

```python
def render_with_queries(request, template, context):
    '''
    Render the given template with the given context and show the queries
    :param request: request object
    :param template: template name
    :param context: context dictionary
    :return: rendered template
    '''
    r = render(request, template, context)
    r.write("<h4>SQL-queries issued:</h4>")
    r.write("<pre>")
    r.write(show_queries())
    r.write("</pre>")

    return r
```

```python
# shows the queries from the connection and pretty-prints them
2 usages  Prof. Dr. Jens Dittrich
def show_queries(number_of_queries_only=False):
    ret = ""
    if number_of_queries_only:
        ret += "number of queries:", len(connection.queries)
        return

    count = 0
    for q in connection.queries:
        ret += "**********************************************************************\n"
        ret += "Query " + str(count) + ":" + "\n"
        ret += sqlparse.format(q['sql'], reindent=True, keyword_case='upper') + "\n"
        count += 1
        ret += "\n"
        # execute the query again, but this time with cursor:
        ret += "Result:\n"
        with connection.cursor() as cursor:
            cursor.execute(q['sql'])
            for row in cursor:
                ret += str(row) + "\n"

    ret += "**********************************************************************\n"
    ret += "number of queries: " + str(len(connection.queries)//2) + "\n"
    ret += "**********************************************************************\n"
    return ret
```

Template Engine: Merging Data with HTML

The template engine fuses (joins) an HTML-template with data, i.e. context.

Example:

```
8   <table class="table table-bordered">
9       <thead class="table-dark">
10      <tr>
11          <th>ID</th>
12          <th>Name</th>
13          <th>Year</th>
14          <th>Rank</th>
15          <th>Director</th>
16      </tr>
17      </thead>
18      {% for m in movies %}
19          <tr>
20              <td><a href="details/{{ m.pk }}">{{ m.pk }}</a></td>
21              <td>{{ m.name }}</td>
22              <td>{{ m.year }}</td>
23              <td>{{ m.rank }}</td>
24              <td>
25                  {% for d in m.director_set.all %}
26                      {{ d }}
27                  {% endfor %}
28              </td>
29          </tr>
30      {% endfor %}
31  </table>
```

'{% ... %}' is the syntax for a code block.

'{{ ... }}' returns the value of a variable.

Example:
In line 18: we loop over all objects in variable 'movies', which is part of the context created in the corresponding view. The HTML-snippet contained in the block of the for loop (lines 19–29) will be repeated for every iteration.

In line 21: we output the name of the movie.

Rest

2. JSON-views: Django as a Restful Server

Django serves JSON-data to the Web application. Rest-'endpoints', i.e. views returning JSON data, can easily be defined using the Django REST framework.

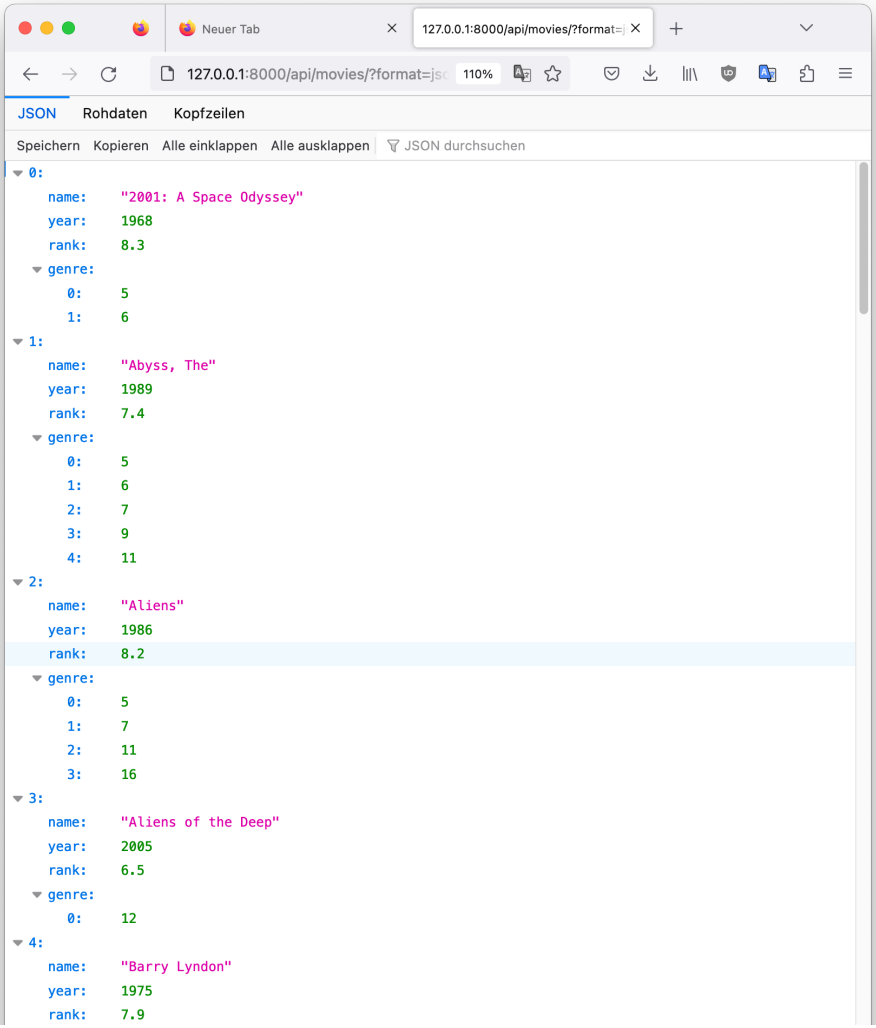
```
26  class MovieSerializer(serializers.ModelSerializer):
        Prof. Dr. Jens Dittrich
27      class Meta:
28          model = Movie
29          fields = ['name', 'year', 'rank', 'genre']
30
31
32  # ViewSets define the view behavior.
    1 usage   Prof. Dr. Jens Dittrich
33  class MovieViewSet(viewsets.ModelViewSet):
34      queryset = Movie.objects.all()
35      serializer_class = MovieSerializer
36
37  # Routers provide an easy way of automatically determining the URL conf.
38  router = routers.DefaultRouter()
39  router.register(r'movies', MovieViewSet)
40
41  urlpatterns = [
42      path('', include(router.urls)),
43      path('api/', include(router.urls)),
44      path("html/", include("movies.urls")),
45      path('admin/', admin.site.urls),
46  ]
47
```

The MovieSerializer defines which fields from Model 'Movie' should be handled by the rest-endpoint.

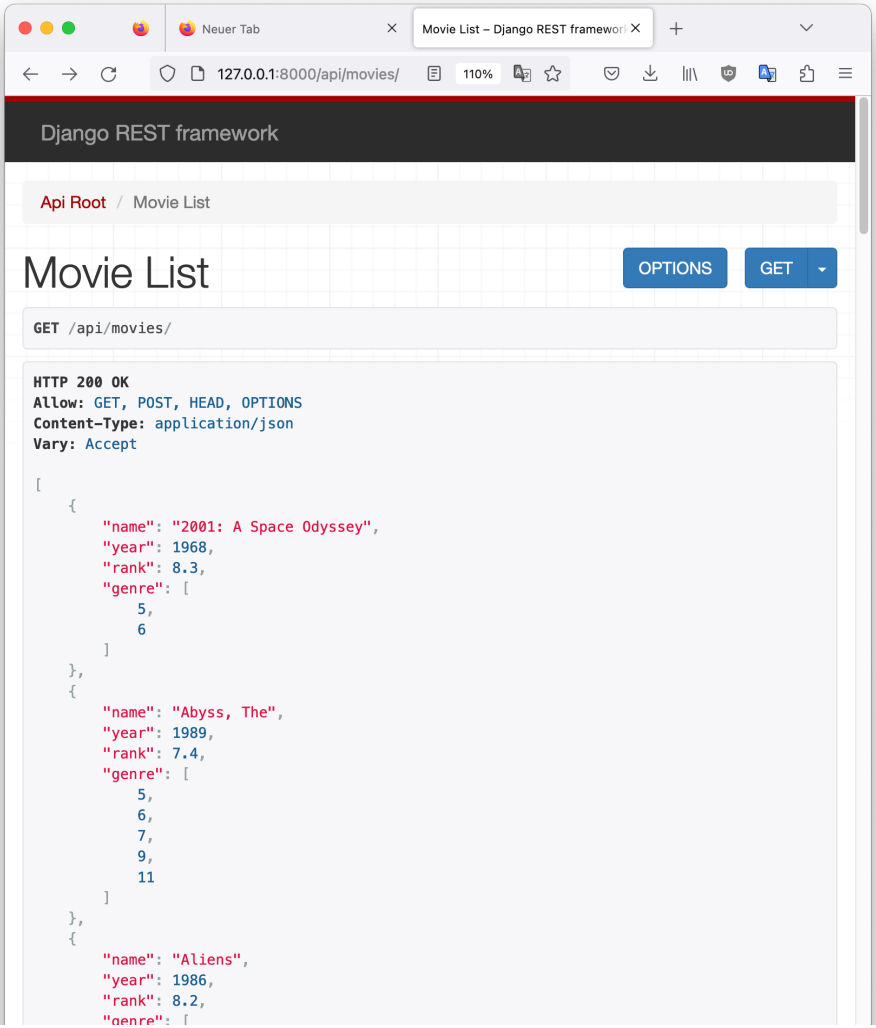
The MovieViewSet defines the behaviour of a REST view, in particular the data to return.

With a router we can easily add the new view to standard routing. Here the serialisers were defined in `urls.py`, however for bigger projects they should be defined in `serializers.py`.

Rest Example



The json data returned from the endpoint and rendered by the browser as a hierarchical view.



The json data returned from the endpoint and rendered by an HTML-page served from the Django Rest-framework to make it more accessible.

HTTP-Requests to Django: GET vs POST

HTTP-Requests to Django: GET vs POST

The Django server has to handle (basically) two types of HTTP-requests:

1. GET: a request that retrieves data from the Django server, but **does not modify** any data in the database. GET-requests get typically parameterised through appending parameters to the URL, e.g.:

   http://www.test.com/index.html?attribute1=value1&attribute2=value2.

2. POST: a request that sends data to the Django server, and potentially **modifies** data in the database. POST-requests send all their data including parameters through a message body.

see `https://www.w3schools.com/tags/ref_httpmethods.asp` for details

Security in a Web Application

Security in a Web Application

Rule 1 When building a web application it is important to design and implement security from the beginning and not just as an afterthought.

Rule 2 Goto Rule 1.

Django ships with some built-in security features, e.g.

- access control based on user roles
- SQL injection prevention
- Cross site request forgery (CSRF)
- etc.

See here for a list.
However, those features are **not enough** to deploy a Web application to the Internet.

How to Build a Secure Web Application

Get a security expert on board. Early. From the beginning. I mean, really.

SQL Injection

Idea

If an application generates a string and then concatenates it to a SQL command string, change the string such that not only a parameter is inserted into the SQL query (as expected), but the SQL query is changed as a whole.

Suppose we write in Python:

```
statement = " SELECT * FROM users WHERE name = ' " + userName + " '; "
```

and userName is set to (e.g. in the frontend):

```
userName = " ' OR '1'='1 "
```

This creates the string:

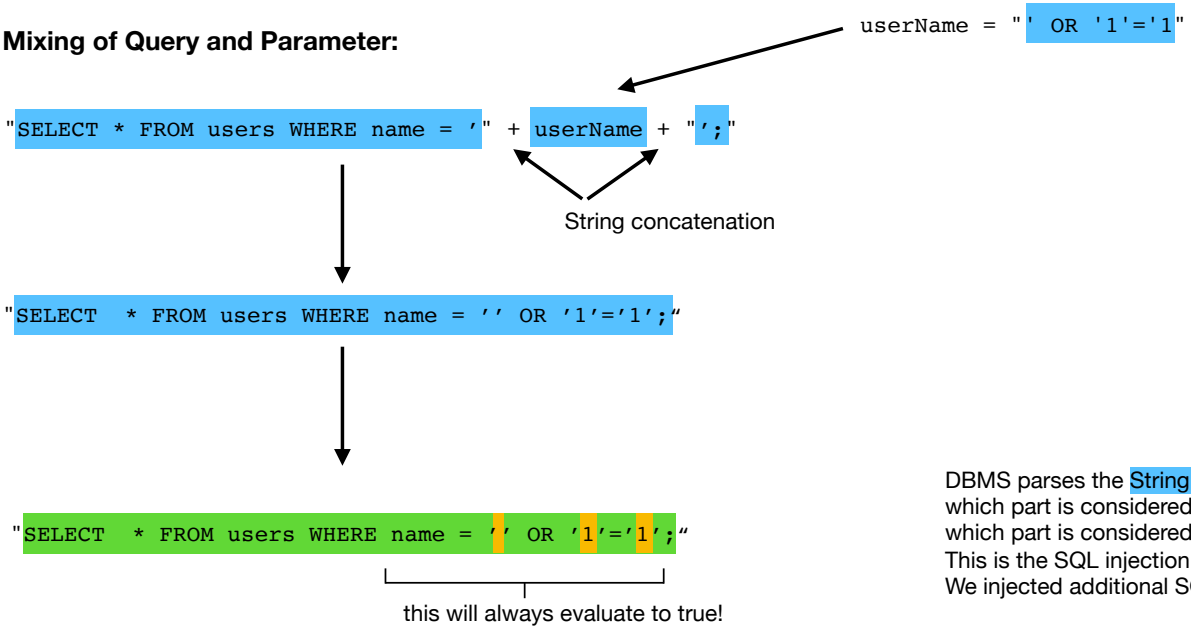
```
" SELECT * FROM users WHERE name = '' OR '1'='1'; "
```

Here, the condition of the WHERE clause will always evaluate to true.

This means, now all tuples are returned and not only a certain user!
Note the creative use of `'` !

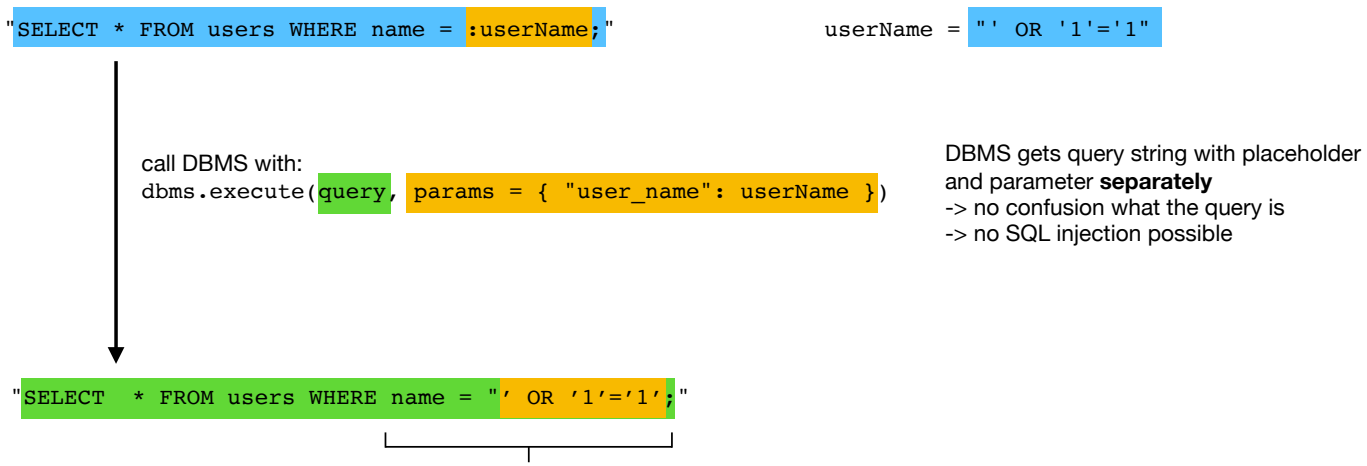
SQL Injection vs Separated Query and Parameters

Mixing of Query and Parameter:



DBMS parses the String and determines itself
which part is considered the query and
which part is considered the parameter
This is the SQL injection **problem**:
We injected additional SQL into the original query!

Clear separation of Query und Parameter:



DBMS gets query string with placeholder
and parameter **separately**
-> no confusion what the query is
-> no SQL injection possible

SQL Injection with Changing the Database

Suppose we write in Python again:

```
statement = " SELECT * FROM users WHERE name = ' " + userName + " '; "
```

and userName is set to (e.g. in the frontend):

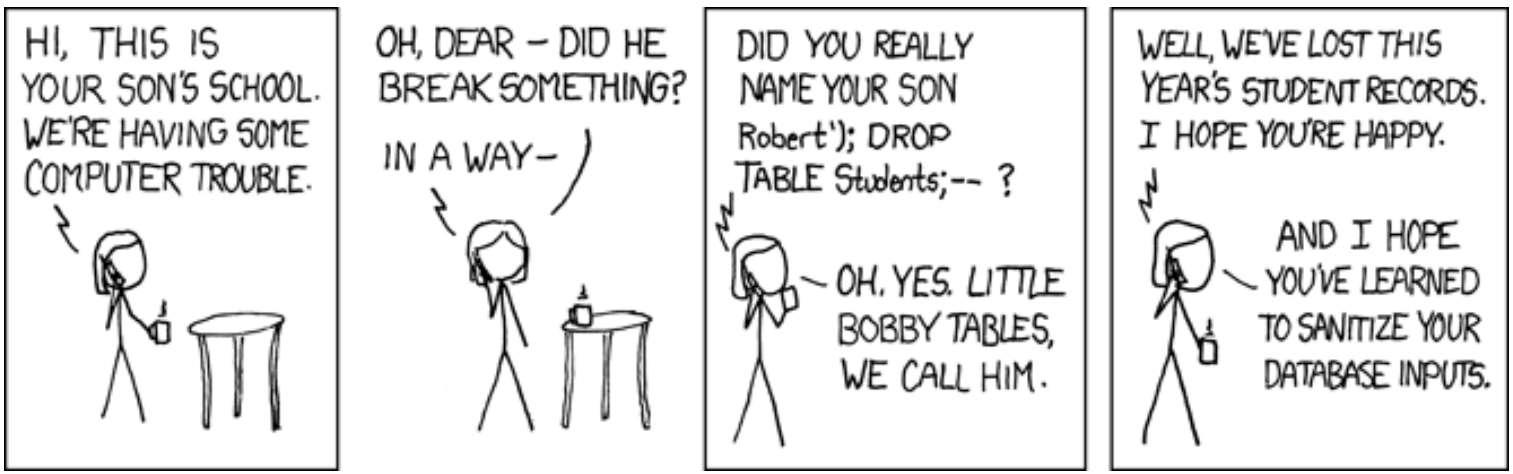
```
userName = " bla'; DROP TABLE users; SELECT * FROM foo WHERE '1' = '1 "
```

This creates the string:

```
SELECT * FROM users WHERE name = 'bla'; DROP TABLE users;
SELECT * FROM foo WHERE '1' = '1';
```

This means, we delete the entire users table!

Little Bobby Tables



[Source: https://xkcd.com/327/]

“Sanitise your Database Inputs”

Basic Problem

Confusion: What belongs to the SQL query?

vs

What is a parameter of the query?

Warning

Unless you are extremely competent with your query language **and** have a security background: **never** write your own sanitiser.

Most DB libraries / ORMs have some features to do sanitisation for you, usually: “prepared statements” or “bind params”.

SQL Injection in Jupyter (SQL Injection_DuckDB.ipynb)

We expect the user to enter something like this.

In [3]:
```
username = "marcel"

get_user_info(username)
```

```
[(6, 'marcel', 's3cby0psc
')]
```

However, the user can also enter something like this, which should be a valid user in our system.

In [4]:
```
username = "' OR '1'='1"
```

This constructs the following query,

```
SELECT * FROM users WHERE username = '' OR '1'='1'
```

which results in a WHERE clause that is always true and therefore, returns the complete `users` table.

SQL Injection_DuckDB.ipynb

Further Material

- Django documentation
- W3schools Django Tutorial
- RealPython Getting Started with Django