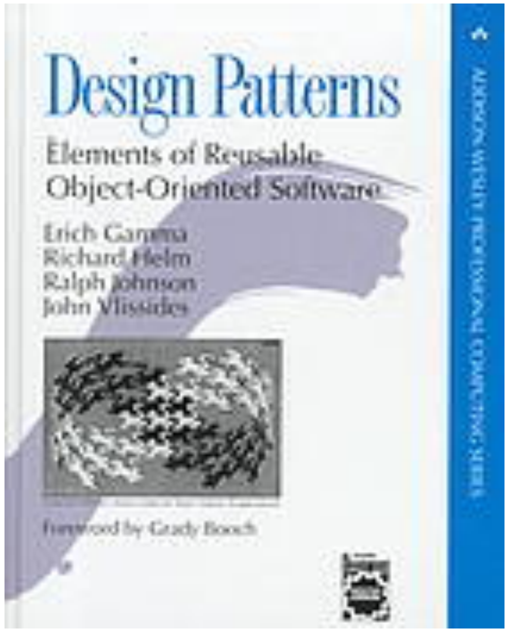
Object Oriented Design Patterns

COSC346

Design Patterns

- Reusable solution to a commonly occurring problem
- Lies between a paradigm and an algorithm
- First book appeared in 1994
  - The "Gang of Four" (GoF)
  - Language features make some patterns unnecessary
  - Can unnecessarily increase complexity

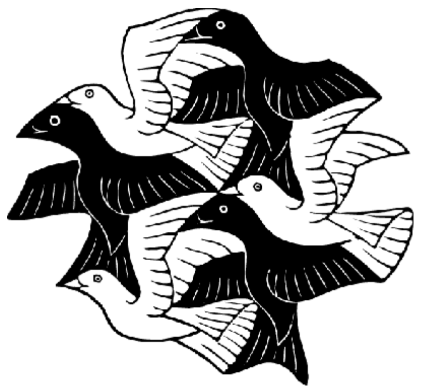
USE WITH CAUTION!

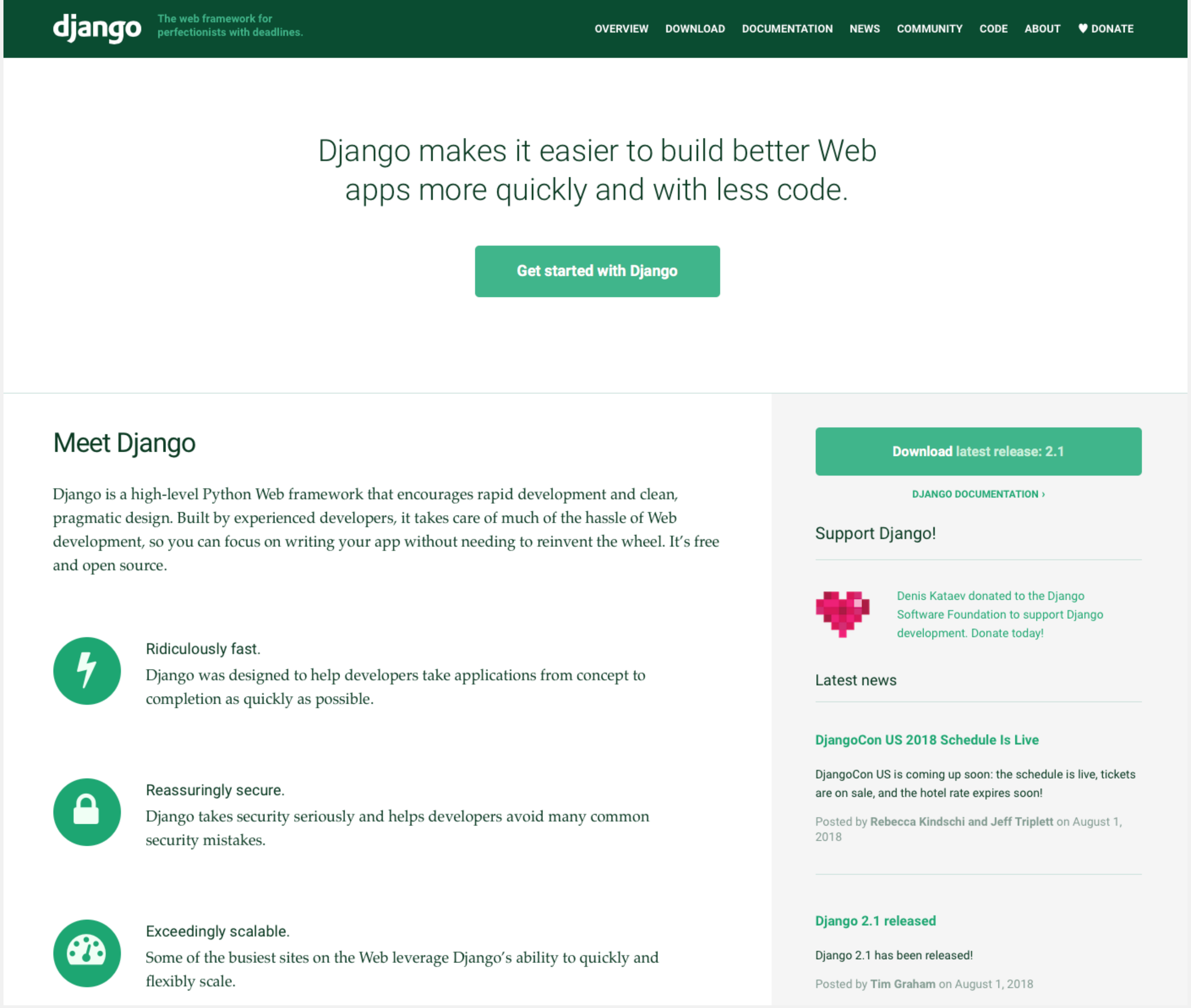

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Addison-Wesley Professional Computing Series

Three types of patterns:
Creational, Behavioural, Structural

Why design patterns?

- Knowing OOP basics does not automatically make you a good OOP designer
- Patterns show you how to build systems with good OO design qualities
    - Patterns don’t give you code, but general solutions to design problems
    - **Patterns aren’t invented, they’re discovered**
    - **Most patterns and principles address issues of change in software**
    - **Most patterns allow some part of a system to vary independently of other parts**


From Head First Design patterns, O'Reilly Media



Django makes it easier to build better Web apps more quickly and with less code.

Meet Django

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Ridiculously fast.
Django was designed to help developers take applications from concept to completion as quickly as possible.

Reassuringly secure.
Django takes security seriously and helps developers avoid many common security mistakes.

Exceedingly scalable.
Some of the busiest sites on the Web leverage Django's ability to quickly and flexibly scale.

Support Django!

Denis Kataev donated to the Django Software Foundation to support Django development. Donate today!

Latest news

DjangoCon US 2018 Schedule Is Live

DjangoCon US is coming up soon: the schedule is live, tickets are on sale, and the hotel rate expires soon!

Posted by Rebecca Kindschi and Jeff Triplett on August 1, 2018

Django 2.1 released

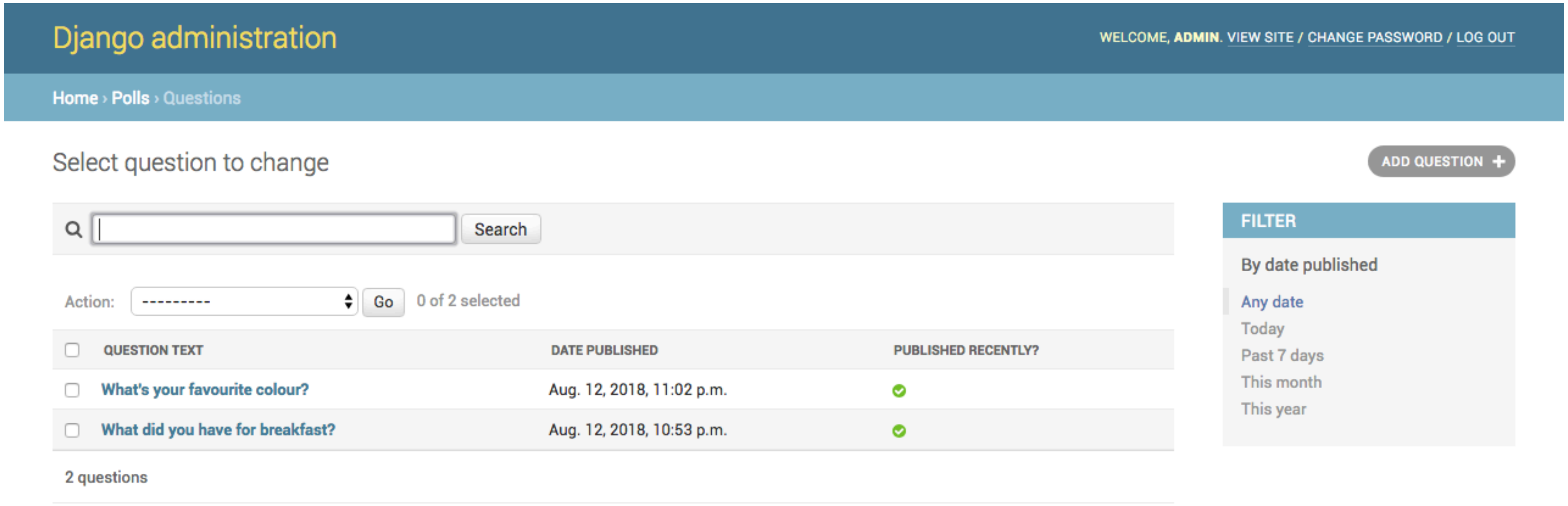
Django 2.1 has been released!

Posted by Tim Graham on August 1, 2018

https://www.djangoproject.com

Polls Application Example

- Anyone can view questions and vote
- Admins can add/remove/change questions and vote options


Django administration

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Polls › Questions

Select question to change

ADD QUESTION +

Search

Action: ---------- Go 0 of 2 selected

	QUESTION TEXT	DATE PUBLISHED	PUBLISHED RECENTLY?
☐	What's your favourite colour?	Aug. 12, 2018, 11:02 p.m.	✓
☐	What did you have for breakfast?	Aug. 12, 2018, 10:53 p.m.	✓

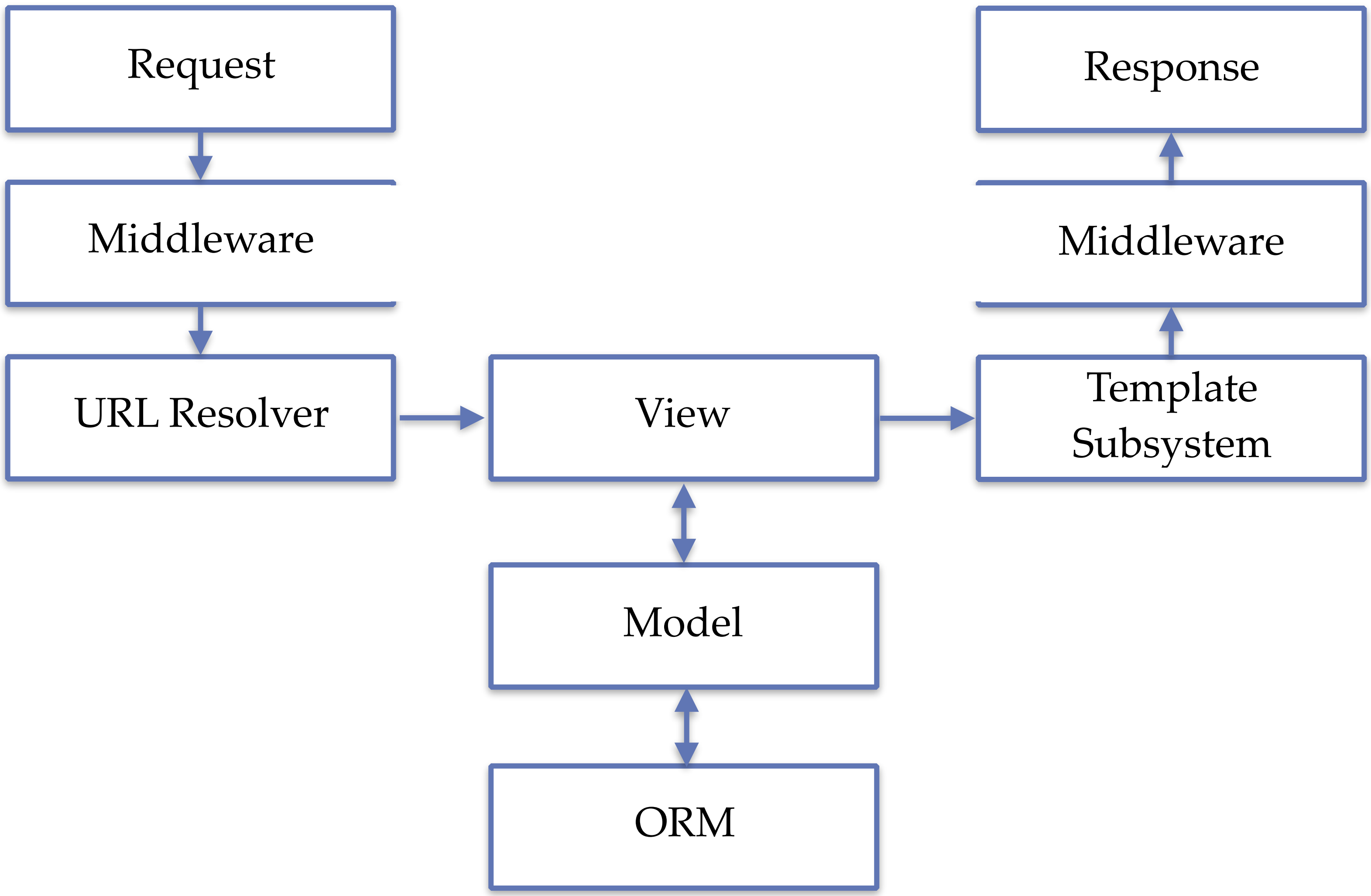
2 questions

FILTER

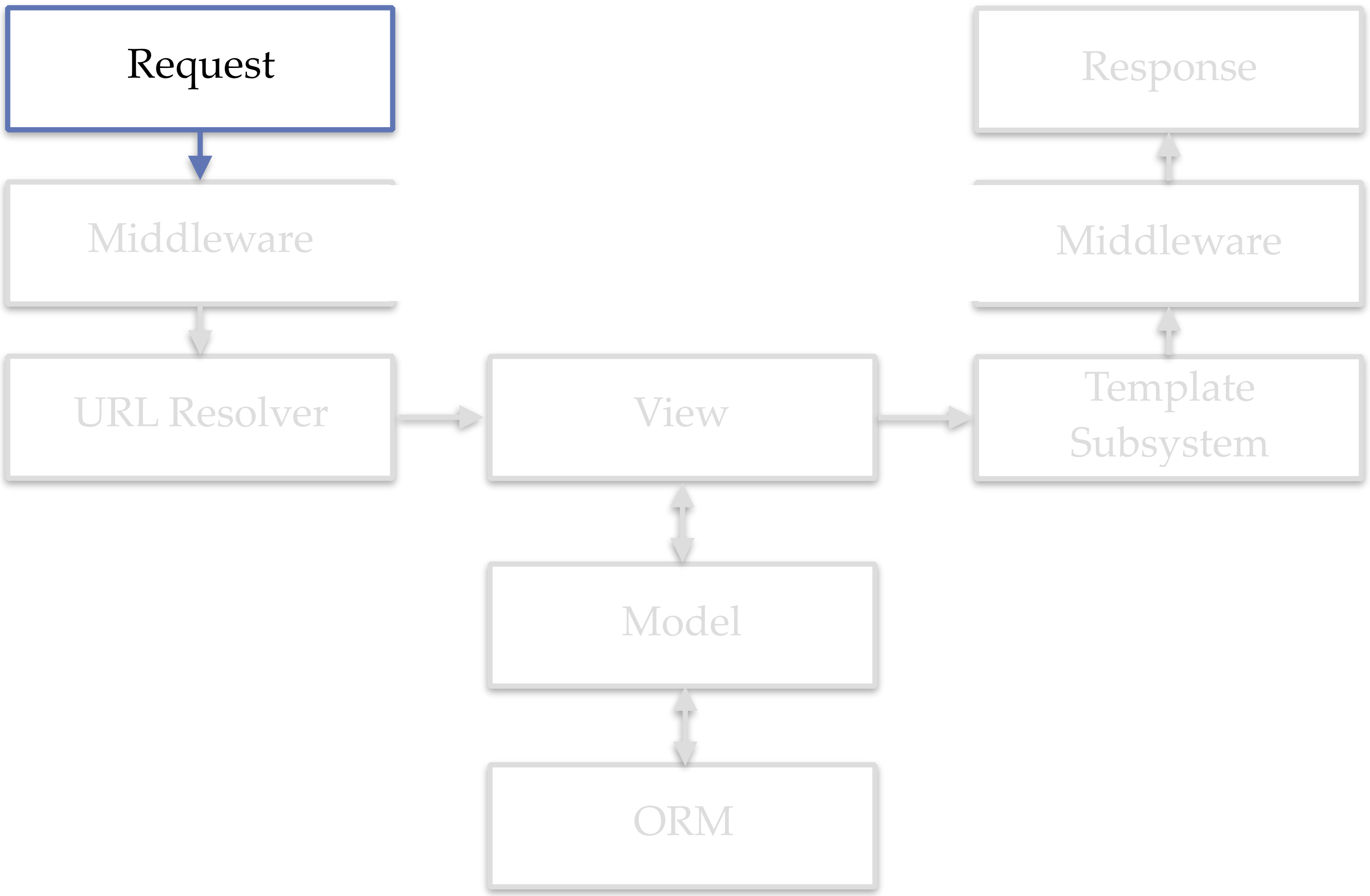
By date published

- Any date
- Today
- Past 7 days
- This month
- This year

Simplified Django Architecture



Simplified Django Architecture



Lifecycle of an HTTP Request

- Client sends request to server
- Server processes the request (middleware)
  - Security
  - Compression
  - Session Handling
  - URL Normalisation
  - Authentication of users
- Server generates response
- Server returns response

HttpRequests in Django

- The request is a command

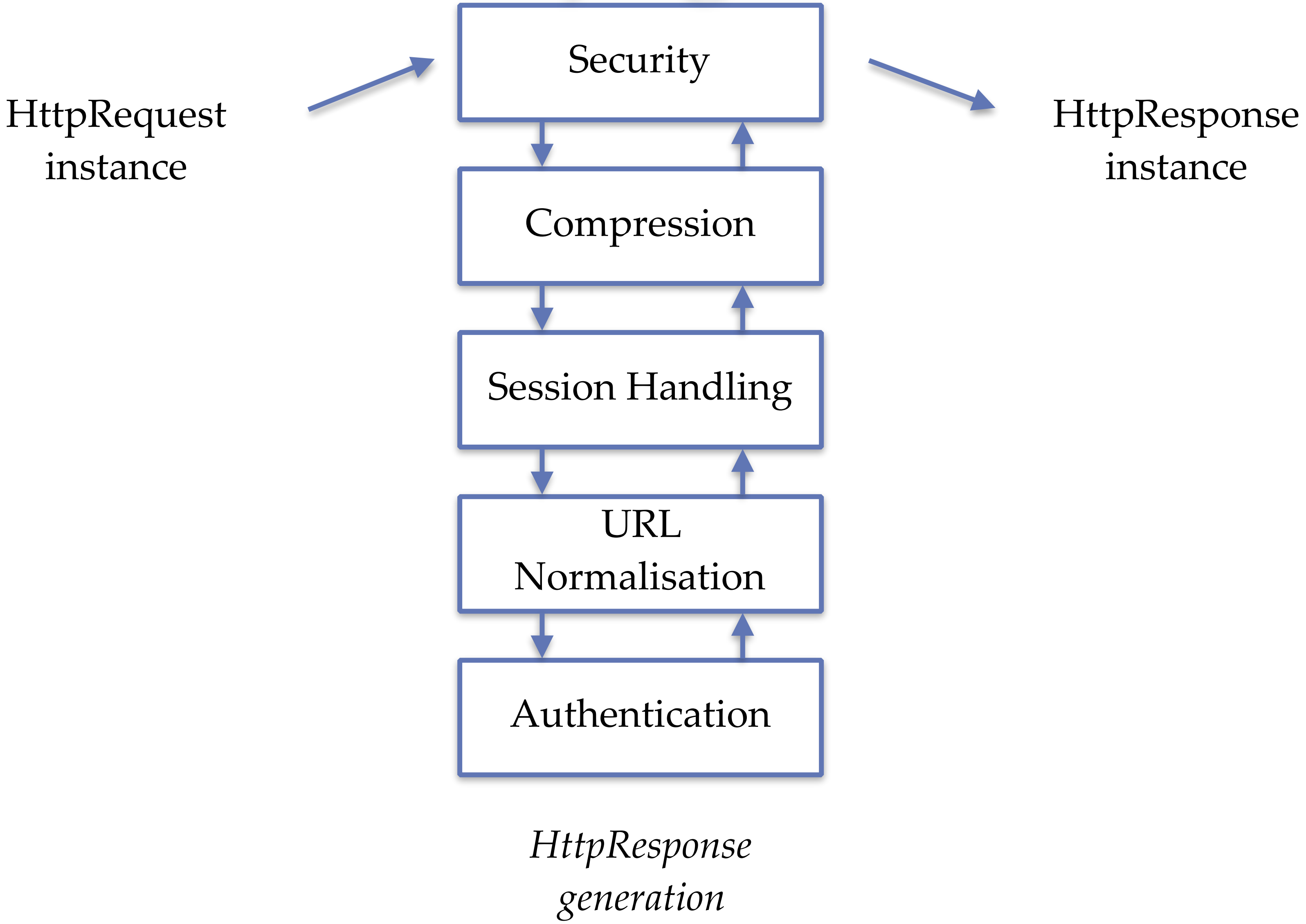
```
GET /polls/ HTTP/1.1
Host: localhost:8000
```

- Django's HttpRequest classes pass state through the system
  - scheme ('http')
  - method ('GET')
  - path ('/polls/')

Middleware - Django

- Security
  - various security options, like HSTS, XSS filtering
- Compression
  - to save data
- Session Handling
  - storing arbitrary data for each visitor (cookies)
- URL Normalisation
  - append slashes, prepend 'www'
- Authentication of users
  - adds currently logged in user to the request

Middleware - Django

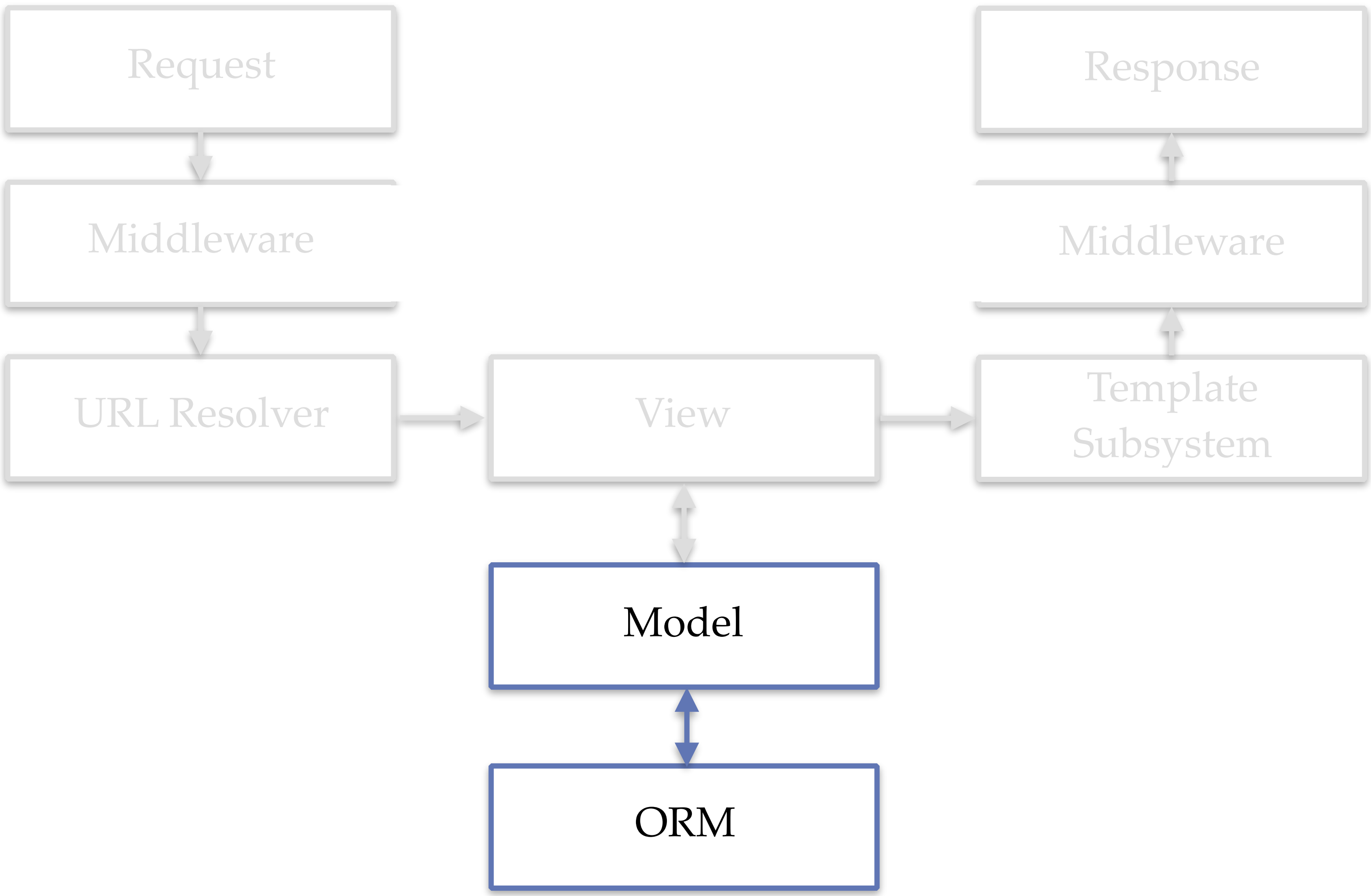


Middleware - Django

- All these middleware have the same interface:

```
class SimpleMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialisation.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response
```

Simplified Django Architecture



Models

- Basic building block of your application
- Defines the data you'll store in the database and what's available in the views.

```
class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def __str__(self):
        return self.question_text


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    def __str__(self):
        return self.choice_text
```

Model Fields

- Map to/from Python and Database Types
  - NULLable?
  - lookup values — primary/foreign key
  - relationships — one-to-one, one-to-many, …
- Validation
  - null or not, valid choices, etc.
- Database table/column names
  - ensure name's valid for the database

All this depends on the DB and the data type

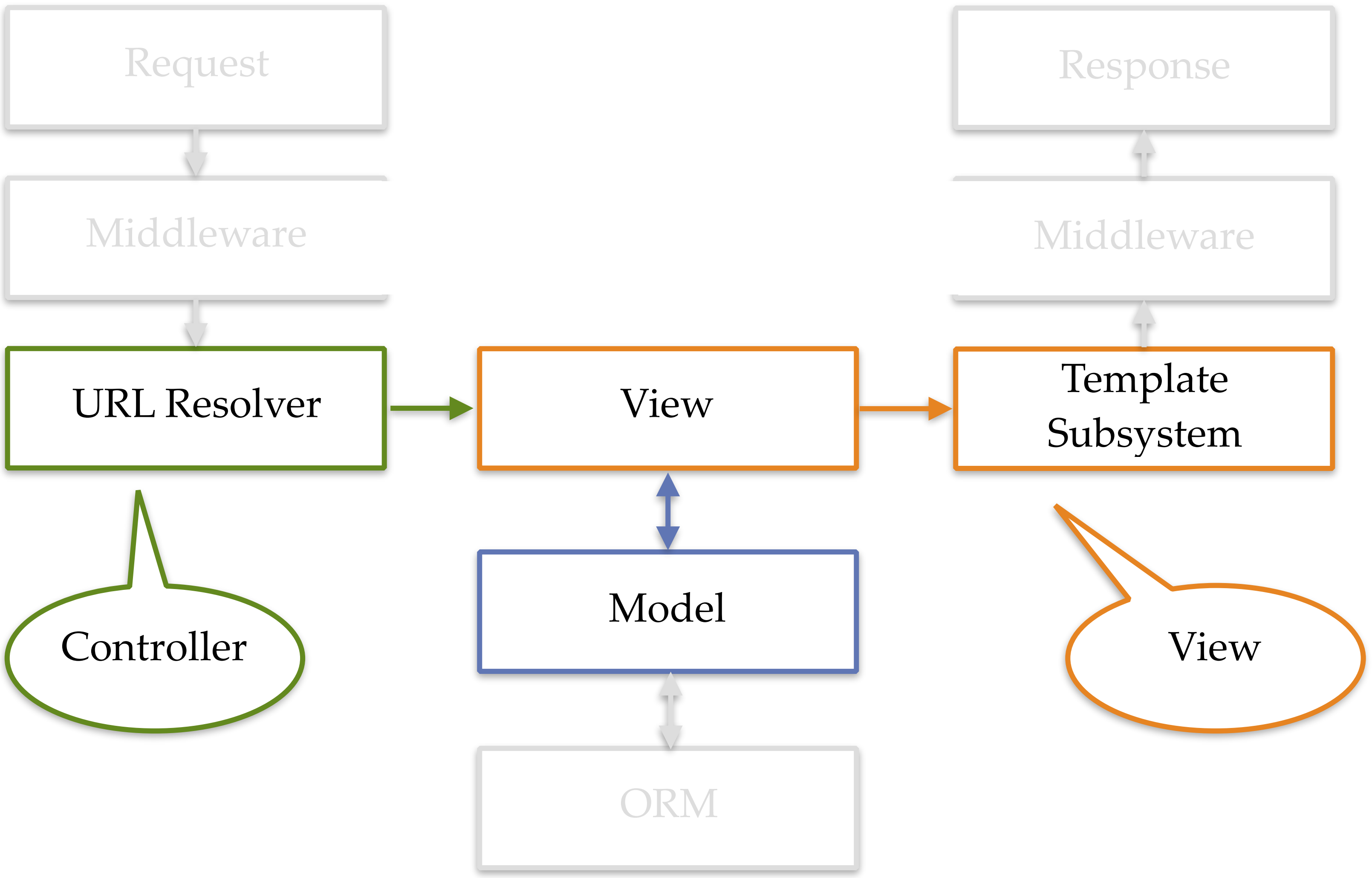
Model Fields

```
class Field():
    """Base class for all field types"""

   # skipping some stuff

    def __init__(self, verbose_name=None, name=None, primary_key=False,
            max_length=None, unique=False, blank=False, null=False,
            db_index=False, rel=None, default=NOT_PROVIDED, editable=True,
            serialize=True, unique_for_date=None, unique_for_month=None,
            unique_for_year=None, choices=None, help_text='', db_column=None,
            db_tablespace=None, auto_created=False, validators=(),
            error_messages=None):
```

Model-View-Controller

- Separation between state, logic, and presentation
- Probably *the* most common pattern
  - Android
  - iOS
  - Django (and a lot of other web frameworks)
  - (probably) most GUI applications
- We'll see a Swift example in the UI part

Simplified Django Architecture



Controller - Django

- Controls the flow of information between the model and the view.
  - url patterns route the request to the view
  - add extra data to help load the correct model

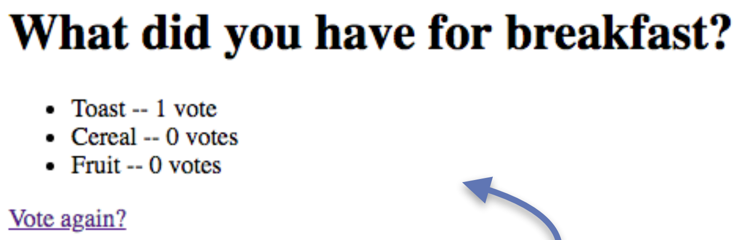
```
urlpatterns = [
    path(r'', views.IndexView.as_view(), name='index'),
    path(r'<int:pk>/', views.DetailView.as_view(), name='detail'),
    path(r'<int:pk>/results/', views.ResultsView.as_view(), name='results'),
    path(r'<int:question_id>/vote/', views.vote, name='vote'),
]
```

Views

- Easily change representation of the objects
  - HTML/CSS/JS for humans
  - JSON/XML for computers
  - CSV/XLS for further processing
  - Charts/graphs
  - Images
- Multiple views of the data are possible
  - table and graph showing the same content

Views - Django

```
class IndexView(generic.ListView):
    template_name = 'polls/index.html'
    context_object_name = 'latest_question_list'

class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'

class ResultsView(generic.DetailView):
    model = Question
    template_name = 'polls/results.html'
```



What did you have for breakfast?

- Toast -- 1 vote
- Cereal -- 0 votes
- Fruit -- 0 votes

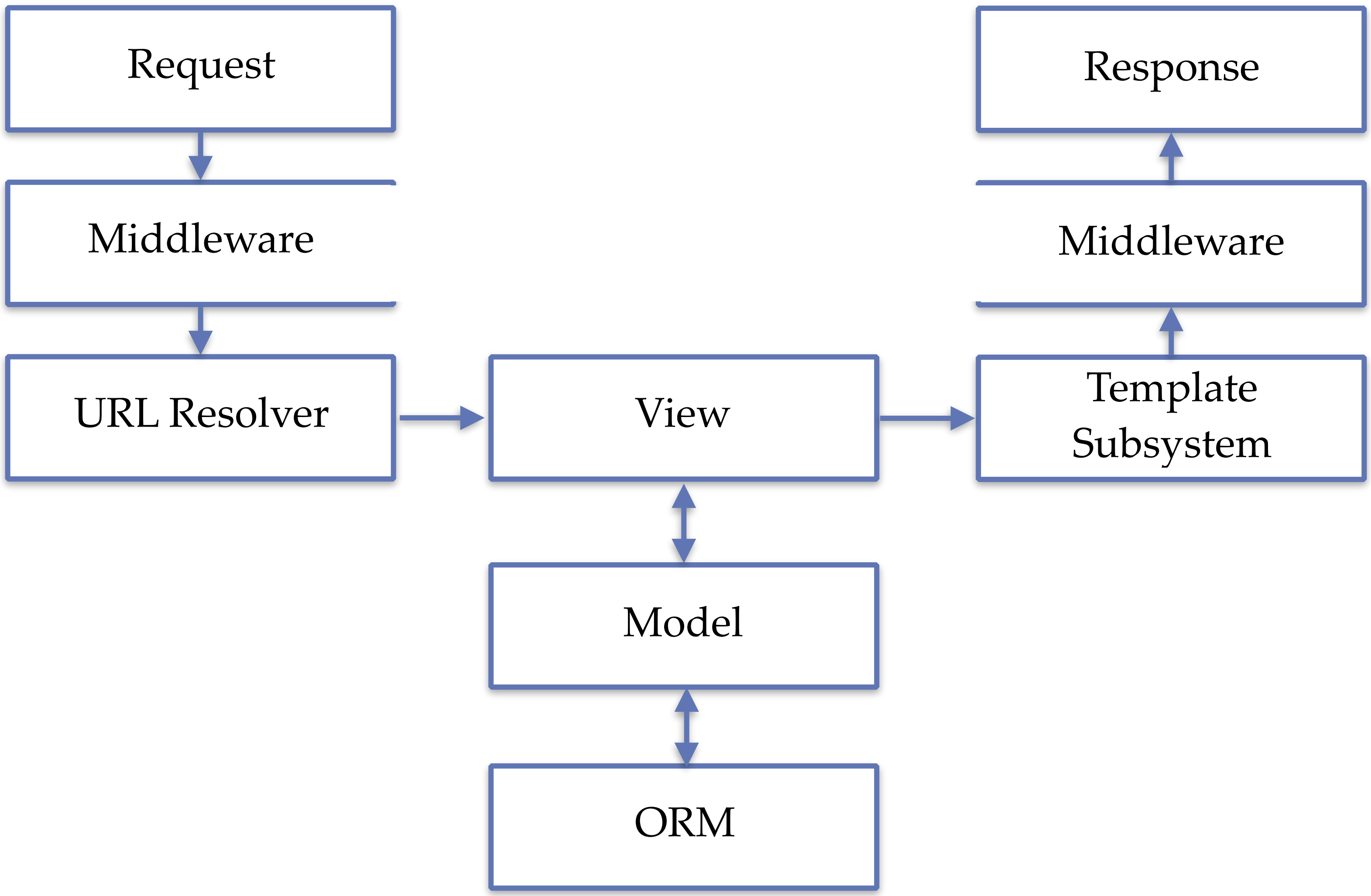
Vote again?

Renders to …

```
<h1>{{question.question_text}}</h1>

<ul>
{% for choice in question.choice_set.all %}
    <li>{{choice.choice_text}} -- {{choice.votes}} vote{{choice.votes|pluralize}}</li>
{% endfor %}
</ul>

<a href="{% url 'polls:detail' question.id %}">Vote again?</a>
```

Django's Template Language

Overview


Request
Middleware
URL Resolver
View
Model
ORM
Template Subsystem
Middleware
Response

Design Pattern - Decorator

- Structural
- Add/remove functionality at runtime
- Wrap the original code

- Adds complexity (cognitive load)
- Can cause problems when specific types are needed

- Django's Middleware wrapping the view

Decorator Example

Swift

Toolmaker

```swift
protocol Coffee {
    func getCost() -> Double
    func getIngredients() -> String
}

class CoffeeDecorator: Coffee {
    private let decoratedCoffee: Coffee
    fileprivate let sep: String = ", "

    required init(decoratedCoffee: Coffee) {
        self.decoratedCoffee = decoratedCoffee
    }

    func getCost() -> Double {
        return decoratedCoffee.getCost()
    }

    func getIngredients() -> String {
        return decoratedCoffee.getIngredients()
    }
}
```

```swift
class SimpleCoffee: Coffee {
    func getCost() -> Double {
        return 3.0
    }

    func getIngredients() -> String {
        return "Coffee"
    }
}

final class Milk: CoffeeDecorator {
    required init(decoratedCoffee c: Coffee) {
        super.init(decoratedCoffee: c)
    }

    override func getCost() -> Double {
        return super.getCost() + 1.0
    }

    override func getIngredients() -> String {
        return super.getIngredients() + sep
           + "Milk"
    }
}
```

Builder

```swift
var simpleCoffee: Coffee = SimpleCoffee()
print("Cost : \(simpleCoffee.getCost()); Ingredients: \(simpleCoffee.getIngredients())")

var coffeeWithMilk: Coffee = Milk(decoratedCoffee: simpleCoffee)
print("Cost : \(coffeeWithMilk.getCost()); Ingredients: \(coffeeWithMilk.getIngredients())")
```

Decorator in the real world

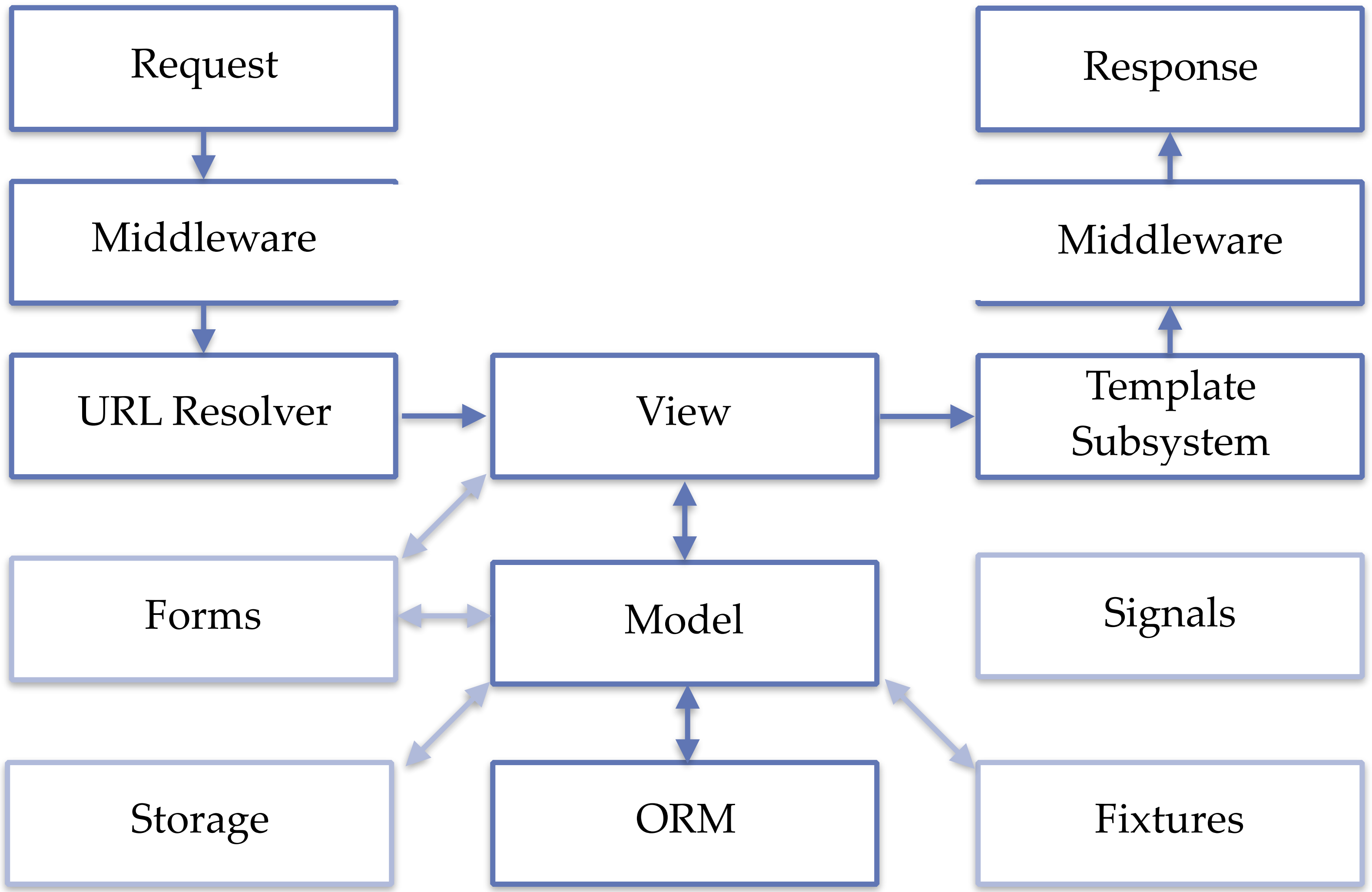
- Java's Input/Output Stream
  - FileInputStream
  - BufferedInputStream
  - GzipInputStream
  - ObjectInputStream

NOT python's @decorator

Summary

- Real-word OO software
  - Django (web framework)
  - Polls (application)
- Design Patterns discussed:
  - Factory
  - Command
  - Decorator
  - Model-View-Controller

Complete* Django Architecture



*more or less