# Parte IV :
# web-app con Django

# Django: caratteristiche principali

- **Full-stack**:
  - Si puo' costruire una intera web application usando solo Django
- **Batteries included**:
  - Moltissimi strumenti built-in di supporto per molteplici aspetti (autenticazione, gestione sessioni, gestione admin, ....)
- Linguaggio lato server: **Python**

# Django: caratteristiche principali

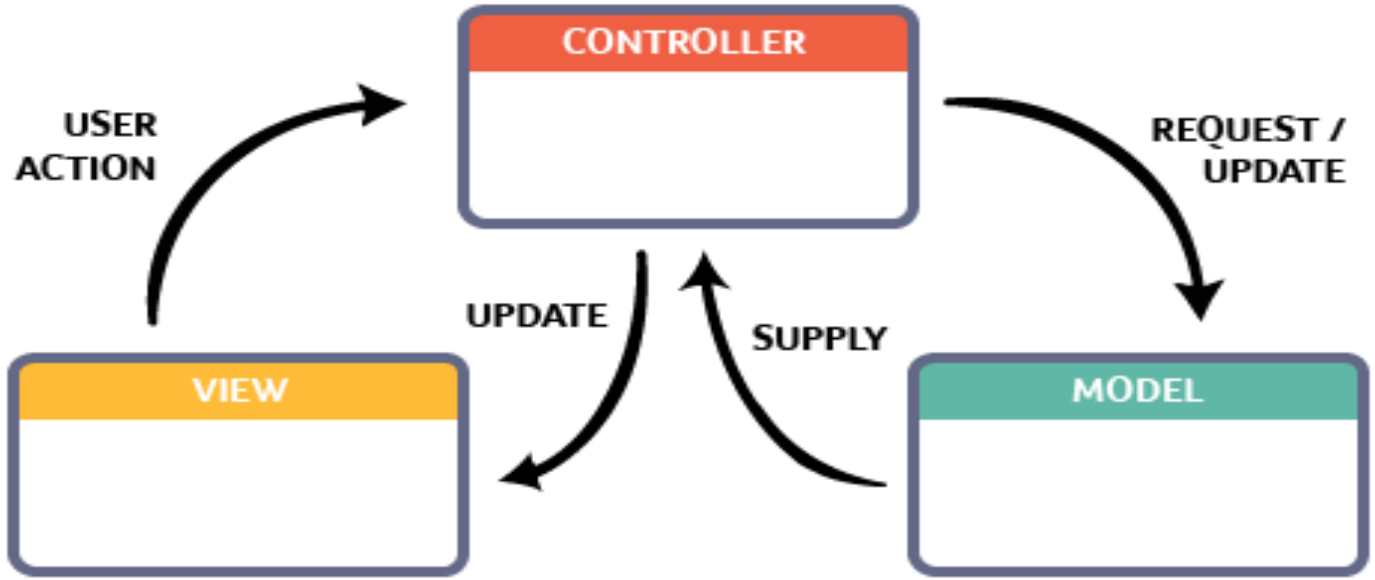
- Architettura basata su **Model-Template-View** (interpretazione del modello MVC Model-View-Controller)
- Realizza un **ORM** (Object Relational Mapping) che descrive il modello dei dati agilmente ed in modo astratto (es. da SQL)
- Supporta la filosofia di progetto **DRY**: Don't Repeat Yourself - ovvero massimo riuso del design dei componenti

# Django: vantaggi e svantaggi

- Vantaggi:
  - Potente
  - Full-stack
  - Ben strutturato
  - Produce siti robusti e scalabili
- Svantaggi:
  - Monolitico e piuttosto pesante  (alternativa light: Flask)
  - Richiede know-how specifico
  - Una certa rigidezza nelle soluzioni e nel loro sviluppo

# MTV vs MVC

- MVC : Model-View-Controller - pattern architetturale applicativo:
  - Model: definizione del modello dati
  - View: definizione della visualizzazione
  - Controller: definizione della logica

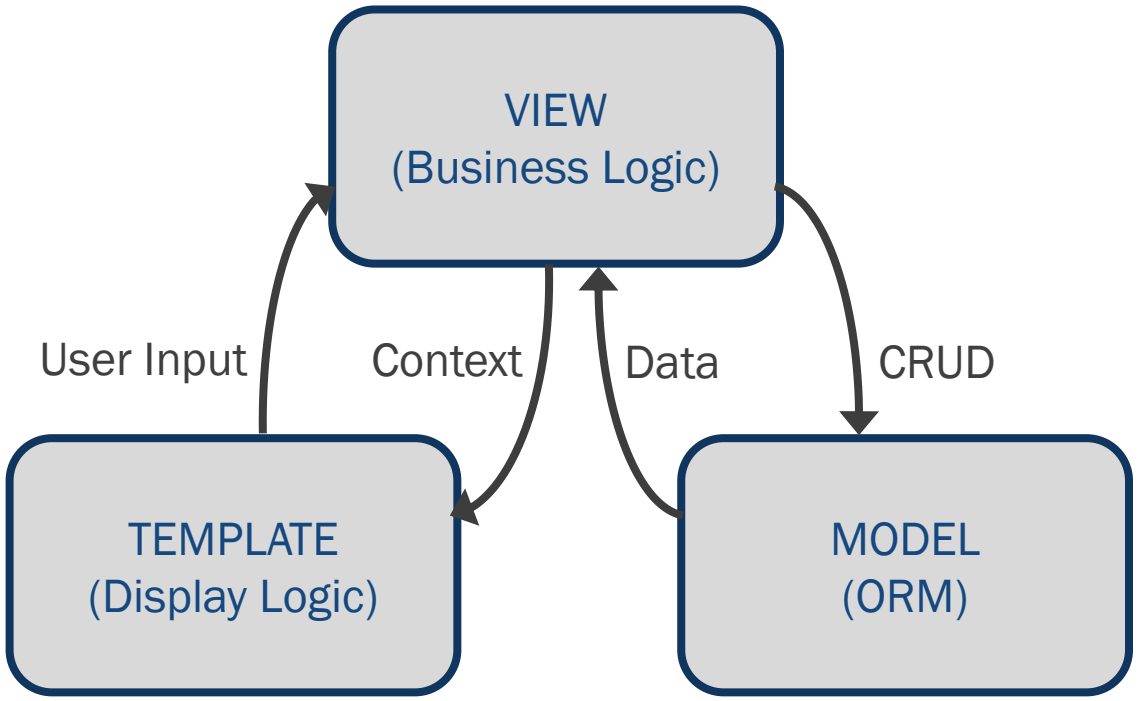


# MTV vs MVC

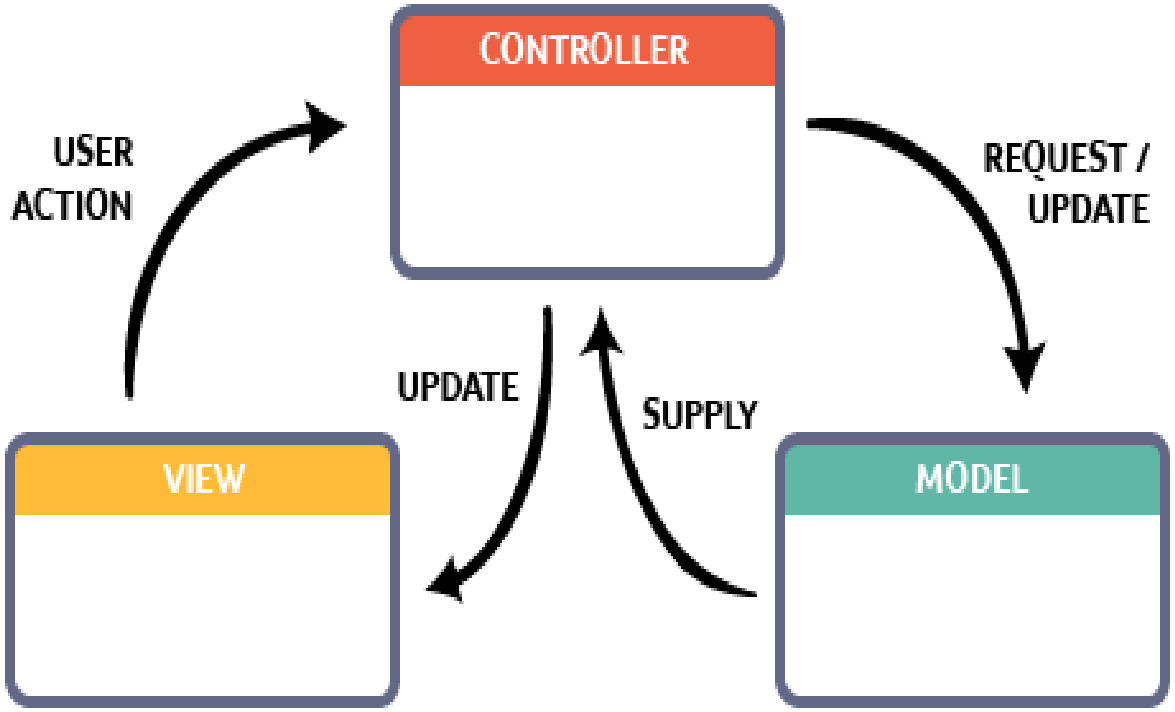
- MTV : acronimo Django per Model-Template-View
- Pattern architetturale che corrisponde 1-a-1 a MVC:
  - Model = Model
  - **Template = View**
  - **View = Control** (certamente crea confusione! Sorry…)

# MTV vs MVC

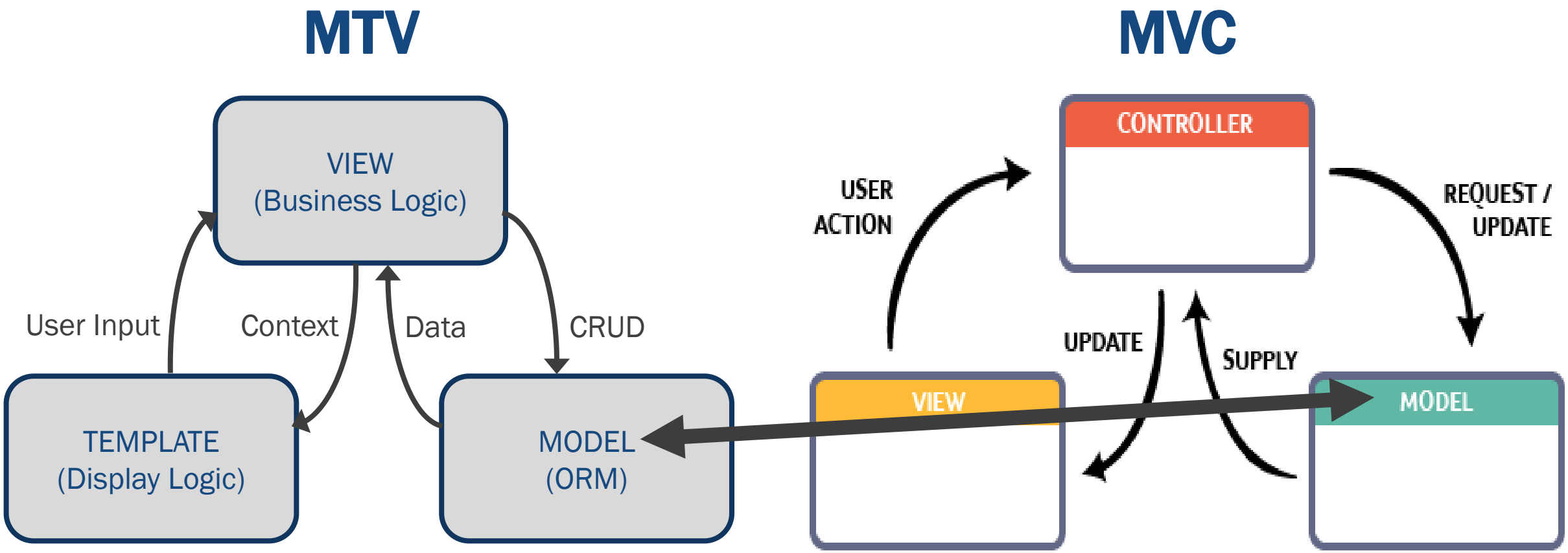
## MTV



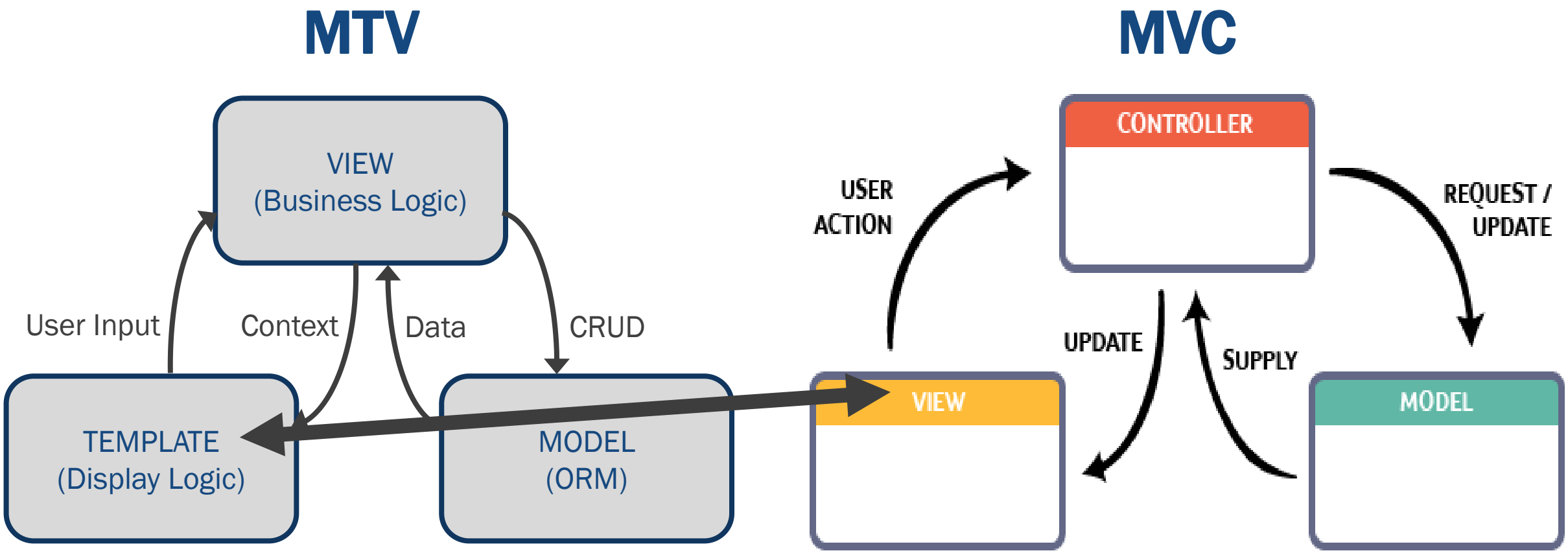

## MVC



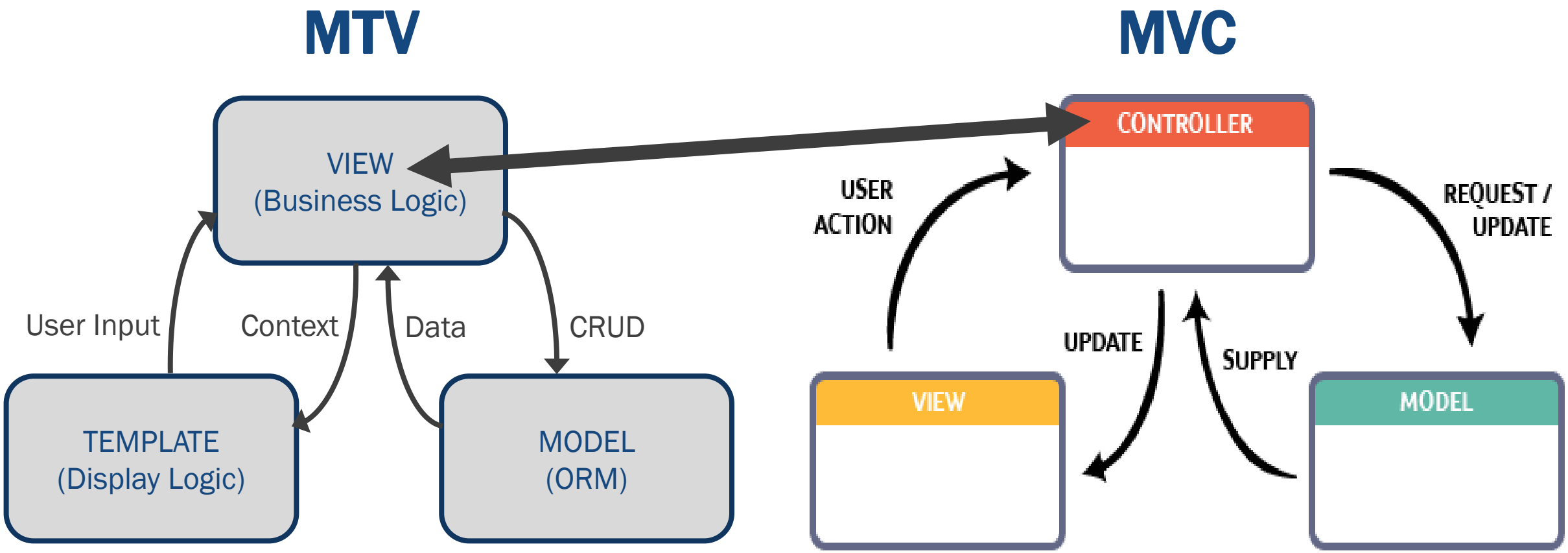
# MTV vs MVC


MTV
VIEW
(Business Logic)
User Input
Context
Data
CRUD
TEMPLATE
(Display Logic)
MODEL
(ORM)
MVC
CONTROLLER
USER
ACTION
REQUEST /
UPDATE
UPDATE
SUPPLY
VIEW
MODEL

# MTV vs MVC


MTV
VIEW
(Business Logic)
User Input
Context
Data
CRUD
TEMPLATE
(Display Logic)
MODEL
(ORM)
MVC
CONTROLLER
USER
ACTION
REQUEST /
UPDATE
UPDATE
SUPPLY
VIEW
MODEL

# MTV vs MVC


MTV
VIEW
(Business Logic)
User Input
Context
Data
CRUD
TEMPLATE
(Display Logic)
MODEL
(ORM)
MVC
CONTROLLER
USER
ACTION
REQUEST /
UPDATE
UPDATE
SUPPLY
VIEW
MODEL

# MTV

- Model : definizione dei dati attraverso un ORM
  - ORM: Object Relational Mapping
  - Definisce, in Python, la struttura del modello
  - Prescinde dalla implementazione sottostante come DB

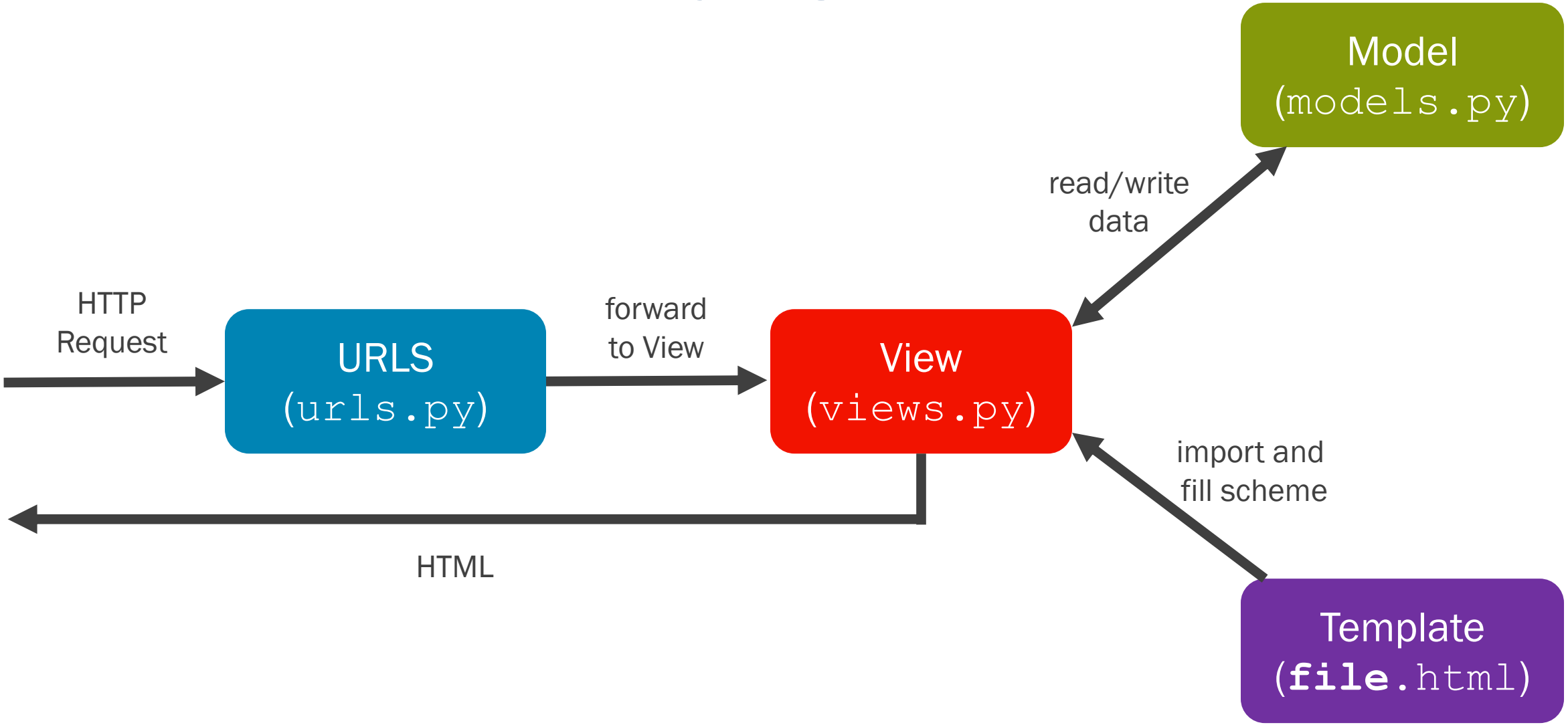
- Template :
  - Schemi delle pagine: HTML piu' contesto dinamico
  - Contesto e elementi schematici definiti con linguaggio specifico di template (non Python!)

# MTV

- View:
  - Logica applicativa e di accesso ai dati
  - Realizzata in Python, via funzioni, oggetti, metodi di Django

- **Inoltre:** URL mappings:
  - Associazioni fra pattern di URL e istanze di MTV

# Django :


Model
(models.py)
read/write
data
HTTP
Request
URLS
(urls.py)
forward
to View
View
(views.py)
import and
fill scheme
HTML
Template
(file.html)

# Prerequisiti a design webapp con Django

- Contesto web:
  - Molto utile: comprensione dei concetti chiave di Internet e web
  - Fondamentale: comprensione della struttura client-server
  - Molto utile: comprensione dei concetti e costrutti di HTTP

# Prerequisiti a design webapp con Django

- Lato client:
  - Fondamentale: comprensione di HTML
  - Molto utile: comprensione di CSS
  - Molto utile: comprensione di Javascript

- Lato server:
  - Conoscenza di Python, inclusa la sua object-orientation
  - Conoscenza specifica di Django

# Django : come procederemo

- Prerequisito: object orientation in Python

- Applicazione web di esempio: biblioteca
  - Descrizione generale
  - Versione incrementali: 1, 2, …, 13
  - Note a estensioni ulteriori

# OOP: recap

- OOP nella prassi (in Python, Java, C++, ...):
  - Gli oggetti sono definiti come :
    - Insiemi di **proprieta'** (dati)
    - **Metodi** che le accedono e modificano
  - Comunicazione fra oggetti: invocazione di metodi
  - Insiemi di oggetti con le stesse caratteristiche (dati e metodi) sono definiti attraverso "schemi" dette **classi**, poi istanziate
  - Meccanismi per riuso di codice derivando classi da altre classi

# OOP: recap

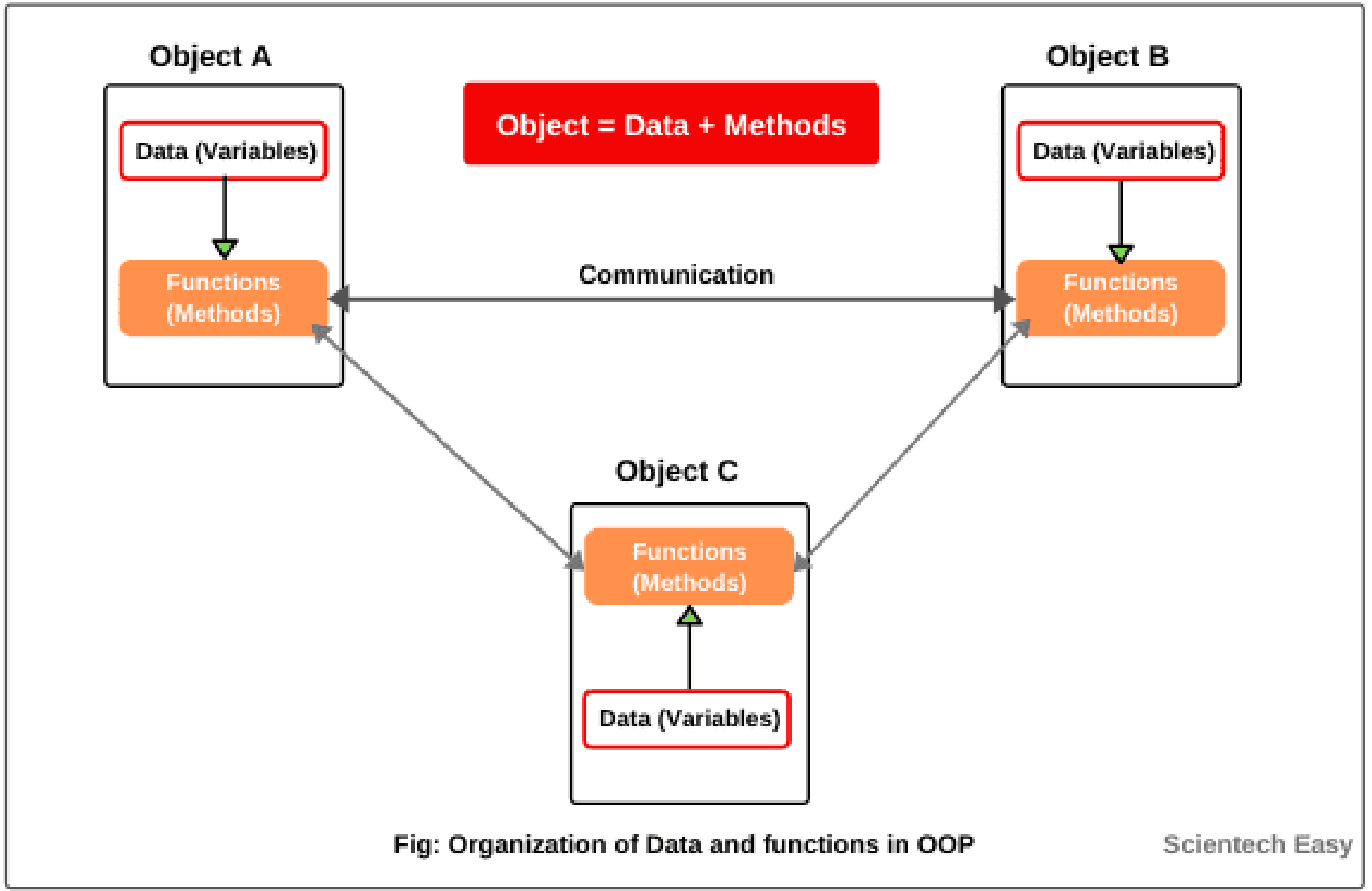



Fig: Organization of Data and functions in OOP

Scientech Easy

# OOP in Python

Classe in Python:

- Definita attraverso la keyword `class`
- A livello di rappresentazione dello stato, una classe contiene:
  - Attributi comuni a ogni istanza: ***class attributes***
    (var. dichiarate e inizializzate a inizio classe)
  - Attributi tipici di una istanza: ***instance attributes***
    (var. dichiarate e inizializzate nel metodo `__init__` )

# OOP in Python

- `__init__` riceve per default almeno l'argomento speciale `self`
- `self` rappresenta la istanza corrente dell'oggetto in creazione
- l'accesso alle proprieta' avviene con la dot notation: "."

```
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age  = age
```

# OOP in Python

Creazione di un oggetto:

- Istanziazione a partire da una classe
- Si fa "chiamando" la classe, valorizzando gli instance attributes
- L'oggetto puo' venir assegnato ad una variabile
- Es.: `my_dog = Dog("Botolo", 6)`

# OOP in Python

Accesso e modifica delle proprieta' (attributi) di un oggetto:

- Attraverso la sintassi "." sulla variabile che contiene l'oggetto
- Gli attributi possono venir acceduti, ma anche modificati

```
my_dog = Dog("Botolo", 6)
a = my_dog.age
a = a + 1
my_dog.age = a
```

# OOP in Python

- Una classe definisce anche gli **instance methods**
- Instance method:  funzione che opera su una istanza di oggetto
- Definita con `def` ed avente come primo parametro `self`
- `__init__`  e' un metodo speciale, attivato in istanziazione

# OOP in Python

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def description(self):
         return f"{self.name} is {self.age} years old"

    def speak(self, sound):
         return f"{self.name} says {sound}"
```

# OOP in Python

Utilizzo degli instance methods: sintassi "." sulla istanza di oggetto

```
my_dog = Dog("Botolo", 6)
d = my_dog.description()
s = my_dog.speak("Bau!")
```

# OOP: decorazioni

- Un modo per incapsulare funzioni/metodi fra altro codice
- Sintassi:

```
@decorazione
 def metodo (...)
          ...
```

- la `@decorazione` definisce codice da venir eseguito prima e/o dopo **`metodo`**
- Decorazioni built-in di Python: `@property`, `@classmethod`, ...
- Anche Django ne definisce di sue: `@login_required`, ...

# OOP : inheritance

- OOP incoraggia il riuso del codice attraverso la **inheritance**

- Idea: permettere di scrivere una nuova classe F specificando che "deriva" da una altra classe pre-esistente P:
  - P e' detta "classe padre";  F e' detta "classe figlio"
  - Si dice che "F eredita da P": ne eredita metodi e attributi

- A livello linguistico, si fa citando P come parametro della classe F

# OOP : inheritance

- Es.
```
class Bulldog(Dog) :
    pass
```

- Questa classe Bulldog semplicemente eredita tutto da Dog (`pass`: "non fare nulla")

- Istanze di `Bulldog` hanno i metodi e attributi di istanze di `Dog`

# OOP : inheritance

- L'aspetto rilevante e' che la classe figlio F puo' estendere e/o modificare quanto ereditato dalla classe padre P:
  - F eredita tutti gli attributi e metodi di P
  - F puo' introdurre attributi ulteriori (di classe e di istanza)
  - F puo' introdurre metodi ulteriori
  - F puo' ridefinire (override) i valori degli attributi ereditati da P
  - F puo' ridefinire (override) i metodi ereditati da P

# OOP : inheritance

```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"This dog {self.name} says {sound}"
```

In questo caso:

- `JackRusselTerrier` e' una classe derivata da `Dog`
- Riscrive il metodo `speak` (dando un default al parametro `sound`)

# OOP : inheritance multipla

- Una classe puo' anche derivare da piu' classi
- Si parla di **ereditarieta' multipla**
- La classe eredita tutti gli attributi ed i metodi dei padri
- In caso di conflitti: esiste un ordine di priorita' fra i padri

# OOP : inheritance multipla

- Es.: supponiamo che:
  - `Pentagono` abbia un attributo "`lati`", array di 5 elementi
  - `Regolare` abbia un metodo "`angolo`"

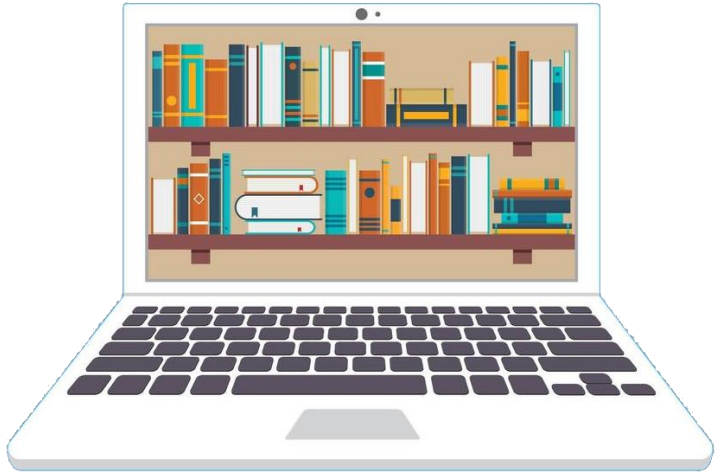
```
class PentagonoRegolare (Regolare, Pentagono):
      // Eredita sia i 5 lati da Pentagono che
      // il metodo per il calcolo angoli da Regolare
```

# OOP in Python: recap

- Un paradigma concettualmente fondamentale
- Si basa su costrutti e concetti elementari semplici
- Tuttavia prevede anche concetti e costrutti piu' avanzati
- La sua comprensione di base e' sufficente ai nostri fini

# Django: la web app di esempio

- Vogliamo sviluppare un sito web per una piccola biblioteca



- Dati che dovremo gestire:
  - Libri, caratterizzati da autore, anno di pubblicazione, genere, …
  - Copie dei libri: istanze di libro caratterizzate da uno stato (es. presente/in prestito) ed eventuali dati sul prestito (utente, data di prestito e ritorno)
  - Dati accessori ai libri: schede autore, schede genere, …

# Django: la web app di esempio

- Mk I (difficile ed esteso: tutti i concetti di base)
  - Sito "di base", solo libri + autori
  - Tre pagine: home, lista libri, lista autori
  - Argomenti: infrastrutture MTV e processo di sviluppo

- Mk II (facile)
  - Estensione con CSS custom
  - Obiettivo: files statici, applicazione delle definizioni CSS

# Django: la web app di esempio

- Mk III (facile)
  - Estensione con paginazione
  - Argomenti: supporto alla paginazione

- Mk IV (facile)
  - Completamento strutture dati (istanze libro, etc)
  - Argomenti: estensione conoscenza tipi dato

# Django: la web app di esempio

- Mk V (media: diversi nuovi concetti)
  - Estensione con pagine di dettaglio a libro e copia
  - Argomenti: nuovi tipi di class-based views e di URL mapping

- Mk VI (facile)
  - Personalizzazione della pagina amministrativa
  - Argomenti: supporto alla personalizzazione della pagina admin

# Django: la web app di esempio

- Mk VII (facile)
  - Estensione con uso dei cookies di sessione
  - Argomenti: comprensione ed uso dei cookies di sessione

- Mk VIII (difficile: molti dettagli)
  - Estensione con meccanismi di autenticazione utente
  - Argomenti: meccanismi di autenticazione e loro gestione

# Django: la web app di esempio

- Mk IX **(difficile: molti dettagli)**
  - Estensione con permessi ad-hoc e pagine ad accesso limitato
  - Argomenti: meccanismi di permesso e loro gestione
- Mk X **(difficile: nuovi concetti)**
  - Estensione con pagina con moduli utente (rinnovo prestito)
  - Argomenti: comprensione ed uso dei moduli (forms)

# Django: la web app di esempio

- Mk XI (facile)
  - Estensione con pagina con ricerca per titolo
  - Argomenti: uso di SQL in Django; filtri Django

- Mk XII (media: diversi nuovi concetti)
  - Estensione con funzione "Metti il libro in manutenzione"
  - Argomenti: uso di Javascript e AJAX nel contesto Django

# Django: la web app di esempio

- Mk XIII (facile)
  - Estensione con tests
  - Argomenti: unit testing in Django

- Argomenti finali:
  - Note al deploy in produzione

# Django: la web app "parallela"

*"Se ascolto dimentico, se vedo ricordo, se faccio capisco"*

[Confucio; Cina, 551-479 a.C.]

- **Fortemente consigliato** sviluppare *indipendentemente ed offline* una app analoga al running example "biblioteca"

# Django: la web app "parallela"

- Ad esempio: web-app per ditta di noleggio vetture:
  - Vetture analoghe ai libri (ma dati diversi)
  - Case produttrici analoghe agli autori (ma dati diversi)
  - Utenti… analoghi agli utenti

- Altre possibilita':
  - Prenotazione lezioni di personal trainer
  - Emeroteca
  - Ludoteca,
  - ….

# Django: installazione

- Prerequisiti a installazione:
  - (essenziale) Python
  - (essenziale) Package manager di Python
  - (possibile) Gestore di ambienti virtuali

# Django: installazione

- Python - per Django 4, e' necessaria almeno la versione 3.8
  - Linux (Ubuntu):
    - Normalmente gia' e' installata (sotto `usr/bin` )
    - Check: `python3 -V`
    - Altrimenti install: `sudo apt install python3`
  - MacOs:
    - Normalmente non c'e' (check: `python3 -V`)
    - Install: da `https://www.python.org/downloads/macos/` scarica ed esegui il package

# Django: installazione

- Windows (10 o 11):
  - Normalmente non c'e' (check da cmd line: `python3 -V`)
  - Install: da https://www.python.org/downloads/windows/ scarica ed esegui il package
  - Ricorda di attivare la opzione "Add Python to PATH"

# Django: installazione

- E' necessario anche il package manager `pip` di Python
  - **Linux** (Ubuntu):
    - Normalmente non c'e'. Check: `pip3 list`
    - Install: `sudo apt install python3-pip`
  - **MacOs**:
    - Viene con la installazione di python3
  - **Windows** (10 o 11):
    - Viene con la installazione di python3

# Django: installazione

- Gestore di ambienti virtuali:
  - Utile per avere ambienti Django separati per siti diversi
  - Installazione non difficile (specialmente in Windows)
  - Buona prassi di sviluppo
  - Proponiamo tale approccio, anche se non e' un req. stringente
  - Ci baseremo su `virtualenvwrapper`, ma esistono alternative (es. `virtualenv`, `venv` , `pipenv` , pur con features differenti)

# Django: installazione

- Linux (Ubuntu):
  - Install: `sudo pip3 install virtualenvwrapper` (fino a Ubuntu 23) oppure
    `sudo apt install python3-virtualenvwrapper` (da Ubuntu 24)
  - Localizzare `virtualenvwrapper.sh` (default: `/usr/local/bin`)
  - Modificare il file `~/.bashrc` aggiungendo alla fine :

```
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
export VIRTUALENVWRAPPER_VIRTUALENV_ARGS=' -p /usr/bin/python3 '
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
```

  - Eseguirlo (`source ~/.bashrc`) o riaprire shell di comando

# Django: installazione

- MacOs (Ubuntu):
  - Install: `sudo pip3 install virtualenvwrapper`
  - Localizzare `virtualenvwrapper.sh` (default: `/usr/local/bin`)
  - Modificare il file `~/.bash_profile` oppure `~/.zshrc` aggiungendo:

```
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
```

  - Eseguirlo con `source`: es. `source ~/.bash_profile` oppure riaprire la shell di comando

# Django: installazione

- Windows (10 o 11):
  - Install: `pip3 install virtualenvwrapper-win` *...e basta!*

# Django: installazione

- Crea (ed accedi automaticamente) un ambiente virtuale python:

  `mkvirtualenv` **`nome_di_ambiente`**

- Da qui in poi:
  - Per listare gli ambienti virtuali: `workon`
  - Per entrare nell'ambiente: `workon` **`nome_di_ambiente`**
  - Per uscirne: `deactivate`
  - Assumeremo da qui in poi di essere nell'ambiente virtuale

# Django: installazione

- Linux (Ubuntu) oppure MacOs:
  - Install: `pip3 install django`
  - Check: `python3 -m django --version`

- Windows (10 o 11):
  - Install: `pip3 install django`
  - Check: `py -3 -m django --version`

- Note:
  - In Windows, e' possibile dover usare `py` invece di `py -3`
  - Da qui in poi assumeremo di essere in Linux

# Django: installazione

- Verifica la installazione creando uno scheletro di test:
  - Crea una cartella, es. `home/user/Django/django_test`
  - Accedila
  - Crea la struttura del sito attraverso `django-admin`:

```
django-admin startproject mytestsite
```

# Django: installazione

- Django ha creato una cartella contenente lo script di gestione `manage.py` che e' usato per attivare il web server locale

```
cd mytestsite
python3 manage.py runserver    // Causera' warning
```

- Il server Django gira in locale su `http://127.0.0.1:8000/`:
  - Apri un browser su tale URL
  - Termina la esecuzione del server con CTRL-C.

**A questo punto abbiamo Django up and running**

# Django: struttura di progetto

- Django distingue fra progetto e applicazione:
  - Un progetto si riferisce ad una intera applicazione web
  - Una applicazione indica in realta' un modulo indipendente
  - Terminologia sviante ma metodologia utile al riuso: DRY

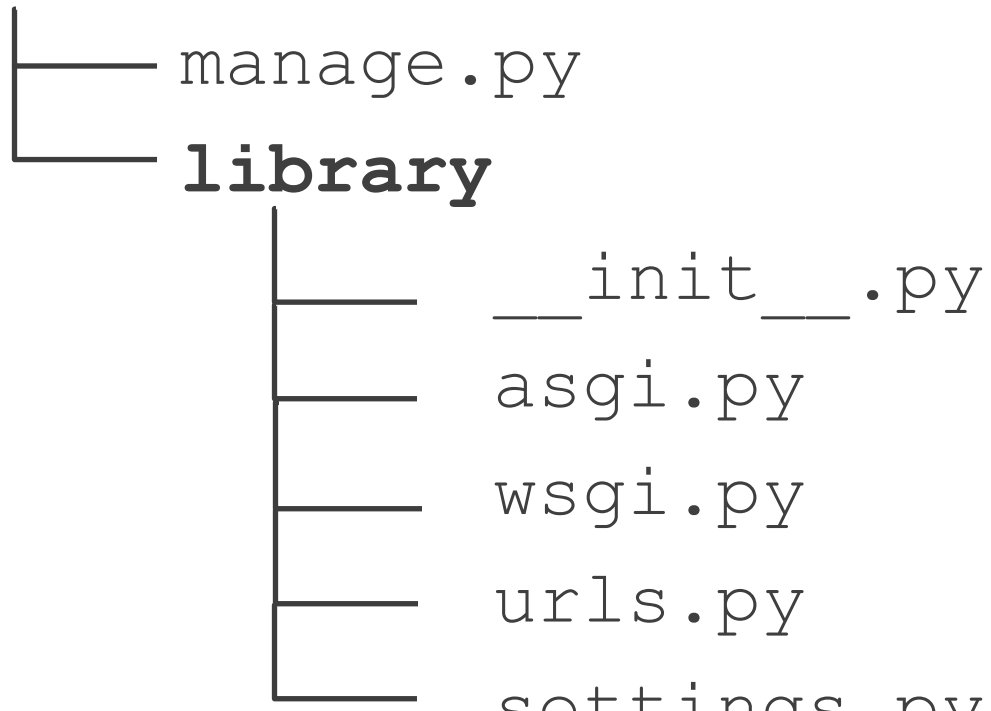
- Nella fattispecie della nostra (semplice) applicazione:
  - Noi avremo un solo progetto `Library`
  - In `Library` avremo una sola applicazione, `catalog`

# Django: struttura di progetto

- Django supporta la creazione di struttura della cartella di progetto:

```
django-admin startproject library
```

- Esito:



# Django: struttura di progetto

Files principali:

- `manage.py` : script di gestione del progetto (comandi per far partire il server, per creare i DB, etc etc. ).
  FISSATO e usato
- `settings.py` : file di configurazione, pre-settato ma:
  lo modificheremo
- `urls.py` : file di progetto di mapping fra URL e views, vuoto,
  lo modificheremo

# Django: struttura di progetto

Altri files:

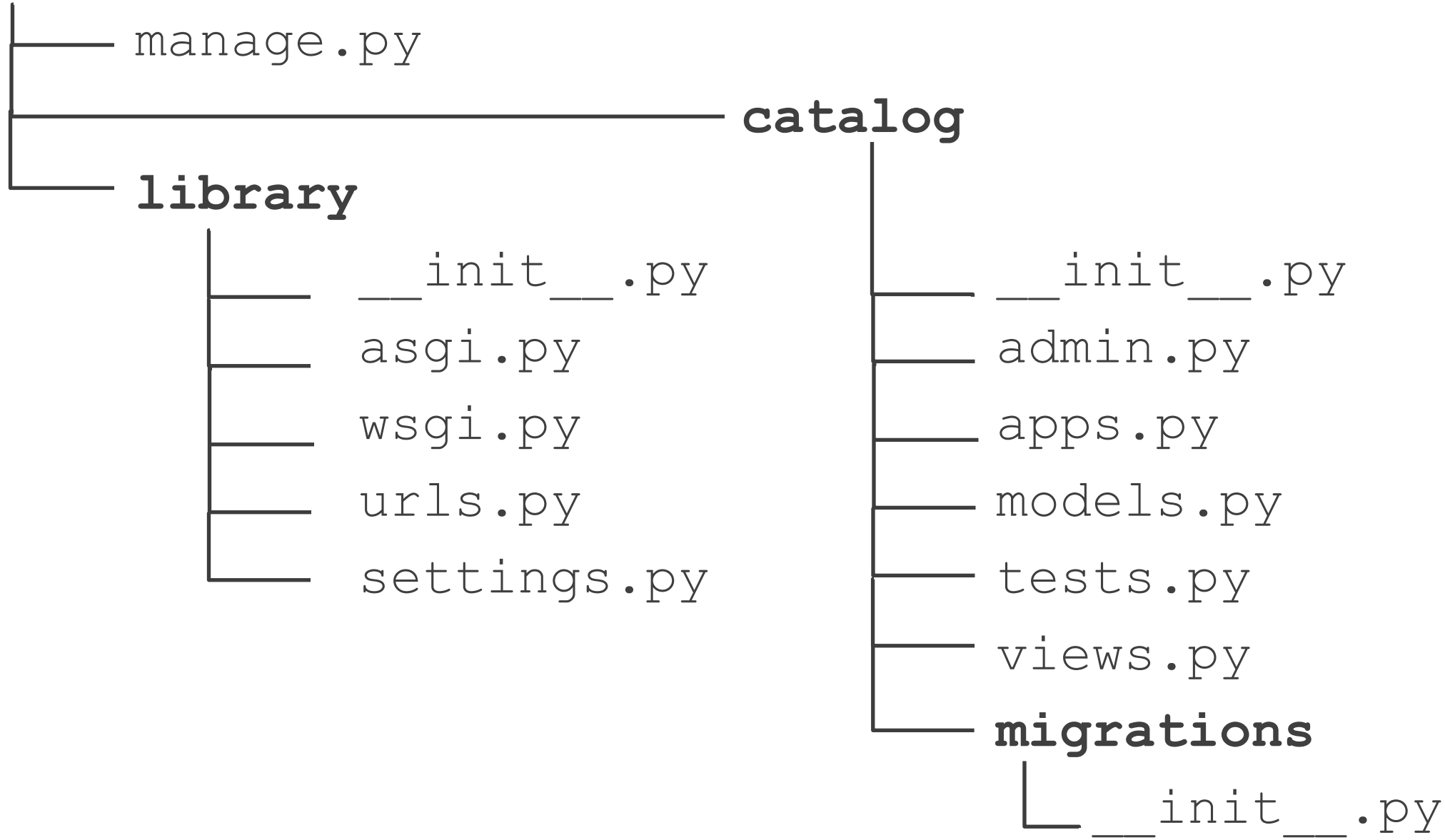
- `__init__.py` : vuoto, indica che la cartella e' un package.
  **FISSATO e non usato.**
- `asgi.py` : script di interfacciamento a server asincrono .
  **FISSATO e non usato.**
- `wsgi.py` : script di interfacciamento a server Tomcat.
  **FISSATO e non usato.**

# Django: cartella applicazione

La cartella di una applicazione viene creata con lo script di gestione:

```
cd library
python3 manage.py startapp catalog
```

- Esito: **library**



# Django: cartella applicazione

Files principali:

- `admin.py` : script di amministrazione della app.

  **Lo modificheremo.**

- `models.py` : contenitore delle definizioni dei modelli dati.

  **Lo modificheremo.**

- `views.py` : contenitore delle definizioni del codice di logica.

  **Lo modificheremo.**

# Django: cartella applicazione

Altri files:

- `tests.py`: file di gestione dei tests.

  Vuoto, non ci opereremo.
- `__init__.py` : vuoto, indica che la cartella e' di un package.

  FISSATO e non usato.
- `apps.py` : script di configurazione della app.

  FISSATO e non usato.
- `migrations` : directory di definizione dei DB, gestita da Django

  NON vi opereremo.

# Django: settaggi base e registrazione app

Per registrare la app nel progetto: edit di `settings.py`, aggiungendo in coda a `INSTALLED_APPS` la classe creata da Django in `apps.py`:

```
INSTALLED_APPS = [ ..., "catalog.apps.CatalogConfig" ];
```

Inoltre, in `settings.py`, si possono settare:

- `DATABASES`: quale DB usare (il default `SQLITE` e' ok)
- `TIME_ZONE` (consigliato per noi: `"CET"`)
- `LANGUAGE_CODE` (se si vuole sito in italiano: `"it-IT"`)

**Status quo: settata infrastruttura generica per la app Catalog**

# Ulteriori strumenti di supporto:

- Necessario: un editor
  - Motivazione: scrivere/modificare files HTML, CSS, JS, Python
  - Opzioni: **`gedit`**, ma anche molti altri (**`notepad++`**, **`gedit`**, ...)

- Consigliato: app per analisi differenziale di files/cartelle
  - Motivazione: insieme ai materiali forniti, permette una piu' rapida comprensione della evoluzione della web-app guida
  - Opzioni: **`kdiff3`** , ma anche altri (**`meld`**, **`winmerge`**, ...)

# Roadmap versioni

- Versione 1: Sito base (libri e autori, home + 2 liste)
- Versione 2: Gestione files statici + migliorie grafiche
- Versione 3: Paginazione
- Versione 4: Completamento del modello dati
- Versione 5: Viste di dettaglio
- Versione 6: Pagina amministrativa customizzata
- Versione 7: Con uso dei cookies di sessione

# Roadmap versioni

- Versione 8: Con riconoscimento utenti autenticati
- Versione 9: Con permessi specifici di accesso alle pagine
- Versione 10: Con pagine con moduli (forms)
- Versione 11: Con ricerca per titolo (SQL e filtri Django)
- Versione 12: Con bottoni "metti in manutenzione" (AJAX)
- Versione 13: Con testing integrato

# Design del sito (versione 1)

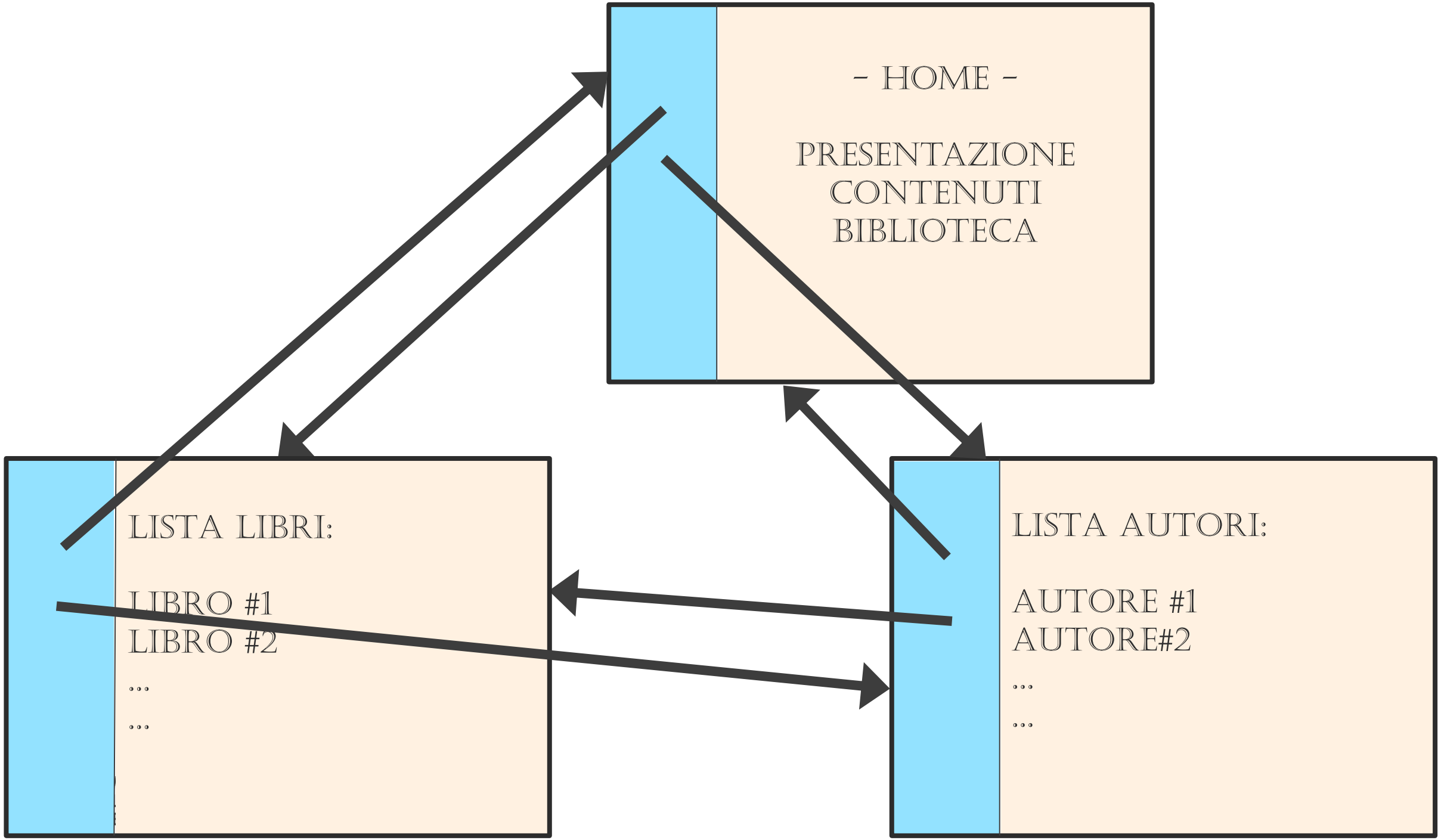
Il sito rappresenta una libreria locale in cui:

- Ogni libro e' caratterizzato da:
  - Titolo
  - Autore
  - Sommario
  - Codice (univoco) ISBN
- Ogni autore e' caratterizzato da:
  - Nome, Cognome
  - Data di nascita, Data di morte

# Design del sito (versione 1)

Nella prima versione, il sito avra' 3 pagine:

- **Home page** (`catalog/`)
  - Al centro mostra un benvenuto
  - A sinistra, un menu permette di accedere alle 3 pagine
- **Lista di tutti i libri** (`catalog/books/`)
  - Al centro mostra la lista di tutti i libri, con autore e titolo
  - A sinistra, un menu permette di accedere alle 3 pagine
- **Lista di tutti gli autori** (`catalog/authors/`)
  - Al centro mostra la lista di tutti gli autori, con nome e cognome
  - A sinistra, un menu permette di accedere alle 3 pagine


- HOME -
PRESENTAZIONE
CONTENUTI
BIBLIOTECA
LISTA LIBRI:
LIBRO #1
LIBRO #2
...
...
LISTA AUTORI:
AUTORE #1
AUTORE#2
...
...

# Design del modello relazionale

- Per ora ci basta modellare Autori e Libri
- Modellazione semplice, con l'ovvio legame uno-a-molti

# Django: modello relazionale

- Obiettivo: definire il modello relazionale via ORM

- Due passi:
  - Descrizione del modello (in **`models.py`**)
  - Registrazione del modello (in **`admin.py`**)

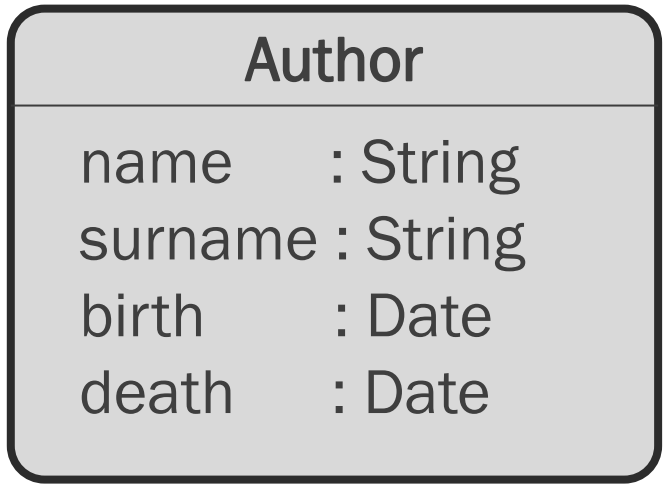
- Documentazione di riferimento:
  https://docs.djangoproject.com/it/5.1/**topics/db/models/**

# Django: modello relazionale

- Il modello (una volta progettato) va descritto in **`models.py`**
- Ogni struttura e' una classe Python che eredita da `Models.model`
- La classe definisce:
  - I campi
  - Le relazioni fra la classe in oggetto ed altre classi del modello
  - I metodi della classe
  - I metadati

# Django: modello relazionale

- Per procedere incrementalmente, partiamo dai soli autori:




- Tale modello "non necessita di relazioni", dunque definiremo:
  - Campi
  - Metodi
  - Metadati

# Django: modellazione dei dati

- Campi:
  - Un campo e' definito da un nome e da un tipo
  - Django (con la classe `Models`) supporta circa 20 tipi; per ora:
    - `CharField` : campo testuale (su una linea)
    - `DateField` : data

# Django: modellazione dei dati

- Le definizioni di dati (e delle relazioni) ammettono degli **attributi**
- Esistono attributi globali ed altri specifici del tipo/relazione
- Per ora considereremo:
  - `blank` : (globale) se puo' esser vuoto o no
  - `null` : (globale) se il valore vuoto sia salvato come `null`
  - `max_length` : lunghezza massima di tipi `CharField`
  - `verbose_name`: (globale) nome del campo in interfaccia. Puo' venir dato come stringa in primo parametro, od anche omesso.

# Django: modellazione dei dati

- metodi: per ora ci limitiamo a un metodo che va definito:
  - `__str__ (self)` : contenuto come deve apparire nel sito Admin

- metadati:
  - Riguardano diversi aspetti
  - Dichiarati come attributi nella sottoclasse `Meta` del modello
  - Per ora ci limitiamo a un attributo:
    - `ordering` : lista di campi per ordinamento in visualizzazione

# Django: modellazione dei dati

```
class Author(models.Model):
  first_name    = models.CharField(max_length=100)
  last_name     = models.CharField(max_length=100)
  date_of_birth = models.DateField('Born', null=True, blank=True)
  date_of_death = models.DateField('Died', null=True, blank=True)

  class Meta:
        ordering = ['last_name', 'first_name']

  def __str__(self):
        return f'{self.last_name}, {self.first_name}'
```

# Django: modellazione dei dati

```
class Author(models.Model):
  first_name    = models.CharField(max_length=100)
  last_name     = models.CharField(max_length=100)
  date_of_birth = models.DateField('Born', null=True, blank=True)
  date_of_death = models.DateField('Died', null=True, blank=True)

  class Meta:
        ordering = ['last_name', 'first_name']

  def __str__(self):
        return f'{self.last_name}, {self.first_name}'
```

"verbose_name": Nomi espliciti per rappresentazione

# Registrazione del modello dati

- I modelli (in questo caso `Author`) vanno registrati nella app
- Si modifica **`admin.py`** in modo che li importi e registri:

```
from .models import Author

admin.site.register(Author)
```

# Registrazione del modello dati

- Definito il modello, va creato/aggiornato il DB, usando **`manage.py`**

```
python3 manage.py makemigrations
python3 manage.py migrate
```

- Quel che succede:
  - Viene creato un database
  - Per ogni modello **`M`** nella app **`A`** viene creata una tabella **`A_M`**
  - La tabella **`A_M`** contiene i campi specificati nel modello **`M`**
  - Volendo, potremo poi accedere alle tabelle con SQL
  - In pratica, Django ci dara' strumenti per bypassare SQL

# Istanziazione del modello dati

- Obiettivo: istanziare i dati (creare gli autori)
- Modalita': ci avvaliamo della pagina di admin di Django

- Primo passo: creare un superuser; ci pensa `manage.py`:

```
python3 manage.py createsuperuser
```

- Immesse le credenziali si avvia il server locale:

```
python3 manage.py runserver
```

- ...e si accede da browser alla pagina admin:

```
http://127.0.0.1:8000/admin
```

- Ora come superuser possiamo gia' creare gli autori!

# Django: modello relazionale

- Secondo passo: immettiamo anche il modello dei libri:


- Tale modello porta a dover introdurre una relazione, dunque:
  - Campi
  - Relazioni
  - Metodi
  - Metadati

# Django: modellazione delle relazioni

- Relazioni:
  - Django permette di modellare relazioni:
    - 1-to-1 : (`OneToOneField`)
    - Many-to-1 (`ForeignKey`)
    - Many-to-many (`ManyToManyField`)
  - Ogni relazione:
    - E' modellata come fosse un campo
    - E' direzionale
    - Definisce (anche implicitamente) campi nelle classi relate

# Django: modellazione delle relazioni

- Caso 1-to-1:
  - Se ad una istanza di **`A`** corrisponde una di **`B`**, ed al piu' viceversa, pongo in **`A`**:

    ```
    rel_b = OneToOneField(B)
    ```
  - La dichiarazione puo' anche avere attributi (vedremo dopo)
  - A questo punto:
    - Una istanza **`ia`** di **`A`** avra' un campo **`ia.rel_b`** di tipo **`B`**
    - Una istanza **`ib`** di **`B`** puo' avere un *'campo'* **`ib.a`** di tipo **`A`**
    - Se lo abbia o no e' definito dalla funzione `hasattr(ib,'a')`

# Django: modellazione delle relazioni

- Caso 1-to-1:
  - Se ad una istanza di **A** corrisponde una di **B**, ed al piu' viceversa, pongo in **A**:

    ```
    rel_b = OneToOneField(B)
    ```

  - La dichiarazione puo' anche avere attributi (vedremo dopo)
  - A questo punto:
    - Una istanza **ia** di **A** avra' un campo **ia.rel_b** di tipo **B**
    - Una istanza **ib** di **B** puo' avere un *'campo'* **ib.a** di tipo **A**
    - Se lo abbia o no e' definito dalla funzione `hasattr(ib,'a')`

Nome costruito da Django: lower-case di **A**

- **Esempio di caso 1-to-1**:
  - Ad un negozio `Shop` e' associato univocamente il luogo `Place`

```
class Shop(models.Model):
    ...
    pl   = models.OneToOne(Place [,...])
```

  - A questo punto:
    - Il luogo di uno shop `s` e' acceduto come `s.pl`
    - Un luogo `p` puo' essere uno shop: lo dice `hasattr(p,shop)`
    - Nel caso lo sia, il suo shop e' `p.shop`
    - Posso assegnare il luogo di uno shop con `s.pl = p`
    - Posso dire che un luogo e' legato a uno shop con `p.shop = s`

# Django: modellazione delle relazioni

- Caso many-to-1:
  - Se ad una istanza di **A** corrisponde una di **B**, ma ad una di **B** ne possono corrispondere arbitrarie di **A**, pongo in **A**:

    ```
    rel_b = ForeignKey(B)
    ```
  - La dichiarazione puo' anche avere attributi (vedremo dopo)
  - A questo punto:
    - Una istanza **ia** di **A** avra' un campo **ia.rel_b** di tipo **B**
    - Una istanza **ib** di **B** avra' il *'campo'* **ib.a_set** (insieme di **A**)
    - Il suo contenuto e' **ib.a_set.all()** e puo' essere vuoto

# Django: modellazione delle relazioni

- Caso many-to-1:
  - Se ad una istanza di **A** corrisponde una di **B**, ma ad una di **B** ne possono corrispondere arbitrarie di **A**, pongo in **A**:

    **rel_b** = ForeignKey(**B**)

  - La dichiarazione puo' anche avere attributi (vedremo dopo)
  - A questo punto:
    - Una istanza **ia** di **A** avra' un campo **ia.rel_b** di tipo **B**
    - Una istanza **ib** di **B** avra' il '*campo*' **ib.a_set** (insieme di **A**)
    - Il suo contenuto e' **ib.a_set.all()** e puo' essere vuoto

Nome costruito da Django: lower-case di **A**

- **Esempio di caso many-to-1**:
  - Ad un libro `Book` e' associato l'autore `Author` (di arbitrari libri)

```
class Book(models.Model):
    ...
    auth   = models.ForeignKey(Author [,...])
```

  - A questo punto:
    - L'autore del libro `b` e' acceduto come `b.auth`
    - I libri dell'autore `a` sono contenuti in `a.book_set`
    - In particolare la loro totalita' (anche 0) e' `a.book_set.all()`
    - Posso assegnare l'autore di un libro `b` con `b.auth= a`
    - Posso aggiungere un libro ad un autore: `a.book_set.add(b)`

# Django: modellazione delle relazioni

- Caso many-to-many:
  - Se ad una istanza di **A** corrispondono arbitrarie di **B**, e viceversa, pongo in **A**:

    ```
    rel_b = ManyToManyField(B)
    ```
  - La dichiarazione puo' anche avere attributi (vedremo dopo)
  - A questo punto:
    - Una istanza **ia** di **A** avra' un campo **ia.rel_b**
    - Una istanza **ib** di **B** avra' il *'campo'* **ib.a_set**
    - Entrambi i campi sono insiemi, accessi col metodo `.all()`

# Django: modellazione delle relazioni

- Caso many-to-many:
  - Se ad una istanza di **A** corrispondono arbitrarie di **B**, e viceversa, pongo in **A**:

    ```
    rel_b = ManyToManyField(B)
    ```

  - La dichiarazione puo' anche avere attributi (vedremo dopo)
  - A questo punto:
    - Una istanza **ia** di **A** avra' un campo **ia.rel_b**
    - Una istanza **ib** di **B** avra' il *'campo'* **ib.a_set**
    - Entrambi i campi sono insiemi, accessi col metodo `.all()`

Nome costruito da Django: lower-case di **A**

- **Esempio di caso many-to-many**:
  - Un `Musician` puo' stare in molte `Band` (fatte di piu' `Musician`)

```
class Musician(models.Model):
    ...
    bands  = models.ManyToManyField(Band [,...])
```

  - A questo punto:
    - Le bands del musicista `m` sono denotate da `m.bands` e accesse, nella loro totalita', con `m.bands.all()`
    - I musicisti di una band `b` sono denotati da `b.musician_set` e accessi,nella loro totalita', con `b.musician_set.all()`
    - Per aggiungere una band al musicista `m`: `m.bands.add(b)`
    - Per aggiungere `m` alla band: `b.musician_set.add(m)`

- **Esempio di caso many-to-many**:
  - Od anche, simmetricamente:

```
class Band(models.Model):
    ...
    members  = models.ManyToManyField(Musician [,...])
```

  - A questo punto:
    - Le bands del musicista `m` sono denotate da `m.band_set` e accesse, nella loro totalita', con `m.band_set.all()`
    - I musicisti di una band `b` sono denotati da `b.members` e accessi,nella loro totalita', con `b.members.all()`
    - Per aggiungere una band al musicista `m`: `m.band_set.add(b)`
    - Per aggiungere `m` alla band: `b.members.add(m)`

# Django: modellazione dei dati

- Ulteriori tipi di dati che dovremo considerare:
  - `TextField` : campo testuale multilinea

- Ulteriori attributi che dovremo considerare:
  - `unique` : (globale) se un dato debba avere valore univoco
  - `help_text` : (globale) testo di help in interfaccia
  - `on_delete` : cosa accade se rif. di una relazione e' rimosso

```
class Book(models.Model):

    title   = models.CharField(max_length = 200)
    author  = models.ForeignKey(Author,
                                null       = True,
                                on_delete = models.SET_NULL)
    summary = models.TextField(max_length = 1000,
                               help_text  = 'Describe the book')
    isbn    = models.CharField('ISBN',
                               max_length = 13,
                               unique     = True,
                               help_text  = 'ISBN code')

    class Meta:
            ordering = ['title']

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

```
class Book(models.Model):

    title   = models.CharField(max_length = 200)
    author  = models.ForeignKey(Author,
                                null       = True,
                                on_delete = models.SET_NULL)
    summary = models.TextField(max_length = 1000,
                               help_text  = 'Describe the book')
    isbn    = models.CharField('ISBN',
                               max_length = 13,
                               unique     = True,
                               help_text  = 'ISBN code')

    class Meta:
            ordering = ['title']

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

Relazione 1-to-many
e effetti di cancellazione

# Django: modellazione dei dati

- La scrittura delle relazioni in `models.py` dipende dall'ordine!
- Una classe puo'  riferirsi, attraverso una relazione:
  - A classi "precedenti", come classe o con la stringa del nome
  - A classi "successive", solo con la stringa del nome
- Qui `Book` riferisce ad `Author` (non `"Author"`) perche' "viene dopo"

# Registrazione del modello dati

- Si modifica **admin.py** in modo che importi e registri anche `Book`:

```
from .models import Author, Book

admin.site.register(Book)
admin.site.register(Author)
```

# Registrazione del modello e istanziazione

- Definito il modello, va aggiornato il DB, **usando `manage.py`**

```
python3 manage.py makemigrations
python3 manage.py migrate
```

- Ora possiamo istanziare i libri e collegarli agli autori: avviamo il server con `python3 manage.py runserver` e accediamo da browser alla pagina admin: `http://127.0.0.1:8000/admin`

- Ora come superuser possiamo gia' creare e collegare libri e autori

# Uso del modello dati

- A questo punto, il codice che scriveremo (nelle views) puo':
  - Istanziare nuovi oggetti, ad es. `my_book = Book()`
  - Definirne campi e relazioni , es.

```
my_book.title = "Uto"
my_book.author = my_author
```

  - Salvare oggetti/modifiche nel DB: `my_book.save()`
  - Accedere agli insiemi di oggetti attraverso il loro *object manager*

# Uso del modello dati

- Object manager: la interfaccia per accedere ai dati di un modello
- Per ogni classe `C` in models.py, Django crea un manager `C.objects`
- Il manager dispone di molti metodi per interrogare il DB, fra i quali:
  - `all()` : restituisce, per default, tutti gli oggetti della classe:

    ```
    all_books = Book.objects.all()
    ```

    NB si può ridefinire `all()` scrivendo in modo custom il metodo `get_queryset` in `C`
  - `filter()`, `exclude()`: filtra o esclude rispetto a criteri definiti:
    - i criteri sono definiti da nome campo, criterio e valore
    - vi sono una dozzina di criteri a disposizione

      ```
      uto_books  = Book.objects.filter(title__exact = "Uto")
      old_car    = Car.objects.exclude(year__gt = 1970 )
      good_bottles = Wine.objects.filter(year__in = [2003,2007])
      ```

Vedi https://docs.djangoproject.com/en/5.1/ref/models/querysets/

# Uso del modello dati

- Note rilevanti:
  - Le modifiche dei dati non impattano sul DB fino al `save`
  - Si puo' creare/aggiornare relazioni solo quando esse riferiscono ad oggetti gia' presenti nel DB (ovvero: prima fare il `save` ...)
  - Le relazioni "a 1" sono modificate per assegnamento; quelle "a molti" per aggiunta attraverso `add` :

```
my_book.author = auth
my_author.book_set.add(my_book)
```

  - Non serve modificare una relazione su entrambi i lati!

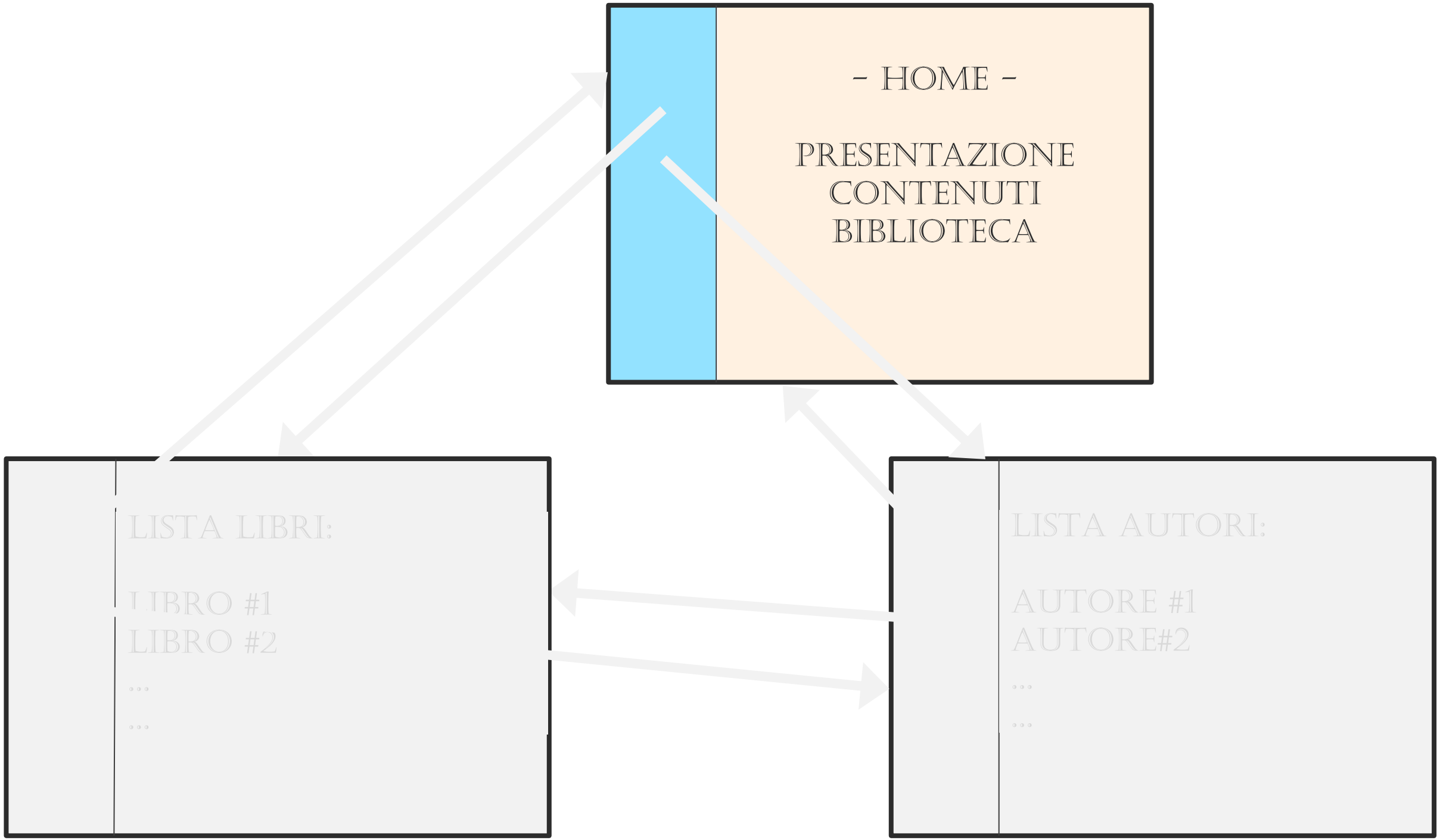
# Pagine: URL mappings, viste e templates

- Remind: struttura del nostro sito:
  - `catalog/` : Home page
  - `catalog/books/` : Lista di tutti i libri
  - `catalog/authors/` : Lista di tutti gli autori

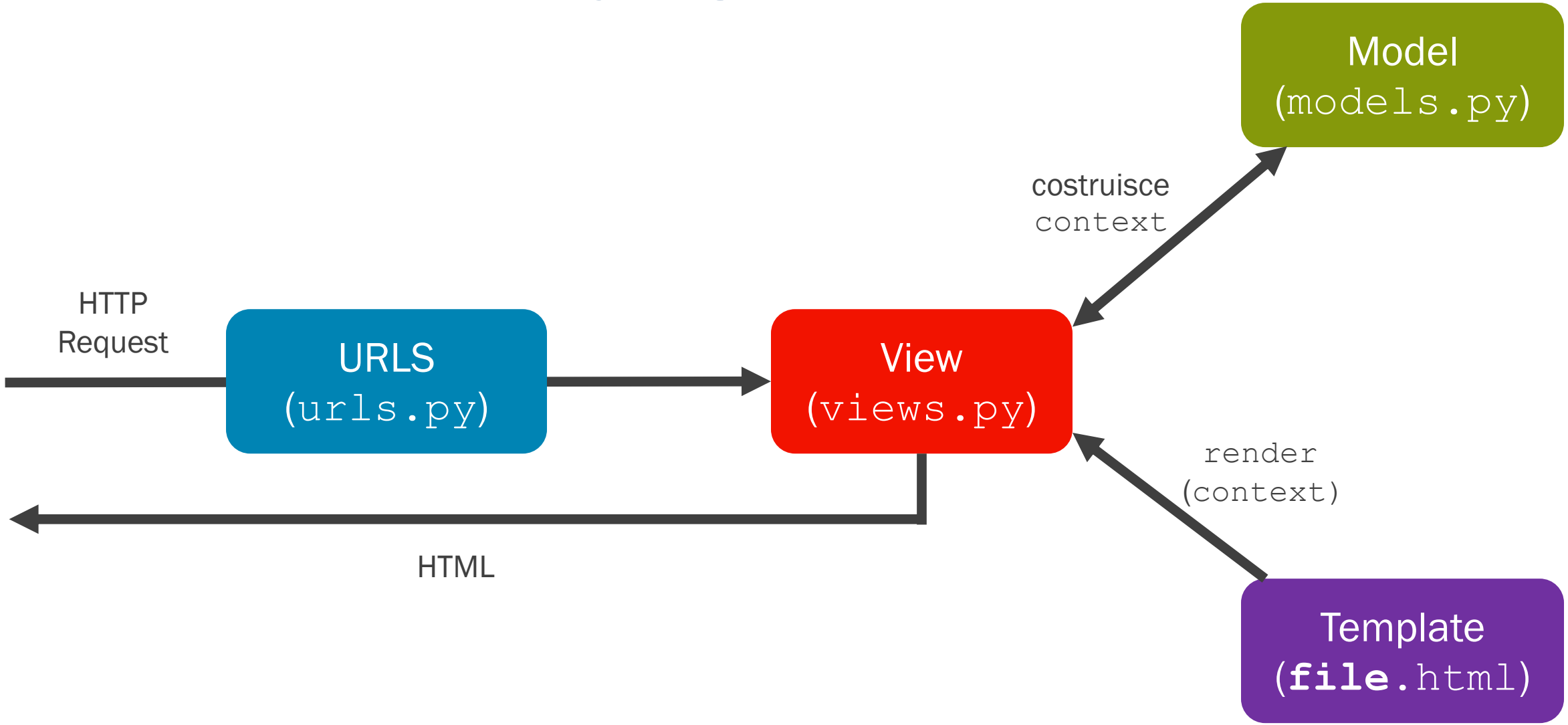
- Per ogni pagina dobbiamo:
  - Definire una view
  - Definire un template
  - Definire uno URL mapping

# Pagine: viste, templates e mappings

- Dunque, un possibile processo di sviluppo per Django e':
  - Definisci il modello dati (M di MTV)
  - In un qualche ordine, definisci T + V + URL map per ogni pagina
  - Notare che questo permette sviluppo parallelo delle pagine
  - Noi faremo, per procedere incrementalmente:
    - T + V + URL map per la Home page
    - Check della home page
    - T + V + URL map per le altre 2 pagine
    - Check del sito "con 3 pagine"


- HOME -
PRESENTAZIONE
CONTENUTI
BIBLIOTECA
LISTA LIBRI:
LIBRO #1
LIBRO #2
...
...
LISTA AUTORI:
AUTORE #1
AUTORE#2
...
...

# Django recap :



# Views

- Contengono la logica che:
  - Data in input una una richiesta HTTP...
  - ...accedendo ai dati del DB e/o usando templates...
  - ...restituisce l'HTML della pagina richiesta

- Espresse in Python nel file **`views.py`**, in uno di due modi:
  - Come funzioni (function-based views: FBVs); rif.:

https://docs.djangoproject.com/it/5.1/**topics/http/views/**

  - Basandosi su classi di supporto (class-based views: CBVs); rif.:

https://docs.djangoproject.com/it/5.1/**topics/class-based-views/**

# Views

- Modi che ha una view **`V`** per costruire l'HTML
  - Farlo "ad hoc" costruendo "la stringa HTML"
  - Redirigere il compito ad un'altra pagina
  - Appoggiarsi ad un template **`T`:**
    - Costruendo un "contesto" **`C`** di nomi-valori per riempirlo
    - Chiamando `render(`**`T`**`)` per integrare **`C`** in uno schema HTML

# Templates

- Un template definisce l'HTML relativo ad una pagina
- Lo costruisce usando un contesto di input
- Per fare cio', integra HTML con un linguaggio detto DTL:
  - DTL : Django Template Language
  - Contempla direttive imperative (`for`, `if-then`, ...) ed altro
  - Permette di riferirsi a dati fornitigli come contesto
  - DTL : espresso fra "`{%`" e "`%}`" nell'HTML

# View della home page

- Definiamo la view della home in modo function-based (FBV)
- La funzione:
  - Riceve un parametro che modella la HTTP request in arrivo
  - Deve produrre esplicitamente un HTML
  - Puo' accedere agli oggetti del modello dati (ovvero: al DB)

# View della home page

- Input della funzione: parametro `request`
- Contiene molteplici campi:
  - `METHOD`: `GET`/`POST`/...
  - `GET` : contenuto dei parametri della HTTP GET
  - `POST`: contenuto della HTTP POST
  - `COOKIES`: dizionario dei cookies
  - `META / headers` : dizionario dei valori negli headers
  - `user` : modella l'utente loggato
  - `session` : modella la sessione del client
  - ....

# View della home page

- Output della funzione: un HTML che puo' essere costruito :
  - Ad hoc, e poi ritornato come HTTP `HttpResponse`
  - Funzione `redirect` per redirezione
  - Funzione `render` che:
    - utilizza un template scritto in HTML + DTL
    - gli passa un "dizionario" di nomi-valori
    - il DTL usa il dizionario per "riempire" l'HTML dove serve

# View della home page

- Logica interna nel caso tipico di delega dal FBV **`V`** a template **`T`** :
  - **`V`** passa un *contesto* (dizionario nomi-valori) a **`T`** via `render`
  - Il contesto serve a riempire il template producendo HTML
  - Tipicamente, il contesto e' costruito accedendo al modello

- Dunque:
  - La FBV **`V`** interpreta la richiesta
  - Costruisce (con logica su modello dati) il contesto **`ctx`**
  - Invoca `render` del template **`T`** con **`ctx`**

# View della home page

```
from .models import Book, Author

def index(request):
    n_books   = Book.objects.all().count()
    n_authors = Author.objects.all().count()

    ctx = {
           'num_books'   : n_books,
           'num_authors' : n_authors,
          }
     return render(request, 'index.html',context=ctx)
```

# View della home page

```
from .models import Book, Author

def index(request):
    n_books   = Book.objects.all().count()
    n_authors = Author.objects.all().count()

    ctx = {
           'num_books'   : n_books,
           'num_authors' : n_authors,
          }
    return render(request, 'index.html',context=ctx)
```

# View della home page

```
from .models import Book, Author

def index(request):
    n_books   = Book.objects.all().count()
    n_authors = Author.objects.all().count()

    ctx = {
           'num_books'   : n_books,
           'num_authors' : n_authors,
          }
    return render(request, 'index.html',context=ctx)
```

# Template della home

- I templates sono definiti come files di tipo `.html`
- Essi integrano direttive nel Django template language DTL
- Il DTL e' un vero e proprio linguaggio di programmazione
- Tipicamente le direttive usano il contesto passato dalla view
- Riferimento:

  https://docs.djangoproject.com/en/5.1/**topics/templates/**

# Template della home

- Conviene partire da un template comune `base_generic.html`

- Esso dara' la struttura comune a tutte le pagine:
  - Sidebar
  - Intestazione
  - Area contenuti

- Poi si usano i meccanismi di estensione di DTL per definire gli altri templates "partendo" da `base_generic.html`

# Template della home

- I templates di una app sono cercati da Django:
  - Di default, nel folder di app `templates` (da creare ex novo)
  - Oppure, volendo, altrove (settaggio in `settings.py`)
- Dunque:

# Template di base: struttura

```
<!DOCTYPE html>
<html lang="en">

HEAD

BODY

</html>
```

# Template di base: head

```
    <head>
      {% block title %}
          <title>Local Library</title>
      {% endblock %}
      <meta charset="utf-8">
      <meta name="viewport"
            content="width=device-width, initial-scale=1">
      <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/boot
strap.min.css" rel="stylesheet">
    </head>
```

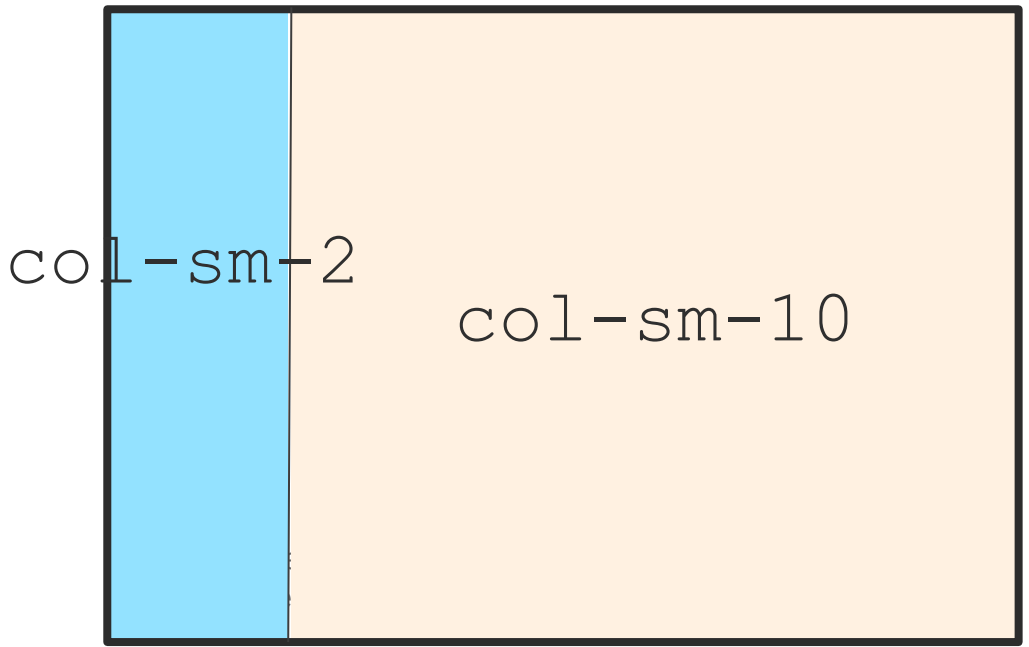
# Template di base: head

- Direttive DTL in questa parte: `block` / `endblock`
  - Realizza un meccanismo di "ereditarieta' dei templates"
  - Descrive un "blocco" (con nome: `title`)
  - Un template "figlio" **puo'** rimpiazzare tale blocco

- Nota: per ora importiamo la libreria `bootstrap` per layout

# Template di base: head

- `bootstrap` e' una notissima libreria di cui noi usiamo la parte CSS
  - `container-fluid` definisce una classe "contenitore adattivo"
  - `row` definisce una classe per una grid di layout di 12 colonne
  - Assegneremo le prime 2 (con `col-sm-2`) al menu laterale e le altre 10 (`col-sm-10`) al contenuto



# Template di base: body

```
<body>
  <div class="container-fluid">
    <div class="row">
      <div class="col-sm-2">
        {% block sidebar %}
            <a href="{% url 'index' %}">Home</a>
        {% endblock %}
      </div>
      <div class="col-sm-10">{% block content %}{% endblock %}</div>
    </div>
  </div>
</body>
```

# Template di base

- Nome: `base_generic.html` , nel folder `templates`
- Menu laterale: blocco `sidebar`
  - Contiene per ora un link alla sola home
  - Direttiva DTL `url` : dal nome di URL-mapping rida' il suo URL

    (...stiamo gia' definendo il nome del mapping per la home)
- Contenuto: blocco `content` (vuoto: poi lo riempiremo)
- Dunque, struttura con 3 blocchi: `title`, `sidebar`, `content`

# Template della home

- La chiameremo, come definito nella chiamata della FBV, `index.html`, e sara' sotto `templates`
- Estendera' , con DTL, il template `base_generic.html`
- In particolare ne riscrivera' il blocco (vuoto) `content` usando le variabili di contesto che riceve dalla sua FBV

# Template della home

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Local Library Home</h1>
  <p>Welcome to Local Library!</p>
  <h2>Dynamic content</h2>
  <p>The library has the following record counts:</p>
  <ul>
    <li><strong>Books:</strong> {{ num_books }}</li>
    <li><strong>Authors:</strong> {{ num_authors }}</li>
  </ul>
{% endblock %}
```

# Template della home

- In questo template:
  - Direttiva `extends`: eredita da un template (`base_generic.html`)
  - Uso di `block`/`endblock` per riscrivere il blocco `content`
  - Le variabili di contesto sono descritte con la sintassi `{{ VAR }}`

# URL mappings

- Adesso dobbiamo definire lo URL mapping che collega `catalog/` alla sua function-based view...

# URL mappings

- Un file di mapping consiste di:
  - Import di classi di supporto Django
  - Definizione dei mappings nell'array `urlpatterns`

- `urlpatterns` : una lista di regole, ognuna che descrive:
    - Criterio di selezione di URL (*con linguaggio di matching*)
    - View da chiamare se il criterio e' soddisfatto
    - Opzionalmente, il nome del mapping

- Rif.: https://docs.djangoproject.com/en/5.1/topics/**http/urls/**

# URL mappings

Es.
```
urlpatterns =
    [
      path('articles/2003/', views.special_case_2003),
      path('articles/<int:year>/', views.y_arch),
      path('articles/<int:year>/<int:month>/', views.m_arch)
    ]
```

- Attivato da richiesta HTTP su URL **`url`**
- Le regole vengono verificate una ad una in ordine contro **`url`**
- La prima soddisfatta da **`url`** viene usata, e la ricerca finisce
- Se nessuna viene soddisfatta da **`url`**: eccezione da Django

# URL mappings

Es.
```
urlpatterns =
    [
      path('articles/2003/', views.special_case_2003),
      path('articles/<int:year>/', views.y_arch),
      path('articles/<int:year>/<int:month>/', views.m_arch)
    ]
```

Qui:

- Il primo path e' rispetto a un URL "fissato"
- Gli altri due, oltre a fare URL matching, assegnano variabili

# URL mappings

Es.
```
urlpatterns =
    [
      path('articles/2003/', views.special_case_2003),
      path('articles/<int:year>/', views.y_arch),
      path('articles/<int:year>/<int:month>/', views.m_arch)
    ]
```

Ad esempio, sulla URL `/articles/2005/03/` :

- Le prime due regole falliscono, la terza e' soddisfatta
- Django chiama `views.m_arch` con `year=2005` e `month=3`

# URL mappings: dove?

- Contenuti nel file URL mapping **di progetto** `urls.py`
- Esso puo' includere altri mappings (**di app**): ed e' consigliato!
- La delega avviene in `urlpatterns` con `include`:

```
from django.urls import include
urlpatterns += [path('catalog/', include('catalog.urls')),]
```

- Quando un path usa `include`: se lo URL inizia con la regola, la ricerca viene fermata e lo URL "rimanente" passato al file delegato

# URL mappings

- Un ulteriore aspetto da gestire: ridirezione della pagina base
- Su  di essa, che matcha con la stringa vuota, si chiama il metodo specifico `RedirectView.as_view` che la ridirige a una URL

```
from django.views.generic import RedirectView

urlpatterns += [ path('', RedirectView.as_view(url='catalog/')),]
```

# URL mappings

- A questo punto possiamo scrivere lo URL mapping di app
- File `catalog/urls.py` creato ex-novo
- Deve importare il modulo `path` e `views` e definire `urlpatterns`:

```
from django.urls import path
from . import views

urlpatterns = [ path('' , views.index  , name='index'  )]
```

# URL mappings

- A questo punto possiamo scrivere lo URL mapping di app
- File `catalog/urls.py` creato ex-novo
- Deve importare il modulo `path` e `views` e definire `urlpatterns`:

```
from django.urls import path
from . import views

urlpatterns = [ path('' , views.index  , name='index'  )]
```

Nome della regola di mapping

# Test!

- Siamo finalmente in grado di testare la home!

- Routine di esecuzione server:

  `python3 manage.py makemigrations`

  `python3 manage.py migrate`

  `python3 manage.py runserver`

  Apri un browser

  Pulisci la cache

  Apri la pagina `http://127.0.0.1:8000`

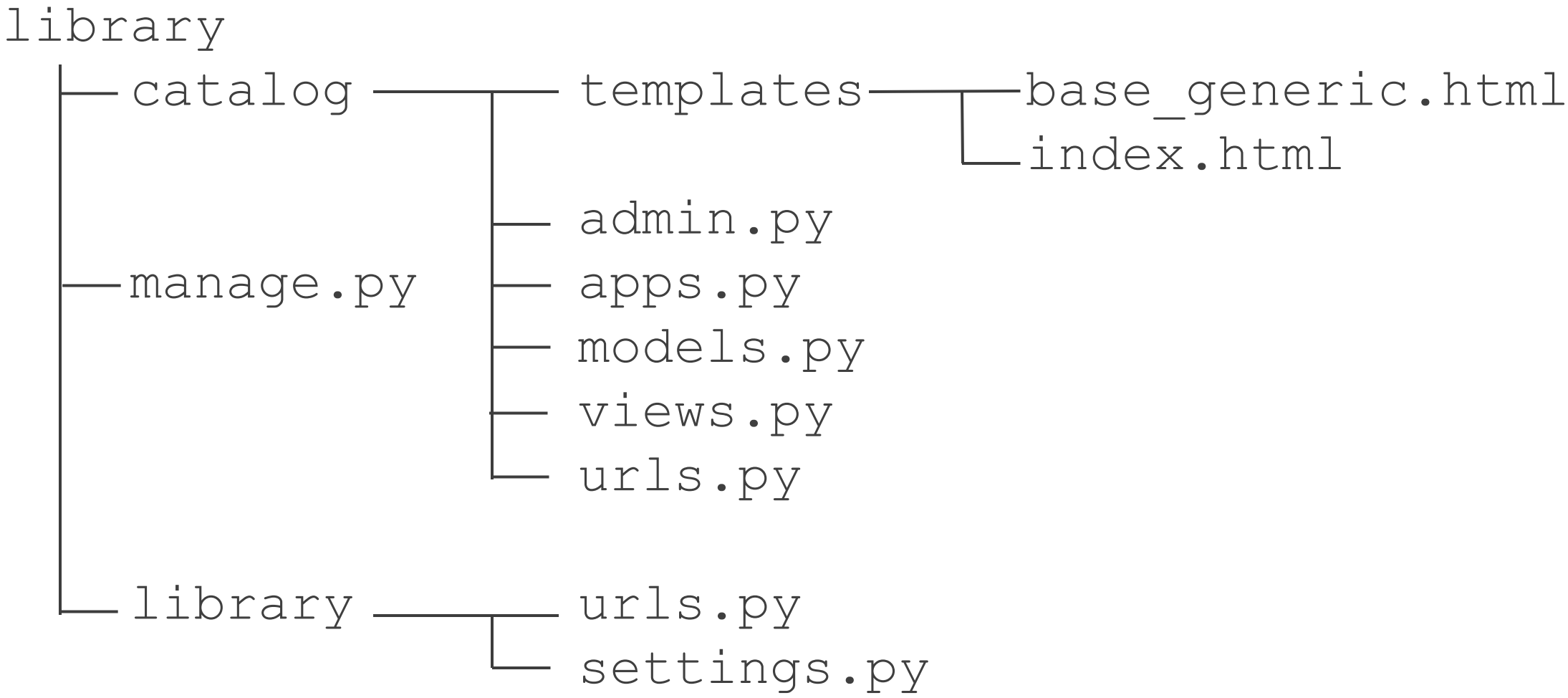
# Test!

- Note:
  - Il debugger del browser evidenza un problema con `favicon.ico`.
    Non è cruciale e ce ne occuperemo in una fase successiva.
  - Per il test, non serve sempre ricostruire il DB (se il modello e' fisso...), e tantomeno è sempre necessario pulire la cache. Nel caso, routine semplificata:

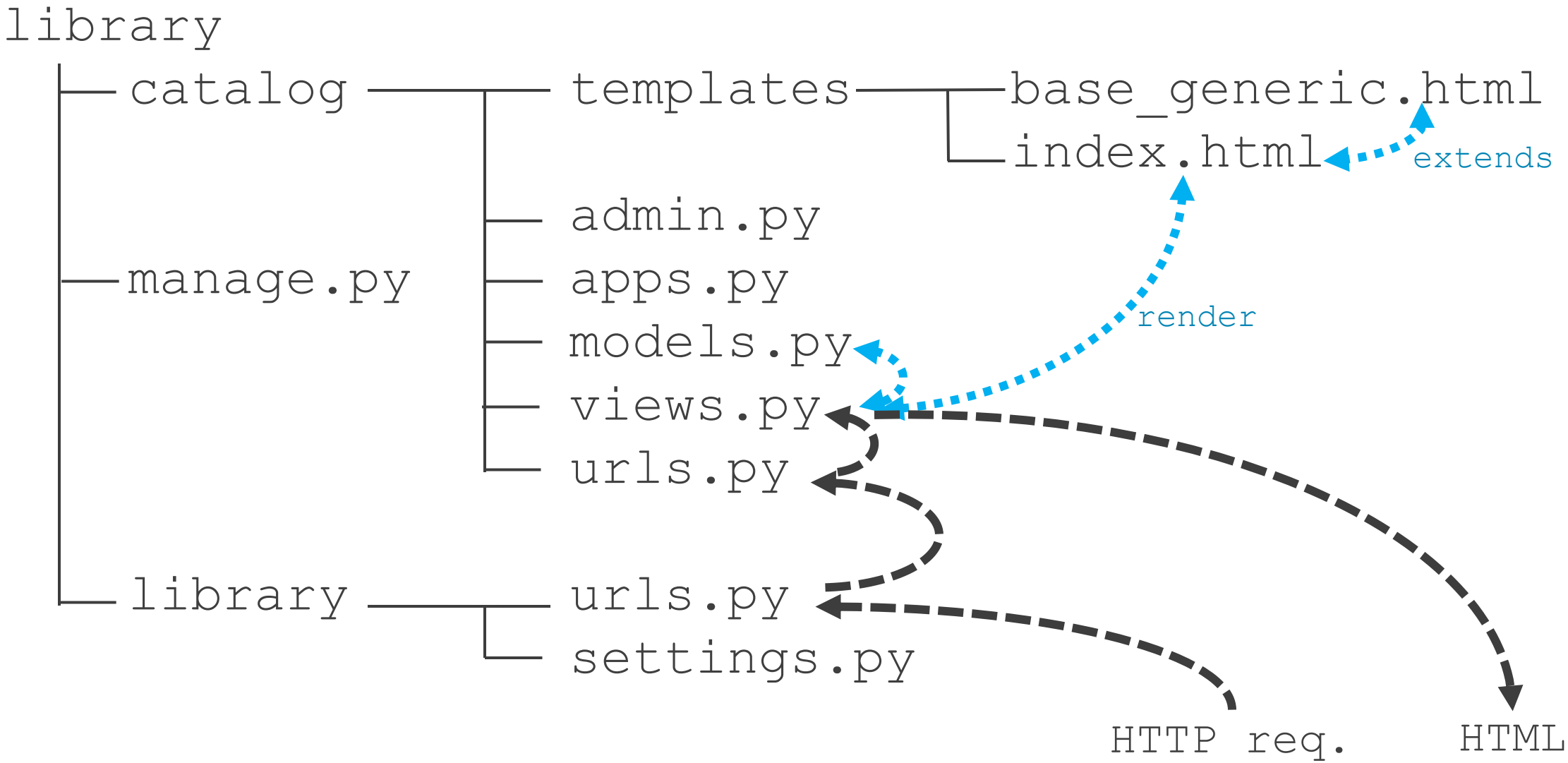
```
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

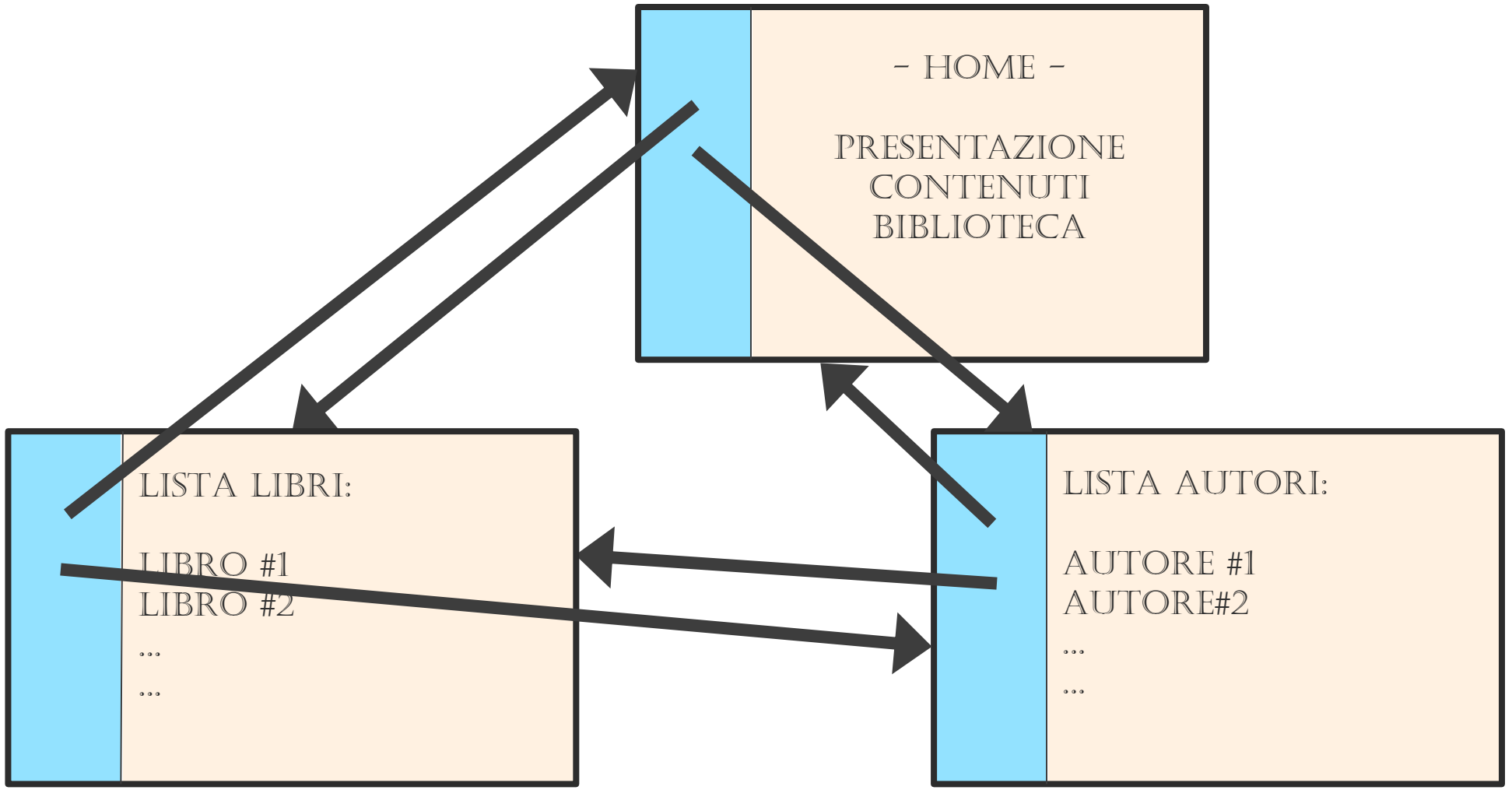
# Recap: struttura files e lifecycle



# Recap: struttura files e lifecycle



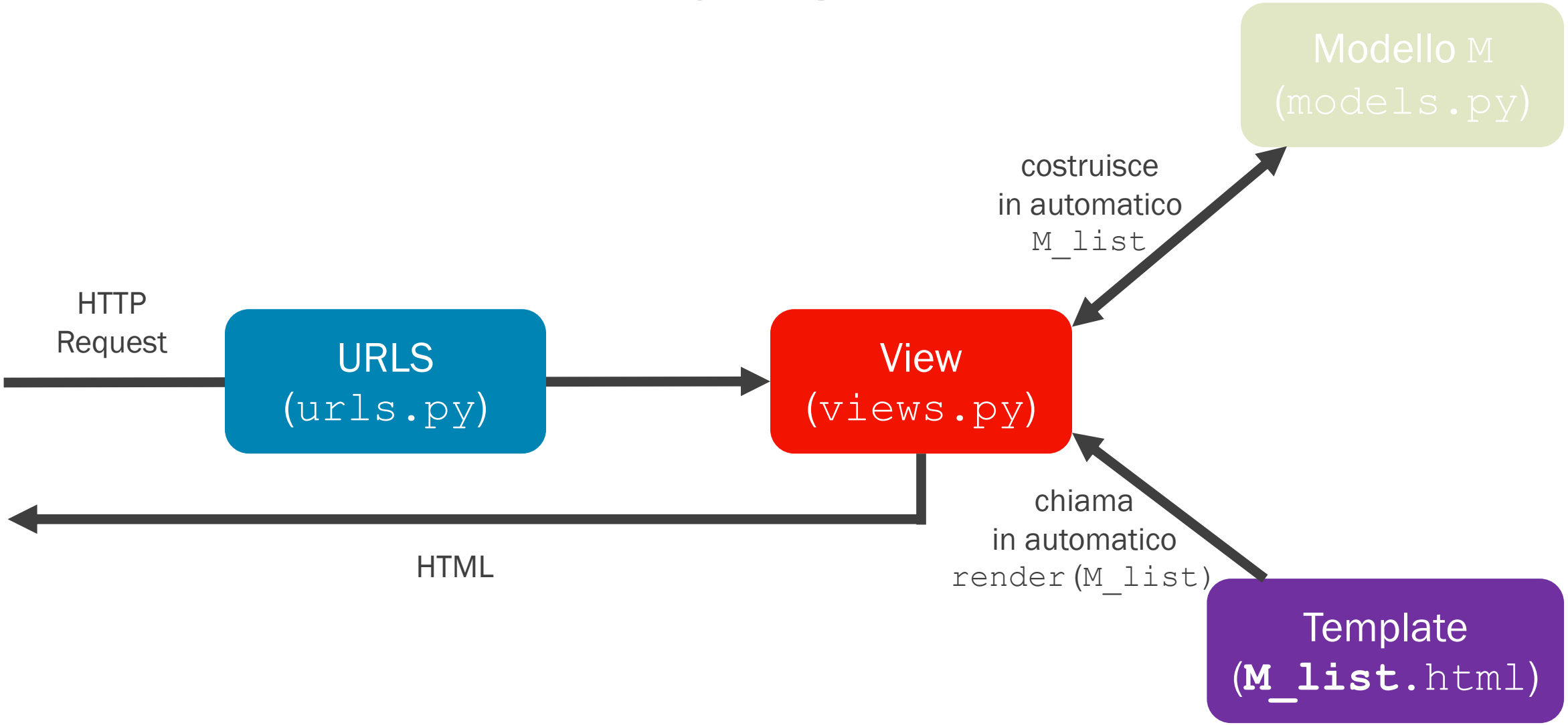
# Obiettivo


- HOME -
PRESENTAZIONE
CONTENUTI
BIBLIOTECA
LISTA LIBRI:
LIBRO #1
LIBRO #2
...
...
LISTA AUTORI:
AUTORE #1
AUTORE#2
...
...

# View della pagina libri

- Django fornisce classi per le view di liste: conviene una CBV
- Rif. : https://docs.djangoproject.com/en/5.1/**ref/class-based-views/**
- Classe di supporto specifica: `ListView`
  - Si aspetta un modello dati di riferimento **`M`** (attributo `model`)
  - Costruisce un contesto **`M`**`_list` (lower-case) di istanze di **`M`**
  - Invoca il template **`M`**`_list.html` (lower-case) con il contesto
  - Lo situa in `templates/`**`app`**`/` (da creare ex novo)
  - La CBV andra' poi invocata con il metodo `.as_view()`

# Django :


Modello M
(models.py)
costruisce
in automatico
M_list
HTTP
Request
URLS
(urls.py)
View
(views.py)
chiama
in automatico
render(M_list)
Template
(M_list.html)
HTML

# View della pagina libri

```
from django.views import generic

class BookListView(generic.ListView):
      model = Book
```

- Invochera' il template `templates/catalog/book_list.html`
- Gli passera' un contesto `book_list` : una lista di oggetti `Book`
- Va invocato con `BookListView.as_view()`

# Template della pagina libri

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Book List</h1>
  {% if book_list %}
    <ul>
      {% for book in book_list %}
        <li>
          <b>{{ book.title }}</b> ({{ book.author }})
        </li>
      {% endfor %}
    </ul>
  {% else %}
    <p>There are no books in the library.</p>
  {% endif %}
{% endblock %}
```

# Template della pagina libri

- Questo e' il template `book_list.html`
  - Ridefinisce il blocco content
    - Direttiva DTL '`if/else/endif`' :
      - Valuta espressione booleana **`expr`**
      - Se **`expr`** e' vera, viene emesso il blocco contenuto in `if`
      - Se e' falsa, viene emesso il blocco contenuto in `else`
    - Direttiva DTL '`for/endfor`'
      - Cicla sugli elementi di un array, istanziandoli in una var **`v`**
      - Emette per ogni ciclo un blocco HTML dove puo' usare **`v`**

# View della pagina autori

- View e template saranno analoghi a quelli visti per i libri
- View (class-based):

```
class AuthorListView(generic.ListView):
      model = Author
```

- Template `author_list.html` : vedi slide successiva

# Template della pagina autori

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Author List</h1>
  {% if author_list %}
     <ul>
     {% for author in author_list %}
       <li>
         <b>{{ author.last_name }}</b> , ( {{author.first_name}} )
       </li>
      {% endfor %}
     </ul>
  {% else %}
     <p>There are no authors in the library.</p>
  {% endif %}
{% endblock %}
```

# Views e templates: recap

- Abbiamo definito, per le tre pagine:
  - Le views:
    - Function-based view `index` per la home
    - Class-based view `BookListView` per la lista libri
    - Class-based view `AuthorListView` per la lista autori
  - I templates (tutti basati su `base_generic.html`, da aggiornare)
    - `index.html` per la home (in `templates`)
    - `book_list.html` per i libri (in `templates/catalog/`)
    - `author_list.html` per gli autori ( " " " " )

# Aggiornamento di `base_generic.html`

```
<body>
  <div class="container-fluid">
    <div class="row">
      <div class="col-sm-2">
        {% block sidebar %}
          <ul class="sidebar-nav">
            <li><a href="{% url 'index' %}">Home</a></li>
            <li><a href="{% url 'books' %}">All books</a></li>
            <li><a href="{% url 'authors' %}">All authors</a></li>
          </ul>
        {% endblock %}
      </div>
      <div class="col-sm-10">{% block content %}{% endblock %}</div>
    </div>
  </div>
</body>
```

# URL mappings

- Adesso dobbiamo aggiungere gli URL mapping che collegano:
  - `catalog/books/` alla sua class-based view
  - `catalog/authors/` alla sua class-based view

# URL mappings

- Va modificato `catalog/urls.py` estendendo `urlpatterns`:

```
from django.urls import path
from . import views

urlpatterns = [
path('' , views.index , name='index' ),
path('books/' , views.BookListView.as_view() , name='books' ),
path('authors/', views.AuthorListView.as_view(), name='authors')]
```

# Test!

- Siamo finalmente in grado di testare il sito con 3 pagine!

- Routine di esecuzione server:

  `python3 manage.py makemigrations`

  `python3 manage.py migrate`

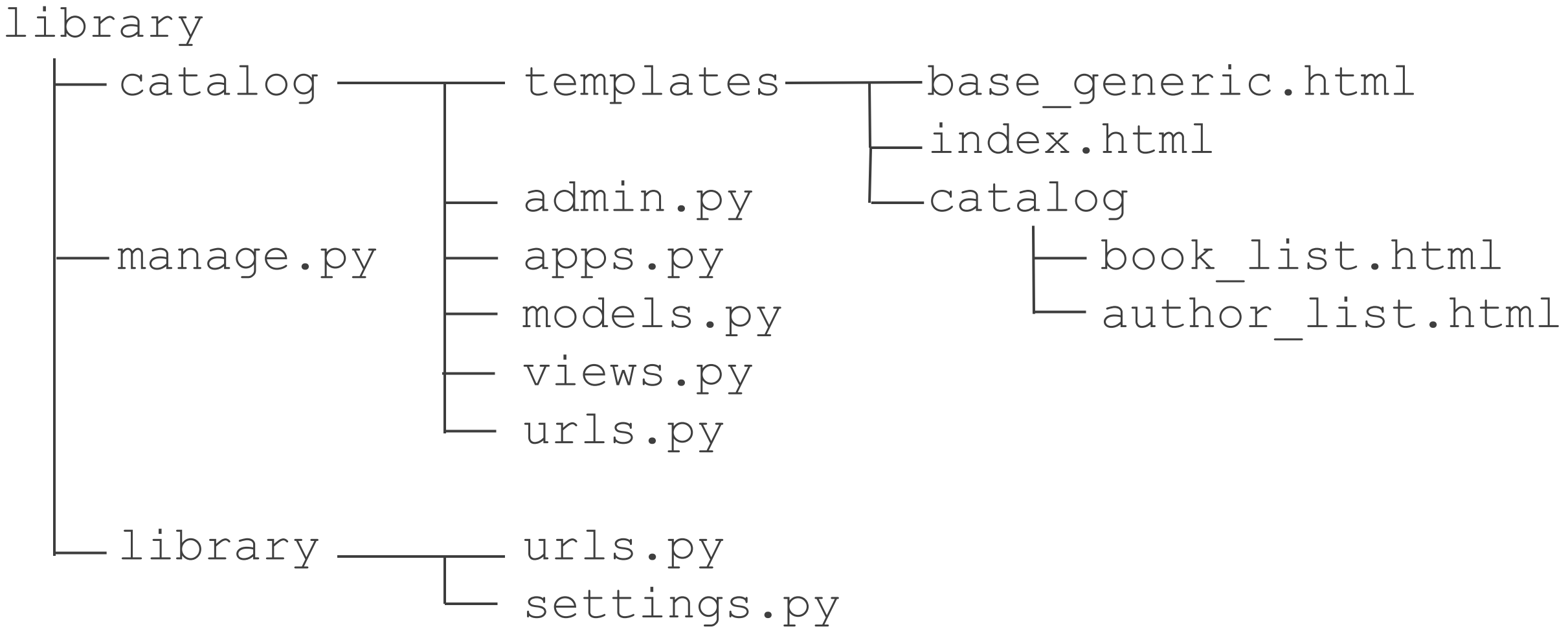
  `python3 manage.py runserver`

  Apri un browser

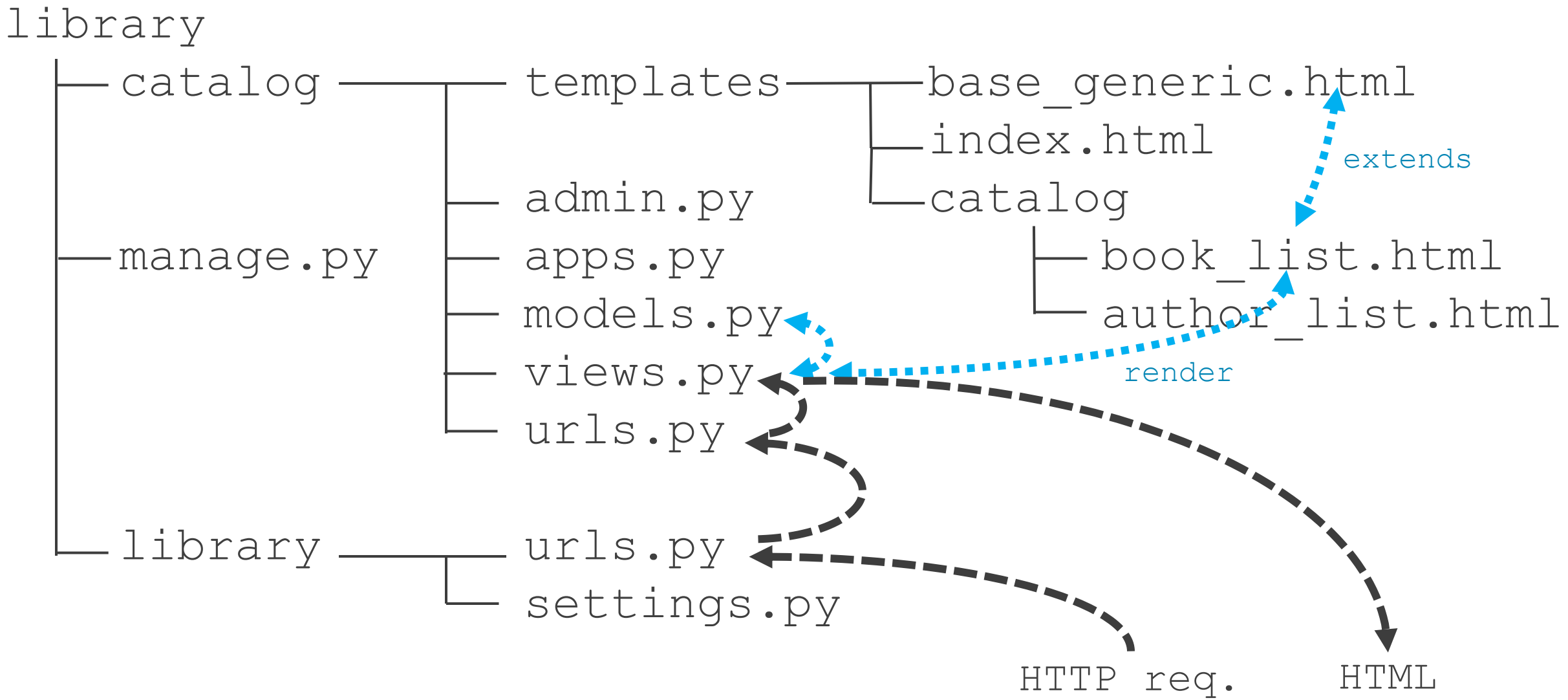
  Pulisci la cache

  Apri la pagina `http://127.0.0.1:8000`

# Recap: struttura files e lifecycle (book list)



# Recap: struttura files e lifecycle (book list)



# Note su CBV vs. FBV

- Le CBV includono molti attributi e metodi definiti per default
- Essi sono tuttava ridefinibili da utente
- *Talvolta puo' essere utile o necessario farlo*, ad esempio se:
  - Si hanno più CBV per la stessa classe (afferenti a templates diverse)
  - Non si vuole che una CBV tratti tutti i dati della classe, ma solo una parte legata ad un criterio
  - Il template necessita di dati ulteriori oltre alla lista di oggetti

# Note su CBV vs. FBV

- Primo esempio: CBV per la lista dei libri scritti dal sig. Rossi
- La lista di oggetti è definita dal metodo `get_queryset`
- Il nome del template è definite dall'attributo `template_name`

```
class BookListView(generic.ListView):
    model = Book
    template_name = 'catalog/books_mr_Rossi.html'

    def get_queryset(self):
        return Book.objects.filter(author.last_name = 'Rossi')
```

# Note su CBV vs. FBV

- Esempio esteso: la CBV passa al suo template anche la data odierna
- Si ridefinisce il contesto costruito dalla CBV attraverso il metodo `get_context_data`

```
class BookListView(generic.ListView):
    model = Book
    template_name = 'catalog/books_mr_Rossi.html'

    def get_queryset(self):
        return Book.objects.filter(author.last_name = 'Rossi')

    def get_context_data(self, **kwargs):
        ctx = super(BookListView,self).get_context_data(**kwargs) # Default…
        ctx['today'] = str(date.today())   # va importato date da datetime…
        return ctx
```

# Note su CBV vs. FBV

- Di fatto una CBV, per default, fa cose che sarebbe facile simulare anche con una FBV, costruendo il contesto e renderizzando il template:

```
def fbv_BookListView(request):
    the_books = Book.objects.all()
    ctx = { 'book_list' : the_books }
    return render(request, 'catalog/book_list.html',context = ctx)
```

- Tuttavia, come vedremo, le CBV danno supporto agile anche a caratteristiche che sarebbe complesso realizzare attraverso FBV.

# Versione II: files statici e stile

- Motivazione: migliorare grafica e svincolarsi da `bootstrap`
- Le stylesheets CSS (ed i files JS e le immagini) sono files *statici,* ovvero fissi e presenti nel server, da caricare localmente
- Dunque:
  - Abilitare la gestione dei files statici (in ambiente di sviluppo)
  - Introdurre un file statico di stile
- Rif.: https://docs.djangoproject.com/it/5.1/**howto/static-files/**

# Versione II: files statici e stile

- Implicazioni e passi:
    1. Estensione del mapping URL **di progetto**
    2. Estensione dei template di base
    3. Creazione del file (o dei files) di stile

# Versione II: files statici e stile

Step 1:

- Verificare che in `settings.py STATIC_URL` sia settato a `/static`
- Verificare che in `settings.py STATIC_ROOT` non sia settato
- Estendere `urls.py` di progetto per gestire files statici includendo:
  - Moduli necessari
  - Pattern ad-hoc

```
from django.conf import settings
from django.conf.urls.static import static
urlpatterns+= static(settings.STATIC_URL,
                     document_root=settings.STATIC_ROOT)
```

# Versione II: files statici e stile

- Step 2: Modifica del template di base (parte head):
  - Includere direttiva DTN per abilitare files statici
  - Includere link (con direttiva DTN) verso files statici

```
{% load static %}
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
<link rel="stylesheet" href="{% static 'css/home_layout.css' %}">
```

  - Rimuovere inclusione (ora inutile) di bootstrap

# Versione II: files statici e stile

- Passo 3:
  - Creazione ex-novo della cartella `.../catalog/static/css`
  - Definizione del file `styles.css`:

```
.sidebar-nav { margin-top: 20px;
               padding: 0;
               list-style: none; }

body         { font-family: "Lato", sans-serif;
               font-size  : 1.3em;
               height : 96vh;
               margin-top: 2vh;
               margin-bottom: 2vh; }
```

# Versione II: files statici e stile

- ...definizione (sotto `/static/css`) del file di "simulazione" di `bootstrap`, `home_layout.css`:

```
.container-fluid { width: 100%;
                   height: 100%; }
.row             { display: block;
                   height: 100%; }
.col-sm-2        { display: inline-block;
                   vertical-align: top;
                   width: 20%;
                   height: 100%;
                   background-color: silver; }
```

# Versione II: files statici e stile

- ...ct'd :

```
.col-sm-10          { display: inline-block;
                      vertical-align: top;
                      width: 80%;
                      height: 100%;
                      background-color: linen;
                      overflow: auto; }
```

- *NB ora, attenzione agli spazi fra i div di classe `col-sm-*`!*

# Versione II: files statici e stile

- Step ulteriore: sistemiamo il problema con `favicon.ico`
- Si tratta della iconcina di default del sito
- Anche essa va posta nei dati statici
- Scarichiamo una icona in formato `.png` da 64x64 pixel ad es. da https://www.flaticon.com/uicons/interface-icons
- La poniamo, rinominata, in `static/images/favicon.png`
- Aggiungiamo, alla `head` di `base_generic.html`:

```
<link rel="shortcut icon" type="image/png"
      href="{% static 'images/favicon.png' %}" >
```

# Test!

```
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000`

NB: i files statici normalmente vanno "rinfrescati" con shift-F5

# Versione III: paginazione

- Le pagine delle liste libri e autori hanno lunghezze arbitrarie

- Vanno paginate:
  - Settato un max di istanze per pagina
  - Va introdotto un meccanismo di navigazione delle pagine

- Django offre supporto per questa funzionalita':
  - A livello di dichiarazione delle CBVs per "liste di oggetti"
  - A livello di definizione del template

# Versione III: paginazione

- A livello delle CBVs, e' sufficiente definire l'attributo `paginate_by`:

```
class BookListView(generic.ListView):
    model = Book
    paginate_by = 3


class AuthorListView(generic.ListView):
    model = Author
    paginate_by = 4
```

# Versione III: paginazione

- Per gestire la paginazione nel template, la CBVs gli passa:
  - `is_paginated` : indica se la pagina e' paginata
  - `page_obj` : oggetto di paginazione con attributi, fra cui
    - `has_previous` / `has_next` : se esista pagina prec./succ.
    - `number` : numero della pagina attuale
    - `num_pages` : numero delle pagine totali
    - `previous_page_number`, `next_page_number`
- Le pagine "paginate" hanno URL `{{request.path}}?page=N`

# Versione III: paginazione

- Ora si modifica il template di base con un blocco "di paginazione" posto a fine pagina (ovvero: dopo il  blocco dei contenuti)
- Cosi' facendo si paginano entrambe le pagine con liste
- Logica del blocco:
  - Se la pagina e' paginata:
    - Scrivi che pagina e'
    - A seconda che abbia una pagina precedente e/o successiva, inserisci i links corrispondenti

# Versione III: paginazione

```
{% block pagination %}
  {% if is_paginated %}
    <div class="pagination">
      {% if page_obj.has_previous %}
        <a href="{{ request.path }}?page=
                 {{ page_obj.previous_page_number }}">previous</a>
      {% endif %}
      Page {{ page_obj.number }} of {{ page_obj.paginator.num_pages }}
      {% if page_obj.has_next %}
        <a href="{{ request.path }}?page=
                 {{ page_obj.next_page_number }}">next</a>
      {% endif %}
    </div>
  {% endif %}
{% endblock %}
```

# Test!

- Prima, da superuser, immettiamo abbastanza istanze
- Poi:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

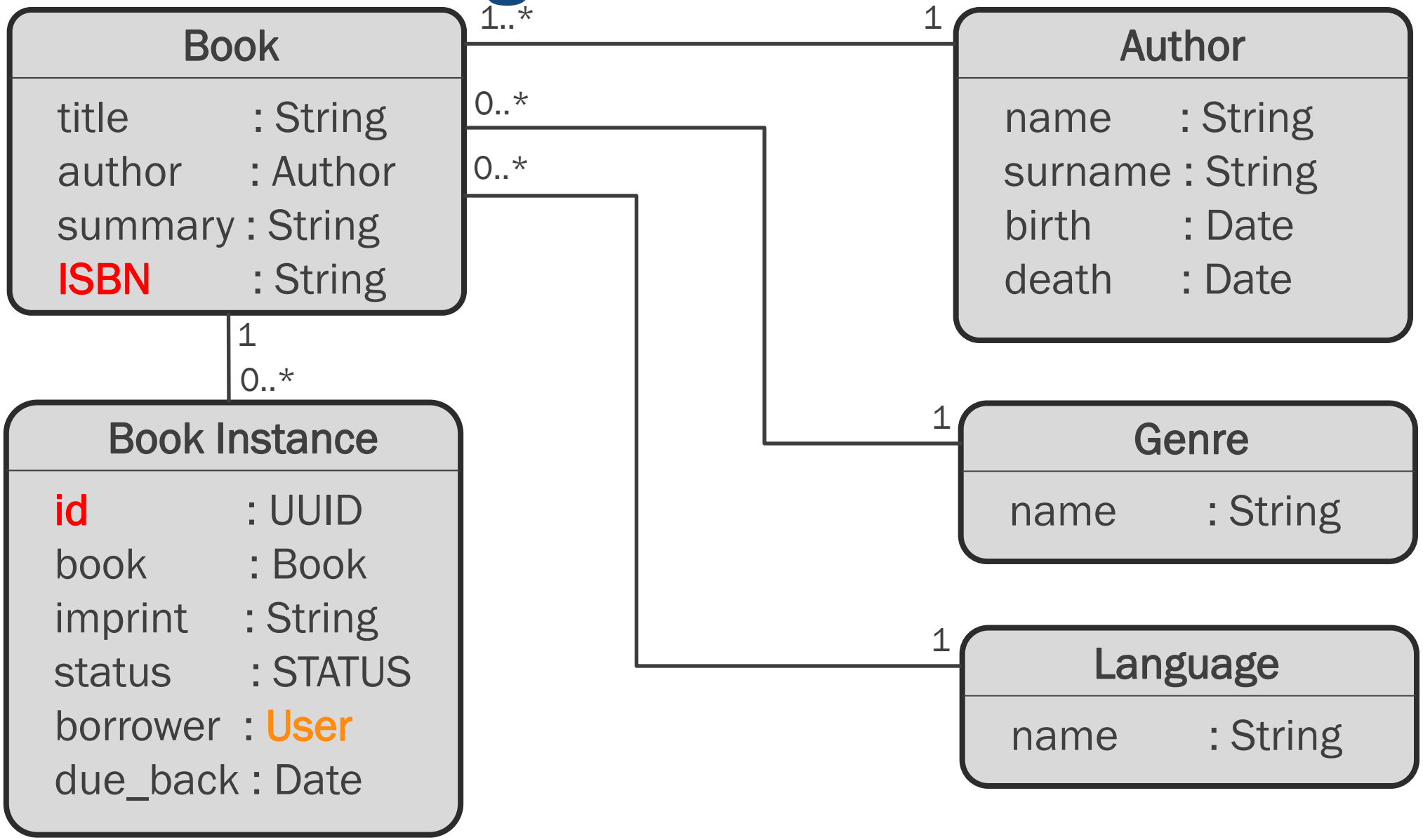
Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- **A questo punto gia' vedremo gli effetti della paginazione**

# Versione IV: completamento modello dati

- Passo propedeutico a fare cose piu' interessanti
- Introdurremo: genere, lingua e istanze di libro
  - Le istanze di libro hanno struttura, sono legate uno-a-molti ai libri e caratterizzate da uno stato (in prestito, disponibile, ...)
  - Genere e lingua sono semplici dati legati uno-a-molti ai libri
- Concettualmente, nulla di nuovo
- Tuttavia, introduce nuovi tipi nel modello dati

# Design del modello dati



# Versione IV: completamento modello dati

- Modelli `Genre` e `Language`: semplici, immessi a inizio di `models.py`

```
class Genre(models.Model):
  name = models.CharField(max_length=200,
                          help_text="Enter a genre" )

  class Meta:
        ordering = ['name']

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

# Versione IV: completamento modello dati

```
class Language(models.Model):
    name = models.CharField(max_length=200,
                            help_text="Enter language")

    class Meta:
        ordering = ['name']

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

# Versione IV: completamento modello dati

- Aggiorniamo il modello di `Book` con i campi `genre` e `language`:

```
genre    = models.ForeignKey(
                  Genre,
                  on_delete = models.SET_NULL,
                  null      = True)
language = models.ForeignKey(
                  Language,
                  on_delete = models.SET_NULL,
                  null      = True)
```

# Versione IV: completamento modello dati

- Modello per istanze di libri: immesso a fine di `models.py`
- Vuole rappresentare una istanza di libro:
  - Con un suo ID univoco
  - Con un suo imprint (etichetta descrittiva)
  - Con un suo stato: disponibile, in prestito, in revisione
  - Se in prestito: utente e data di restituzione prevista

# Versione IV: completamento modello dati

- Serviranno:
  - Il tipo `UUID`, che serve a far generare ID univoci da Django, che useremo come chiave primaria delle istanze di libro
  - Il tipo `User`, che permette di riferirsi agli utenti del sito, tabella gestita automaticamente da Django
  - Gestire un campo di stato con valori prefissati "di scelta"

```
import uuid
from datetime import date
from django.contrib.auth.models import User

class BookInstance(models.Model):
    id = models.UUIDField(primary_key = True,
                          default     = uuid.uuid4,
                          help_text   = "Unique ID for book")
    book = models.ForeignKey(Book,
                             on_delete = models.RESTRICT,
                             null      = True)
    imprint  = models.CharField(max_length=200)
    due_back = models.DateField(null=True, blank=True)
    borrower = models.ForeignKey(User,
                                 on_delete = models.SET_NULL,
                                 null      = True,
                                 blank     = True)
```

```
LOAN_STATUS = ( ('d', 'Maintenance'),
                ('o', 'On loan'),
                ('a', 'Available'),
                ('r', 'Reserved'),)

    status = models.CharField(
                    max_length = 1,
                    choices    = LOAN_STATUS,
                    blank      = True,
                    default    = 'd',
                    help_text  = 'Book availability')

    class Meta:
          ordering = ['due_back', 'book']
    def __str__(self):
          return f'{self.id} ({self.book.title})'
```

# Versione IV: completamento modello dati

- A questo punto il modello di dati e' completo
- Va aggiornato `admin.py` per importare e registrare il modello:

```
from django.contrib import admin
from .models import Author, Book,
                    Genre, Language, BookInstance


admin.site.register(Book)
admin.site.register(Author)
admin.site.register(Genre)
admin.site.register(Language)
admin.site.register(BookInstance)
```

# Test!

- Giacche' abbiamo modificato il modello dati in `models.py`...

  ```
  python3 manage.py makemigrations
  python3 manage.py migrate
  python3 manage.py runserver
  ```

  Apri un browser
  Apri la pagina `http://127.0.0.1:8000`

- Ora come superusers, possiamo creare istanze di libri, di generi, etc. - **e dobbiamo farlo**, ai fini dello sviluppo del test.

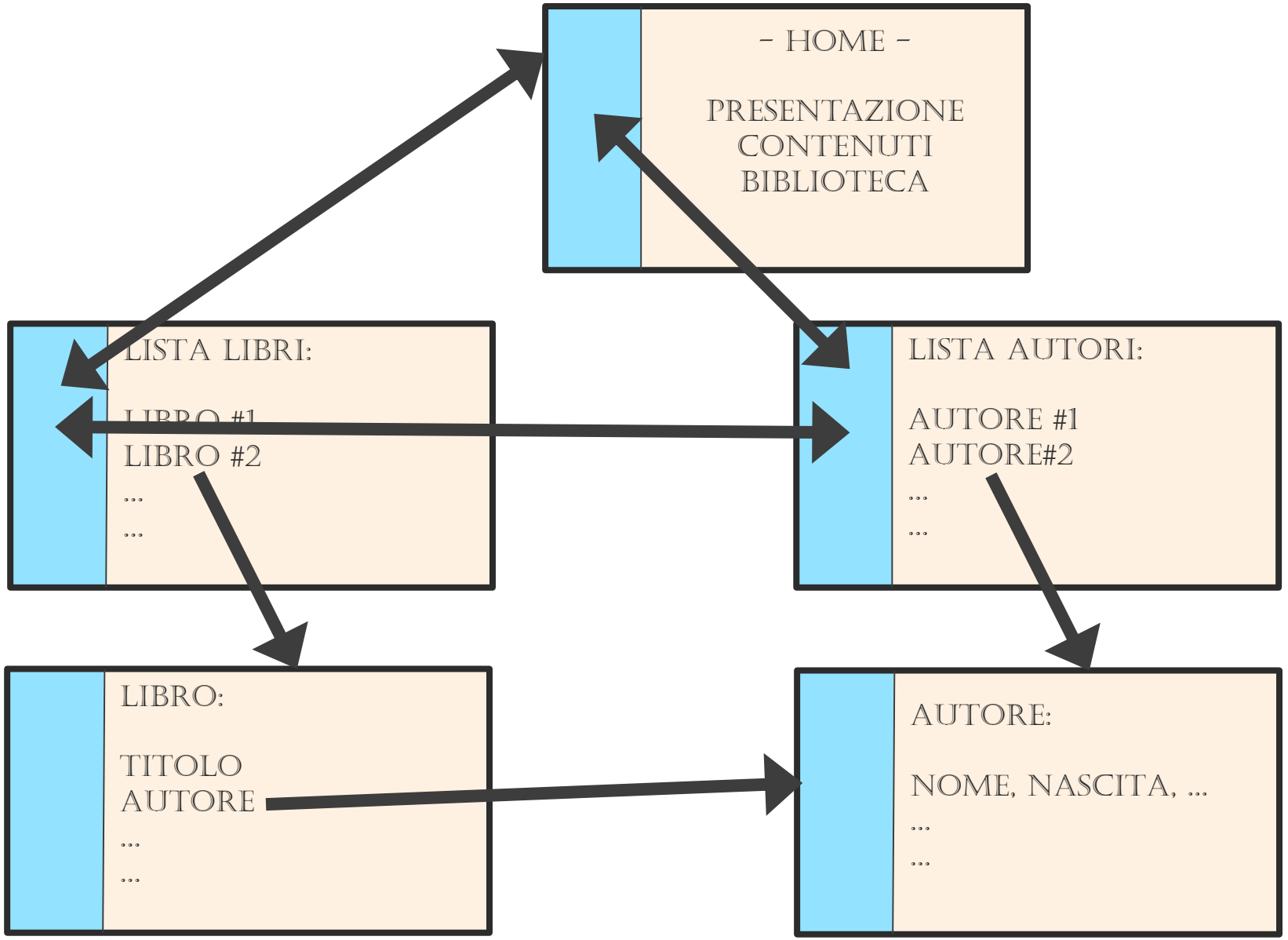
# Versione V: pagine di dettaglio (libri, autori)

- Obiettivo:
  - Dalla pagina "lista autori" aprire pagine sui singoli autori
  - Dalla pagina "lista libri" aprire pagine sui singoli libri (e istanze)

- Concettualmente:
  - Vanno introdotti degli URL mapping "legati a istanze"
  - Vanno definite relative views e templates
  - Vanno raffinati i templates e i modelli esistenti:

    (per linkare gli elementi delle "liste" alle pagine singole)

# Obiettivo


- HOME -
PRESENTAZIONE
CONTENUTI
BIBLIOTECA
LISTA LIBRI:
LIBRO #1
LIBRO #2
...
...
LISTA AUTORI:
AUTORE #1
AUTORE#2
...
...
LIBRO:
TITOLO
AUTORE
...
...
AUTORE:
NOME, NASCITA, ...
...
...

# Versione V: mappings e views

- Anche per elementi singoli, Django fornisce una classe `DetailView`

- Dunque e' ampiamente conveniente usare CBVs, in `views.py`:

```
class BookDetailView(generic.DetailView):
    model = Book

class AuthorDetailView(generic.DetailView):
    model = Author
```

# Versione V: mappings e views

- Le CBVs derivate da `DetailView` si aspettano un parametro `pk` (primary key) come id dell'elemento
- Invocano templates il cui nome e' costruito dalla classe: `book_detail.html` e `author_detail.html`
- Come ogni CBV, li situano in `templates/`**`app`**`/`
- Passano loro un argomento che contiene la istanza di dato: risp. `book` e `author`

# Versione V: mappings e views

- Nello `urls.py` di app vanno introdotte regole di mapping legati a ID:

```
urlpatterns = [ ... ,
    path('book/<int:pk>',
         views.BookDetailView.as_view(),
         name='book-detail'),
    path('author/<int:pk>',
         views.AuthorDetailView.as_view(),
         name='author-detail') ]
```

- **Nota:** Qui si vede l'uso di "`pk`"

# Versione V: template dettaglio autore

- Template per autore in `author_detail.html`
- Logica semplice: mostra i campi del dato `author` ricevuto dalla CBV

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Author: {{ author.first_name }}  {{ author.last_name }}</h1>
  <p><strong> Name:       </strong> {{ author.first_name    }}</p>
  <p><strong> Surname:    </strong> {{ author.last_name     }}</p>
  <p><strong> Birth date:</strong> {{ author.date_of_birth }}</p>
  <p><strong> Death date:</strong> {{ author.date_of_death }}</p>
{% endblock %}
```

# Versione V: template dettaglio libro

- Template per libro in `book_detail.html`:
  - Per ora logica semplice ed analoga a quella per l'autore

```
{% extends "base_generic.html" %}
{% block content %}
  <h1>Title: {{ book.title }}</h1>
  <p><strong> Author:   </strong> {{ book.author   }}</p>
  <p><strong> Summary:  </strong> {{ book.summary  }}</p>
  <p><strong> ISBN:     </strong> {{ book.isbn     }}</p>
  <p><strong> Language: </strong> {{ book.language }}</p>
  <p><strong> Genre:    </strong> {{ book.genre    }}</p>
{% endblock %}
```

# Versione V: template dettaglio libro

- Aggiorniamo i templates per le liste di libri e autori: gli elementi delle liste devono puntare alle pagine.
- Per fare questo servono due metodi che, per un autore o libro, restituiscono il loro URL. Li chiameremo `get_absolute_url`
- Per `author_list.html`:

```
{% for author in author_list %}
    <li>
        <a href="{{ author.get_absolute_url }}">
                {{ author.first_name }}
                {{ author.last_name }}
        </a>
    </li>
{% endfor %}
```

# Versione V: template dettaglio libro

- Aggiorniamo i templates per le liste di libri e autori: gli elementi delle liste devono puntare alle pagine.
- Per fare questo servono due metodi che, per un autore o libro, restituiscono il loro URL. Li chiameremo `get_absolute_url`
- Per `author_list.html`:

```
{% for author in author_list %}
    <li>
        <a href="{{ author.get_absolute_url }}">
                {{ author.first_name }}
                {{ author.last_name }}
        </a>
    </li>
{% endfor %}
```

Chiamata a metodo!

# Versione V: template dettaglio libro

- Similmente per `book_list.html`:

```
{% for book in book_list %}
    <li>
        <a href="{{ book.get_absolute_url }}">
                {{ book.title }}
        </a>
        ({{book.author}})
    </li>
 {% endfor %}
```

# Versione V: template dettaglio libro

- Introduciamo il metodo `get_absolute_url` in `Author` e in `Book`, usando la funzione `reverse` (che dovremo importare):

```
from django.urls import reverse
....

class Author(models.Model):
    ....
    def get_absolute_url(self):
        return reverse('author-detail', args=[str(self.id)])

class Book(models.Model):
    ....
    def get_absolute_url(self):
        return reverse('book-detail', args=[str(self.id)])
```

# Test!

- Ed ora...

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- **Ora possiamo vedere gli effetti: le pagine di dettaglio**

# Versione V: estensione pagina libro

- Estendiamo `book_detail.html` per
  - Linkare all'autore
  - Mostrare le copie (evidenziando stati diversi con stili CSS)

```
{% extends "base_generic.html" %}
{% block content %}
  <h1>Title: {{ book.title }}</h1>
  <p><strong>Author:</strong>
     <a href="{{ book.author.get_absolute_url }}">
                {{ book.author }}</a></p>
  <p><strong> Summary:  </strong> {{ book.summary  }}</p>
  <p><strong> ISBN:     </strong> {{ book.isbn     }}</p>
  <p><strong> Language: </strong> {{ book.language }}</p>
  <p><strong> Genre:    </strong> {{ book.genre    }}</p>
  ...........
```

# Versione V: estensione pagina libro

- Estendiamo `book_detail.html` per
  - Linkare all'autore
  - Mostrare le copie (evidenziando stati diversi con stili CSS)

```
{% extends "base_generic.html" %}
{% block content %}
  <h1>Title: {{ book.title }}</h1>
  <p><strong>Author:</strong>
     <a href="{{ book.author.get_absolute_url }}">
                {{ book.author }}</a></p>
  <p><strong> Summary:  </strong> {{ book.summary  }}</p>
  <p><strong> ISBN:     </strong> {{ book.isbn     }}</p>
  <p><strong> Language: </strong> {{ book.language }}</p>
  <p><strong> Genre:    </strong> {{ book.genre    }}</p>
  ...........
```

Chiamata a metodo!

```
<div style="margin-left:20px;margin-top:20px">
    {% if book.bookinstance_set.all %}
      <h4>Copies</h4>
      {% for copy in book.bookinstance_set.all %}
        <hr>
        <p class="{% if copy.status == 'a' %}text-success
                  {% else %}text-warning{% endif %}">
          {{ copy.get_status_display }}
        </p>
        {% if copy.status == 'o' %}
          <p><strong>Due back by </strong> {{ copy.due_back }}</p>
        {% endif %}
        <p><strong>Imprint:</strong> {{ copy.imprint }}</p>
        <p class="text-muted"><strong>Id:</strong> {{ copy.id }}</p>
      {% endfor %}
    {% else %}
      <h4 class="text-danger">This book has no copies</h4>
    {% endif %}
  </div>
{% endblock %}
```

Chiamata a metodo!

# Versione V: template dettaglio libro

- Note:
  - Set di istanze `bookinstance_set` (ev. vuoto) da `ForeignKey`
  - Lo stato del libro viene usato per switchare stile
  - Il nome "full" di un `choices` **C** si ottiene con `get_C_display`

# Versione V: template dettaglio libro

- Estendiamo `styles.css` per gestire le varie classi `text-*`:

```
.text-success { color: #28a745; }
.text-warning { color: #ffc107; }
.text-danger  { color: #dc3545; }
.text-muted   { color: #6C757D; }
```

# Test!

- Di nuovo...

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- **Ora possiamo vedere i libri linkati alle copie ed agli autori**

# Versione VI: pagina admin personalizzata

- Obiettivo: strutturare ed arricchire la pagina di admin:
  - Migliorando le viste a lista come aspetto e contenuti
  - Migliorando la strutturazione delle viste dei singoli elementi
  - Permettendo di gestire elementi collegati (libro e istanze)
  - Abilitando filtri di selezione
  - Rif.: https://docs.djangoproject.com/en/5.1/**ref/contrib/admin/**

# Versione VI: pagina admin personalizzata

- Concettualmente:
  - Si modifica la sola parte di registrazione modelli in `admin.py`
  - Ci si appoggia ad una classe di supporto
  - La classe fornisce proprieta' e metodi che modificheremo

# Versione VI: pagina admin personalizzata

- Step 1: definire, prima delle `register`, per ognuno dei modelli da personalizzare, una classe derivata da `ModelAdmin` (per ora vuota)

```
class AuthorAdmin(admin.ModelAdmin):
    pass

class BookAdmin(admin.ModelAdmin):
    pass

class BookInstanceAdmin(admin.ModelAdmin):
    pass
```

# Versione VI: pagina admin personalizzata

- Passo 2: usare quelle classi nella registrazione:

```
admin.site.register(Genre)
admin.site.register(Language)
admin.site.register(Author, AuthorAdmin)
admin.site.register(Book, BookAdmin)
admin.site.register(BookInstance, BookInstanceAdmin)
```

(Fin qui, se testiamo il sito, nulla e' cambiato)

# Versione VI: pagina admin personalizzata

- Passo 3: definire le proprieta' rilevanti nelle classi di tipo `Admin`:
  - Primo esempio: definire i campi mostrati nelle viste a lista
  - Lo si fa definendo `list_display`:

```
class AuthorAdmin(admin.ModelAdmin):
    list_display = ('last_name', 'first_name',
                    'date_of_birth', 'date_of_death')

class BookAdmin(admin.ModelAdmin):
    list_display = ('title', 'author', 'genre')

class BookInstanceAdmin(admin.ModelAdmin):
    list_display = ('book', 'status', 'borrower', 'due_back', 'id')
```

# Test!

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000/admin`

- **Ora possiamo vedere gli effetti nella pagina Admin**

# Versione VI: pagina admin personalizzata

- Secondo esempio: definire i campi mostrati nella vista singola, con loro divisione orizzontale/verticale (notazione parentetica)
- Lo si fa definendo `fields`:

```
class AuthorAdmin(admin.ModelAdmin):
    [...]

    fields = ['first_name',
              'last_name',
               ('date_of_birth', 'date_of_death')]
```

# Test!

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000/admin`

- **Ora possiamo vedere gli effetti nella pagina Admin**

# Versione VI: pagina admin personalizzata

- Terzo esempio: divisione dei campi in sottogruppi (con eventuali etichette) nella vista singola, definendo `fieldsets`:

```
class BookInstanceAdmin(admin.ModelAdmin):
   [...]
   fieldsets    = (
                    (None, {
                       'fields': ('book', 'imprint', 'id')
                     }),
                    ('Availability', {
                       'fields': ('status', 'due_back', 'borrower')
                     }),
                  )
```

# Test!

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000/admin`

- **Ora possiamo vedere gli effetti nella pagina Admin**

# Versione VI: pagina admin personalizzata

- Quarto esempio: visualizzazione di un elemento contestualmente ad altri ad esso "collegati" (es. i libri di un autore contestualmente all'autore), attraverso i cosiddetti `inlines`

- Per collegare un modello **`A`** dentro ad un modello **`B`**:
  - Va definita una classe **`inl`** basata su uno dei modelli di inline supportato, ad es. `TabularInLine`, e va settato `model = A`
  - Nel modello **`B`**, la proprieta' `inlines` va istanziata a **`inl`**

# Versione VI: pagina admin personalizzata

- Esempio: contestualmente agli autori, visualizziamo "inline" i loro libri...:

```
class BooksInline(admin.TabularInline):
    extra = 0
    model = Book

class AuthorAdmin(admin.ModelAdmin):
    [...]
    inlines = [BooksInline]
```

# Versione VI: pagina admin personalizzata

- Esempio: contestualmente agli autori, visualizziamo "inline" i loro libri...:

```
class BooksInline(admin.TabularInline):
    extra = 0
    model = Book

class AuthorAdmin(admin.ModelAdmin):
    [...]
    inlines = [BooksInline]
```

Non proporre moduli extra

# Versione VI: pagina admin personalizzata

- ...E contestualmente ai libri, visualizziamo "inline" le loro copie:

```
class BooksInstanceInline(admin.TabularInline):
    extra = 0
    model = BookInstance

class BookAdmin(admin.ModelAdmin):
    [...]
    inlines = [BooksInstanceInline]
```

# Test!

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000/admin`

- **Ora possiamo vedere gli effetti nella pagina Admin**

# Versione VII: gestione sessioni

- Obiettivo:
  - Riconoscere e memorizzare gli accessi da un client specifico
- Concettualmente:
  - Django supporta di default un meccanismo di sessione
  - Lo si utilizza nelle views e nei templates associati
  - Rif.: https://docs.djangoproject.com/en/5.1/**topics/http/sessions/**

# Versione VII: gestione sessioni

- Diango implementa un meccanismo built-in di sessione via cookies
- Possono venir modificati molti parametri (di memorizzazione e aggiornamento) ma vanno bene i default
- Il cookie `session` e' parte della request che viene passata alla view
- Il cookie `session` contiene:
  - un dizionario di nomi-valori liberamente gestibile
  - un campo built-in `modified` , per forzare Django a fare update

# Versione VII: gestione sessioni

- Esempio di accesso e modifica di cookie nella view della home:

```
def index(request):
    num_books   = Book.objects.all().count()
    num_authors = Author.objects.all().count()
    num_visits  = request.session.get('num_visits', 0)
    num_visits  = num_visits + 1
    request.session['num_visits'] = num_visits
    request.session.modified      = True
    ctx = {
            'num_books': num_books,
            'num_authors': num_authors,
            'num_visits': num_visits,
        }
    return render(request, 'index.html', context=ctx)
```

# Versione VII: gestione sessioni

- Corrispondente utilizzo nel template associato (`index.html`):

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Local Library Home</h1>
  <p>Welcome to Local Library!</p>
  <h2>Currently, this is what the library holds:</h2>
  <ul>
      <li><strong>Books:</strong>   {{ num_books   }}</li>
      <li><strong>Authors:</strong> {{ num_authors }}</li>
  </ul>
  <p>Your visits to this page: {{ num_visits }}</p>
{% endblock %}
```

# Test!

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000`

- Ora possiamo verificare gli effetti con viste ripetute alla home
- Per rimuovere il cookie in Chrome: seleziona la (i) vicino all'indirizzo del sito ➡ Cookies... ➡ Manage ➡ Cancella

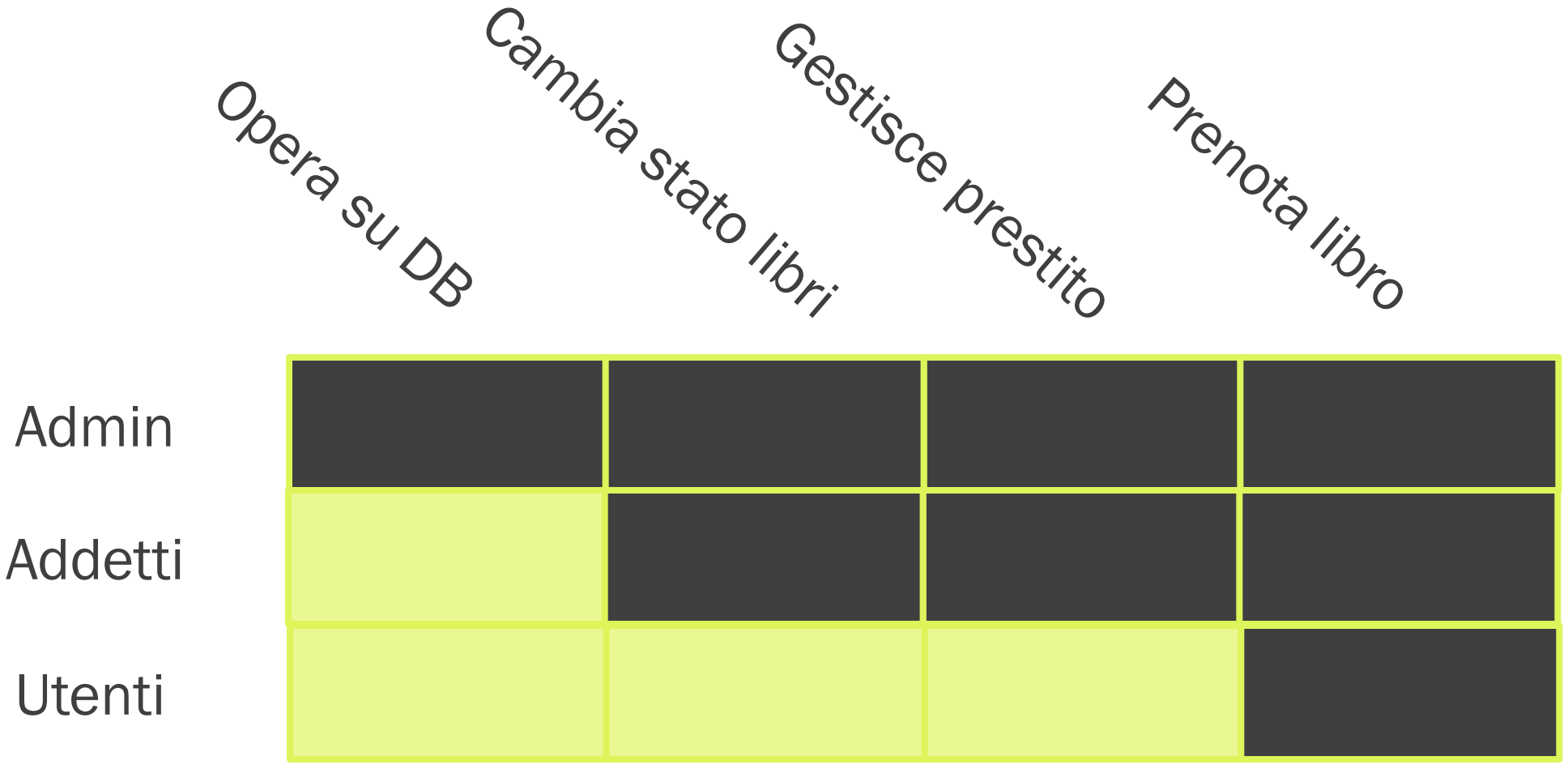
# Versione VIII: gestione autenticazioni

- Obiettivo:
  - Distinguere e gestire utenti autenticati e non
  - Limitare aree di accesso al sito a utenti non autenticati

- Django supporta un ricco sistema di autenticazione basato su:
  - Gruppi di utenti
  - Permessi associati ai diversi gruppi
  - Regole di accesso basate su appartenenza a gruppi e permessi

- Rif. https://docs.djangoproject.com/en/5.1/**topics/auth/**

# Versione VIII: gestione autenticazioni



# Versione VIII: gestione autenticazioni

- Concettualmente:
  - Vanno definiti utenti (e gruppi di utenti)
  - Vanno definite le pagine di autenticazione (login, logout, ...)
  - Vanno definite le regole di accesso alle views
  - Vanno utilizzati i dati su autenticazione dentro i templates

# Versione VIII: gestione autenticazioni

- Passo 1: definizione utenti
  - Si fa come superuser attraverso il sito admin
  - Conviene partire definendo i gruppi di utenti
  - Per ora, agli utenti ed ai gruppi non associamo permessi
  - Iniziamo creando due gruppi `Utenti` ed `Addetti`
  - Creiamo almeno 2-3 utenti per ognuno dei due gruppi
  - Nota: gli utenti vanno prima creati e poi associati ai gruppi

# Versione VIII: gestione autenticazioni

- Passo 2: settaggio della infrastruttura (`settings.py`):
  - Va gestita mail (via console)
  - Va definita una pagina di fallback post autenticazione

```
# Redirect to home URL after login
LOGIN_REDIRECT_URL = '/'

# Redirect Django-generated emails to console
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

# Versione VIII: gestione autenticazioni

- Si modifica `urls.py` **di progetto** includendo infrastruttura Django:

```
urlpatterns +=
   [ path('accounts/', include('django.contrib.auth.urls')), ]
```

- La infrastruttura implicitamente genera 8 URL **mappings e views** :

```
accounts/login/                    [name='login']
accounts/logout/                   [name='logout']
accounts/password_change/          [name='password_change']
accounts/password_change/done/     [name='password_change_done']
accounts/password_reset/           [name='password_reset']
accounts/password_reset/done/      [name='password_reset_done']
accounts/reset/<uidb64>/<token>/   [name='password_reset_confirm']
accounts/reset/done/               [name='password_reset_complete']
```

# Versione VIII: gestione autenticazioni

- ...Non vengono generati pero' i corrispondenti **templates**!
- Vanno creati, nella cartella `templates/registration`

- Passo 3 (teorico): generazione degli **8 templates**
  - E' codice standard ("boilerplate") ma complesso
  - Lo diamo per dato, prendendo implementazioni standard
  - Utilizzare tali implementazioni, rese disponibili nei materiali

# Versione VIII: gestione autenticazioni

- Passo 4: utilizzo di autenticazione in template
  - Lo applichiamo al menu laterale in `base_generic.html`
  - Si usano:
    - `user.is_authenticated` : indica se l'utente e' autenticato
    - `user.get_username` : nome dell'utente
    - `next` : pagina successiva a operazione di autenticazione
    - `request.path` : URL della pagina attuale

```
<ul class="sidebar-nav">
    ...
    {% if user.is_authenticated %}
      <li>User: {{ user.get_username }}
          <form style="display: inline"
                method="post"
                action="{% url 'logout' %}?next={{ request.path }}">
                {% csrf_token %}
                <button style = "border: none; background-color:
                        transparent; font-family: 'Lato', sans-serif;
                        font-size:1em; text-decoration: underline;"
                        type="submit">(Logout)</button>
           </form>
      </li>
    {% else %}
        <li>No user:
          <a href="{% url 'login' %}?next={{ request.path }}">
              Login</a>
        </li>
    {% endif %}
</ul>
```

```
<ul class="sidebar-nav">
    ...
    {% if user.is_authenticated %}
      <li>User: {{ user.get_username }}
          <form style="display: inline"
                method="post"
                action="{% url 'logout' %}?next={{ re
                {% csrf_token %}
                <button style = "border: none; background-color:
                        transparent; font-family: 'Lato', sans-serif;
                        font-size:1em; text-decoration: underline;"
                        type="submit">(Logout)</button>
           </form>
      </li>
    {% else %}
        <li>No user:
          <a href="{% url 'login' %}?next={{ request.path }}">
              Login</a>
        </li>
    {% endif %}
</ul>
```

Per ragioni di sicurezza:
richiesto HTTP POST
(a partire da Django 5.0)

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Già ora possiamo vedere login e logout
- Ma tutte le pagine sono ancora accessibili sempre....

# Versione VIII: gestione autenticazioni

- Passo 5: restrizione dell'accesso a views a soli user autenticati:
  - Per CBVs:
    - Supportato dalla classe `LoginRequiredMixin`
    - Va dichiarata come classe padre della view
    - Va settato `login_url` : pagina di fallback
  - Per FVBs:
    - Meccanismo di decorazione `@login_required`
    - Analogamente, va settato (ma nei settings) `LOGIN_URL`

# Versione VIII: gestione autenticazioni

```
from django.contrib.auth.mixins import LoginRequiredMixin

class BookListView(LoginRequiredMixin, generic.ListView):
    login_url = '/accounts/login/'
    model = Book
    paginate_by = 3

class AuthorListView(LoginRequiredMixin, generic.ListView):
    login_url = '/accounts/login/'
    model = Author
    paginate_by = 4
```

# Versione VIII: gestione autenticazioni

```
class BookDetailView(LoginRequiredMixin, generic.DetailView):
      login_url = '/accounts/login/'
      model = Book

class AuthorDetailView(LoginRequiredMixin, generic.DetailView):
      login_url = '/accounts/login/'
      model = Author
```

# Versione VIII: gestione autenticazioni

- Nota: Django supporta ulteriori forme di restrizione all'accesso:
  - Esempio 1: restrizione basata su definizione di permessi
    - Per CBVs:
      - Supportato dalla classe `PermissionRequiredMixin`
      - Va dichiarata come classe padre della view
      - Va settato `permission_required` : il permesso necessario
    - Per FVBs:
      - Meccanismo di decorazione `@permission_required`
      - Analogamente, va settato il permesso come parametro

# Versione VIII: gestione autenticazioni

- Esempio 2: restrizione specifica a utenze "di staff amministrativo"
  - Per CBVs:
    - Si puo' usare la classe (generica) `UserPassesTestMixin`
    - Va dichiarata come classe padre della view
    - Va definito il metodo booleano `test_func`, che in questo caso puo' basarsi sul contenuto di `user.is_staff`
  - Per FVBs:
    - Meccanismo di decorazione `@staff_member_required`
    - Va importato da `django.contrib.admin.views.decorators`

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- **Ora possiamo vedere gli effetti, agendo da connessi o no**

# Versione VIII: prettyfication

- In questo momento, la home presenta le liste libri/autori a tutti
- Tuttavia accederle senza essere autenticati porta solo alla login
- Ha piu' senso allora presentarle o meno a seconda dello stato di autenticazione...
- E' sufficiente modificare la logica in `base_generic.html`

```
{% block sidebar %}
<ul class="sidebar-nav">
    <li><a href="{% url 'index' %}">Home</a></li>
    <li><a href="{% url 'books' %}">All books</a></li>
    <li><a href="{% url 'authors' %}">All authors</a></li>
    {% if user.is_authenticated %}
        <li><a href="{% url 'books' %}">All books</a></li>
        <li><a href="{% url 'authors' %}">All authors</a></li>
         <li>User: {{ user.get_username }}
             .....
      .....
{% endblock %}
```

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- **Ora possiamo vedere gli effetti, agendo da connessi o no**

# Versione VIII: prettyfication (ct'd)

- La logica "infrastrutturale" di logout resetta la sessione : bene
- Ma quella del login no. Volendo azzerare il contatore al login?

- Idea: passare per una pagina intermedia che resetta il contatore nella session, e "passa la palla" alla infrastruttura
- Gli passiamo la pagina di uscita a fine login, cosi' che la passi alla infrastruttura, come essa si aspetta.

# Versione VIII: prettyfication (ct'd)

- Introduciamo la FBV `resetlogin` in `views.html`
- Serve usare sia il metodo `reverse` nella ridirezione http, per ricostruire l'indirizzo della pagina di login e passarle la palla
- Notare come passa il parametro `next` alla infrastruttura di Django

```
from django.http import HttpResponseRedirect
from django.urls import reverse

def resetlogin(request, next):
    request.session['num_visits'] = 0
    request.session.modified      = True
    return HttpResponseRedirect(reverse('login') + "?next=" + next)
```

# Versione VIII: prettyfication (ct'd)

- Introduciamo la regola necessaria nello `urls.py` di app:

```
urlpatterns = [ ... ,
    path('resetlogin/<path:next>',
         views.resetlogin,
         name='resetlogin') ]
```

- **Nota:** Quel che passiamo e' un path: non una semplice stringa

# Versione VIII: prettyfication (ct'd)

- Modifica a `base_generic.html`

```
{% block sidebar %}
    ....
    ....
    {% else %}
        <li>No user:
          <a href="{% url 'resetlogin' request.path %}">
              Login</a>
          <a href="{% url 'login' %}?next={{ request.path }}">
              Login</a>
        </li>
    {% endif %}
</ul>
{% endblock %}
```

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- **Ora possiamo vedere il reset del contatore di visite in login**

# Versione IX: gestione permessi

- Obiettivo:
  - Distinguere e gestire diverse classi di utenti
  - Limitare funzionalita' e aree di accesso a determinate classi

  - Django include la gestione dei permessi come parte del sistema di autenticazione:

https://docs.djangoproject.com/en/5.1/topics/auth/default/#permissions-and-authorization

# Versione IX: gestione permessi

- Concettualmente:
  1. Vanno definiti i permessi cui riferirsi
  2. Vanno associati i permessi agli utenti (e gruppi di utenti)
  3. Vanno definite le regole di accesso con permesso alle views
  4. Vanno utilizzati i dati sui permessi dentro i templates

# Versione IX: gestione permessi

- Esempio concreto:
  - Creare una pagina che riporta "tutti i libri in prestito"
  - La pagina deve essere accedibile ai soli `Addetti`
  - La pagina non deve essere accessibile agli `Utenti`
  - *La pagina deve evidenziare i libri in prestito "in ritardo"*

# Versione IX: gestione permessi

- Passi concreti:
  1. Definire un permesso `PB` "da addetti"
  2. Associarlo al gruppo degli addetti
  3. Creare istanze di libro in prestito, sia dentro che fuori i termini
  4. Definire la view per la pagina "libri in prestito", vincolata a `PB`
  5. Definire il template associato
  6. Define lo URL-mapping associato
  7. Estendere il template "di base" per linkare la pagina creata
  8. Gestire il caso di mancato permesso a pagina

# Versione IX: gestione permessi

- Passo 1: definizione del permesso nel modello:
  - Si tratta di una caratteristica di `BookInstance`
  - Va definita nell'attributo `permissions` dei `Meta`
  - Si definisce una coppia nome - stringa di visualizzazione

```
class BookInstance(models.Model):
    ....
    class Meta:
        ordering = ['due_back', 'book']
        permissions = (
          ("can_mark_returned", "Set book as returned"),)
    ....
```

# Versione IX: gestione permessi

- Passo 2: aggiornare il DB (e attivare il server)

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

- Passo 3: associazione del permesso agli `Addetti`:
  - Questo viene eseguito come superuser da pagina admin
  - Permesso: `catalog | book instance | Set book as returned`

- Passo 4: creare istanze libro in prestito scaduto e non:
  - Questo viene eseguito come superuser da pagina admin

# Versione IX: gestione permessi

- Passo 5: definizione della view per libri in prestito:
  - CBV basata su `ListView`
  - Condizionata a login : include `LoginRequiredMixin`
  - Condizionata a permesso: ...anche **`PermissionRequiredMixin`**
  - Definisce sia la pagina di fallback che `permission_required`
  - Definiamo anche `template_name` (per evitare collisioni)
  - Definire esplicitamente il metodo `get_queryset` che da' il dato:
    - Filtrando con `filter` le istanze rispetto al loro stato
    - Ordinandole con `order_by`

```
from .models import BookInstance
from django.contrib.auth.mixins import PermissionRequiredMixin

class AllLoanedBooksListView (LoginRequiredMixin,
                              PermissionRequiredMixin,
                              generic.ListView):
    login_url = '/accounts/login/'
    model = BookInstance
    permission_required = 'catalog.can_mark_returned'
    template_name = 'catalog/bookinstance_list_loaned.html'
    paginate_by = 3

    def get_queryset(self):
        all_l = BookInstance.objects.filter(status__exact='o')
        return all_l.order_by('due_back')
```

# Versione IX: gestione permessi

- Passo 6: aggiunta del path allo URL-mapping di applicazione...

```
urlpatterns =
   [
    ... ,
    path('loaned_books/',
         views.AllLoanedBooksListView.as_view(),
         name='all-loaned-books')
   ]
```

# Versione IX: gestione permessi

- ...e creazione del template `bookinstance_list_loaned.html` riferito dalla view:
  - Riceve la lista dei libri prestati
  - Simile al template della lista completa dei libri

```
{% extends "base_generic.html" %}
{% block content %}
    <h1>Borrowed books</h1>
    {% if bookinstance_list %}
      <ul>
        {% for b in bookinstance_list %}
          <li>
            <a href="{% url 'book-detail' b.book.pk %}">
               {{ b.book.title }}
            </a>
            ({{ b.due_back }})
          </li>
        {% endfor %}
      </ul>
    {% else %}
      <p>There are no books being borrowed.</p>
    {% endif %}
{% endblock %}
```

# Versione IX: gestione permessi

- Passo 7: modifica di `base_generic.html` per gestire gli accessi:
  - Si usa l'oggetto `perms` che modella i permessi attuali
  - In particolare: `perms.catalog.can_mark_returned`
  - Solo se l'utente e' connesso ed ha il permesso aggiungiamo, nella side-bar della pagina, il link alla pagina "libri in prestito"

```
....
{% block sidebar %}
 <ul class="sidebar-nav">
    <li><a href="{% url 'index' %}">Home</a></li>
    {% if user.is_authenticated %}
       <li><a href="{% url 'books' %}">All books</a></li>
       <li><a href="{% url 'authors' %}">All authors</a></li>
       <li>User: {{ user.get_username }}
           ....
       </li>
       {% if perms.catalog.can_mark_returned %}
          <li><a href="{% url 'all-loaned-books' %}">
              Loaned books</a></li>
       {% endif %}
 ....
```

# Versione IX: gestione permessi

- Passo 8: gestione di mancanza di permessi
  - Scenario:
    - Siamo dentro una pagina avendone il permesso
    - Cambiamo utenza e ne scegliamo una senza permesso
    - Esito: **`Errore 403 (Forbidden)`**.
  - Gestione di Django:
    - Messaggio di default
    - Oppure mettiamo il template `403.html` , sotto `templates`!
    - Scriviamo `403.html` come una "home" customizzata

# Versione IX: gestione permessi

- La pagina `403.html`:

```
{% extends "base_generic.html" %}
{% block content %}
    <h1>
        Hey!
        <span style="color: red;">
            (You had no permission for the previous page...)
        </span>
    </h1>
    <h2><a href="{% url 'index' %}">Back home</a></h2>
{% endblock %}
```

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Ora possiamo vedere gli effetti, agendo da connessi o no

# Versione IX: gestione permessi - prettyfied

- Mettiamo in evidenza i libri con prestito scaduto
  - In primis: definizione, nel modello, del metodo "prestito scaduto"
  - Si riferisce a una `BookInstance`
  - La definiamo basandoci sulla data attuale

```
class BookInstance(models.Model):
    ....

    def is_overdue(self):
        return bool(self.due_back and
                    (date.today() > self.due_back))

    ....
```

# Versione IX: gestione permessi

...uso nel template `book_detail.html` ...

```
{% for copy in book.bookinstance_set.all %}
   <hr>
   <p class="{% if copy.status == 'a' %}  text-success
             {% elif copy.is_overdue %} text-danger
             {% else %}                 text-warning
             {% endif %}">
         {{ copy.get_status_display }}
   </p>
   ...
{% endfor %}
```

# Versione IX: gestione permessi

...e nel template `bookinstance_list_loaned.html`:

```
....
{% for b in bookinstance_list %}
  <li {% if b.is_overdue %} class="text-danger"{% endif %}>
    <a href="{% url 'book-detail' b.book.pk %}">
      {{ b.book.title }}
    </a>
    ({{ b.due_back }})
  </li>
{% endfor %}
....
```

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Ora possiamo vedere gli effetti, agendo da connessi o no

# Versione X: i moduli (forms)

- Obiettivo:
  - Creare una pagina specifica di rinnovo prestito
  - Sara' accedibile solo agli addetti
  - Prevedera' la interazione dell'addetto con una form

- Django supporta la gestione delle forms sia come definizione dei loro dati, sia per quanto riguarda la parte di logica
- Riferimenti:

https://docs.djangoproject.com/en/5.1/**topics/forms/**

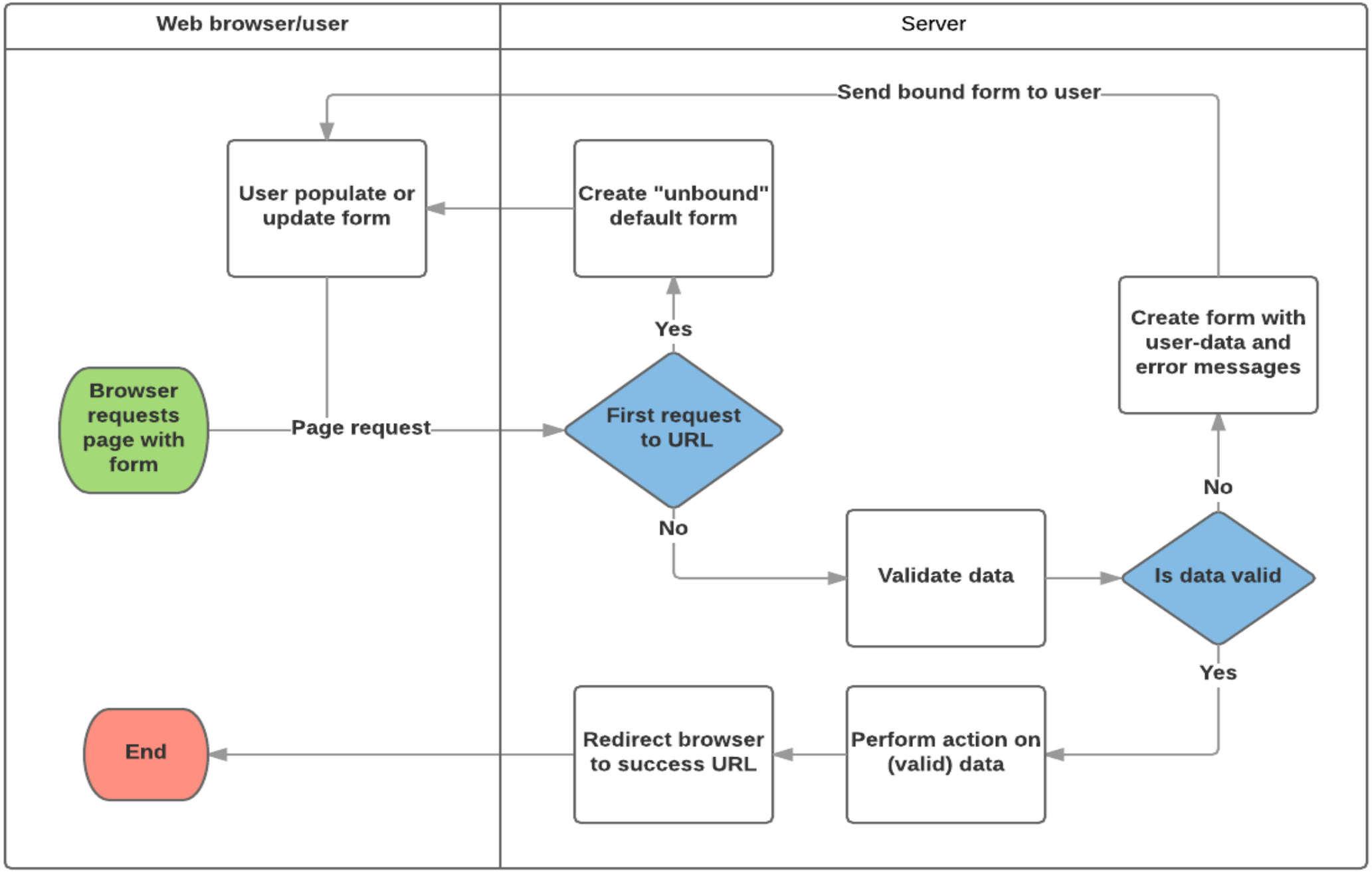
https://docs.djangoproject.com/en/5.1/**ref/forms/api/**

# Versione X: i moduli (forms)

- Concettualmente:
  - Va definita la struttura dati della form
  - Va creata una view che ne gestisca la logica
  - Va creato un template corrispondente e il relativo URL-mapping
  - Vanno creati punti di accesso dalle pagine preesistenti

# Versione X: i moduli (forms)

- Ciclo di vita di una form:
  1. Una form viene creata inizialmente con dati di default
  2. Viene poi renderizzata
  3. L'utente agisce riempiendo i campi ed alla fine con Submit
  4. Vengono eseguiti la validazione e il binding (server-side)
  5. A seconda dell'esito della validazione:
     - KO: la form viene ri-renderizzata e si riparte dallo step 3
     - OK: i dati vengono memorizzati e la pagina ridiretta


Web browser/user
Server
Send bound form to user
User populate or update form
Create "unbound" default form
Yes
Browser requests page with form
Page request
First request to URL
No
Validate data
Is data valid
No
Create form with user-data and error messages
Yes
End
Redirect browser to success URL
Perform action on (valid) data

# Versione X: i moduli (forms)

- Passo 1: creazione struttura dati della form
  - Le forms vanno definite dentro `forms.py` (file ex-novo)
  - Basata su classe `Form` di Django
  - Include:
    - La definizione campi, sulla base di tipi dati
    - Per ogni campo **C**, metodo opzionale di validazione `clean_C`:
      - Si appoggia su `cleaned_data` : dati esitati da validazione
      - Aggiunge tests ad-hoc
      - Puo' sollevare (`raise`) eccezioni di validazione
    - Metodo opzionale di validazione `clean` che esegue controlli che coinvolgono l'intera classe, eventualmente sollevando eccezioni

```
import datetime
from django import forms
from django.core.exceptions import ValidationError

class RenewBookForm(forms.Form):
    renewal_date = forms.DateField(help_text="Enter date.")

    def clean_renewal_date(self):
        data = self.cleaned_data['renewal_date']
        if data < datetime.date.today():
            raise ValidationError('Invalid - renewal in past')
        if data > (datetime.date.today() +
                   datetime.timedelta(weeks=4)):
            raise ValidationError('Invalid date - too late')
        return data
```

# Versione X: i moduli (forms)

- Passo 2: creazione di una FBV
  - Con accesso limitato a persone connesse: `@login_required`
  - In particolare, limitato ad addetti: `@permission_required`
  - Riceve in input l'identificativo (`pk`) della istanza del libro
  - Realizza la parte di logica server del lifecycle della form:
    - Se il metodo di accesso e' `GET`, e' il primo caricamento
    - Se e' `POST`, e' una richiesta con dati

# Versione X: i moduli (forms)

- Vari import necessari in `views.py`:

```
import datetime
from django.contrib.auth.decorators import login_required
from django.contrib.auth.decorators import permission_required
from django.shortcuts               import get_object_or_404
from catalog.forms                  import RenewBookForm
```

```
@login_required
@permission_required('catalog.can_mark_returned',
                     raise_exception = True)
def renew_book_librarian(request, pk):
    # Retrieve the book instance (or raise an error)
    book_instance = get_object_or_404(BookInstance, pk=pk)

    # If this is the first request (i.e. not a POST) then:
    # create the default form and render it
    if request.method != 'POST':
        p    = datetime.date.today() + datetime.timedelta(weeks=3)
        form = RenewBookForm(initial={'renewal_date': p })
        ctx  = { 'form': form, 'book_instance': book_instance }
        return render(request,
                      'catalog/book_renew_librarian.html', ctx)

    # ...otherwise...
```

```
# ...this is a POST request,so:
# - create a form instance and bind it with the POST data
# - validate the form
#   - if the form is valid, process the data in cleaned_data
#     and redirect to the page "exiting the form entry"
#   - if the form is not valid, re-render the page with errors
form = RenewBookForm(request.POST)
if form.is_valid():
    book_instance.due_back = form.cleaned_data['renewal_date']
    book_instance.save()
    return HttpResponseRedirect(reverse('all-loaned-books'))
else:
    ctx  = { 'form': form, 'book_instance': book_instance }
    return render(request,
                  'catalog/book_renew_librarian.html', ctx)
```

# Versione X: i moduli (forms)

- Passo 3: creazione del template chiamato dalla FBV, ovvero `templates/catalog/book_renew_librarian.html`
  - Riceve sia `form` che `book_instance`
  - Renderizza `form` con il metodo di supporto `.as_table`
  - Usa direttiva DTL `csrf_token` : metodo standard anti-hacking

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Renew:    {{ book_instance.book.title }}</h1>
  <p> Borrower: {{ book_instance.borrower   }}</p>
  <p{% if book_instance.is_overdue %} class="text-danger"{% endif %}>
    Due date: {{ book_instance.due_back }}
  </p>

  <form action="" method="post">
    {% csrf_token %}
    <table>
      {{ form.as_table }}
    </table>
    <input type="submit" value="Submit">
  </form>
{% endblock %}
```

# Versione X: i moduli (forms)

- Passo 4: aggiornamento dello URL mapping di applicazione:

```
urlpatterns = [ ... ,
                path('book/<uuid:pk>/renew/',
                     views.renew_book_librarian,
                     name='renew-book-librarian')
              ]
```

# Versione X: i moduli (forms)

- Passo 5: aggiungere link in `.../bookinstance_list_loaned.html`

```
...
<ul>
{% for b in bookinstance_list %}
   <li {% if b.is_overdue %} class="text-danger"{% endif %}>
     <a href="{% url 'book-detail' b.book.pk %}">
        {{ b.book.title }}</a>
     ({{ b.due_back }})
     {% if perms.catalog.can_mark_returned %}
         <a href="{% url 'renew-book-librarian' b.id %}">
            Renew</a>
     {% endif %}
   </li>
 {% endfor %}
</ul>
```

# Test!

- Ed ora...

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Ora agendo da addetti possiamo rinnovare i prestiti

# Versione X, addenda: pagina di sign-up

- Per ora, gli utenti vengono aggiunti dalla pagina amministrativa
- Ma se volessimo permettere ad utenti nuovi di iscriversi “da soli”?

- Soluzione:
  - Creare una pagina specifica con una form
  - Essa sarà accessibile da utente non loggato
  - Django prevede delle classi di support per form specifiche che si appoggiano al modello `User` da esso definito

# Versione X, addenda: pagina di sign-up

La form che usa la classe specifica `UserCreationForm`, dettando quali dei molti campi del modello dati `User` visualizzare:

```
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

class SignupForm(UserCreationForm):
    class Meta:
        model = User
        fields = ['first_name','last_name','email',
                  'username', 'password1', 'password2']
```

# Versione X, addenda: pagina di sign-up

La view, che usa la form e, se corretta, salva il dato, autentica il nuovo utente e lo associa al gruppo `Utenti` , utilizzando vari moduli. Import e codice:

```
from django.contrib.auth import authenticate, login
from django.contrib.auth.models import Group
from .forms import SignupForm

....
....
```

# Versione X, addenda: pagina di sign-up

```
....

def user_signup(request, next):
    if request.method != 'POST':
        form = SignupForm()
        return render(request, 'catalog/signup.html', {'form': form})
    else:
        form = SignupForm(request.POST)
        if form.is_valid():
            form.save()
            auth_user = authenticate(username=form.cleaned_data['username'],
                                     password=form.cleaned_data['password1'])
            login(request, auth_user)
            gruppo_utenti = Group.objects.get(name='Utenti')
            auth_user.groups.add(gruppo_utenti)
            return HttpResponseRedirect(next)
        else:
            return render(request, 'catalog/signup.html', {'form': form})
```

# Versione X, addenda: pagina di sign-up

Il template …/templates/catalog/signup.html :

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Signup</h1>
  <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Signup</button>
  </form>
{% endblock %}
```

# Versione X, addenda: pagina di sign-up

Infine, la regola URL in `urls.py` e la modifica al menu in `base_generic.html`:

```
urlpatterns = [ ... ,
                path('signup/<path:next>', views.user_signup, name='signup')]
```

...e la modifica in `base_generic.html`:

```
    <li>No user:
        <a href="{% url 'resetlogin' request.path %}">Login</a>
        <a href="{% url 'signup' request.path %}">Sign up</a>
    </li>
```

# Test!

- Ed ora...

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Ora un utente non autenticato può creare la propria utenza

# Note ulteriori sulle forms (1)

- Se serve creare una form per un modello dati esistente, si usa `ModelForm`:

```
class InputBookForm(forms.ModelForm):
  class Meta:
     model = Book
     fields = ["summary","isbn"]
```

- NB si può scegliere quali dei campi del modello usare: non necessariamente tutti.

# Note ulteriori sulle forms (2)

- La inizializzazione di una form è ridefinibile, se si desidera popolare in modo ad-hoc i campi della form. Ad esempio qui permetto di associare ad una `BookInstance` solo i libri il cui titolo inizia per A:

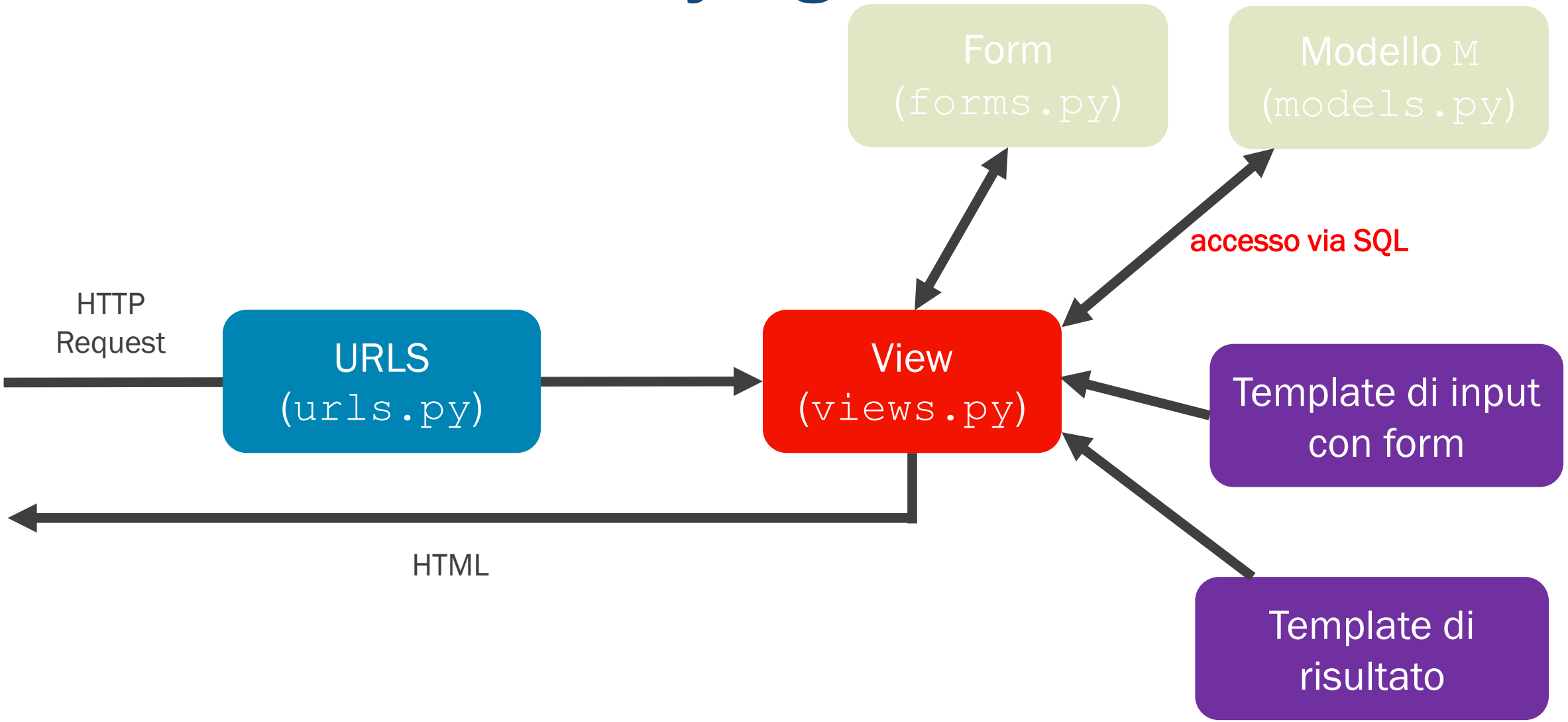
```
class myBookInstanceForm(forms.ModelForm):
  class Meta:
     model = BookInstance
     fields = ["book", "status", "borrower", "due_back"]

  def __init__ (self, *args, **kwargs):
      super (myBookInstanceForm, self).__init__(*args, **kwargs)
      self.fields["book"].queryset =
            Book.objects.filter(title__startswith= "A" )
```

# Versione XI: SQL e filtri

- Obiettivo: aggiungere pagina di "ricerca libri per inizio titolo"
  - La ricerca sara' realizzata come query SQL
  - La pagina sara' ristretta ad addetti connessi
- Concettualmente:
  - Va definita una form di ricerca
  - Va definita una pagina che la usa
  - Va linkata tale pagina alla home, con restrizione accesso
  - Quando la form e' corretta, fa partire la query SQL
  - Gli esiti vanno emessi da un template "di risultato"

# Django :


Form
(forms.py)
Modello M
(models.py)
accesso via SQL
HTTP
Request
URLS
(urls.py)
View
(views.py)
Template di input
con form
HTML
Template di
risultato

# Versione XI: SQL e filtri

- Passo 1: definizione della form in `forms.py`:

```
class SearchBookTitleForm(forms.Form):

    title = forms.CharField( max_length = 100,
                             required   = False,
                             help_text  = "")

    def clean_title(self):
        data = self.cleaned_data['title']
        return data
```

# Versione XI: SQL e filtri

- Passo 2: definizione della view:
  - La struttura e' la stessa di quella della form di rinnovo libri
  - Differenza: la costruzione del dato attraverso SQL
  - Viene usata la funzione `raw` di Django per una query del tipo

```
SELECT * FROM catalog_book WHERE title LIKE titolo%
```

```
from catalog.forms import SearchBookTitleForm

@login_required
@permission_required('catalog.can_mark_returned',
                     raise_exception = True)
def search_book_librarian(request):

    # If this is the first request then:
    # create and render the default form.
    if request.method != 'POST':
        form = SearchBookTitleForm(initial={'title': ''})
        ctx  = { 'form': form }
        return render(request,
                      'catalog/search_book_librarian.html', ctx)

    # otherwise...
```

```
form = SearchBookTitleForm(request.POST)
if form.is_valid(): # Form is valid: search, and render
  s = form.cleaned_data['title']
  # Raw SQL formulation of query
  search_SQL   = s + '%'
  found_books  = Book.objects.raw( \
    'SELECT * FROM catalog_book WHERE title LIKE %s',
    [search_SQL])
  ctx = { 'search_title': s,
          'found_books' : found_books   }
  return render(request,
                'catalog/searched_book_librarian.html', ctx)
else:      # Form is not valid: re-render it
  ctx  = { 'form': form }
  return render(request,
                'catalog/search_book_librarian.html', ctx)
```

# Versione XI: SQL e filtri

- Passo 3: definizione del template di input della form, `catalog/search_book_librarian.html`:

```
{% extends "base_generic.html" %}

{% block content %}
  <h1>Search by book title:</h1>
  <form action="" method="post">
    {% csrf_token %}
    <table>
    {{ form.as_table }}
    </table>
    <input type="submit" value="Submit">
  </form>
{% endblock %}
```

# Versione XI: SQL e filtri

- Passo 4: definizione del template di rendering del risultato, `catalog/searched_book_librarian.html`:

```
{% extends "base_generic.html" %}
{% block content %}
   <h1>Search results for this title: '{{ search_title }}'</h1>
   <br> <br>
   {% for book in found_books %}
      <hr>
      <p> <a href="{{ book.get_absolute_url }}">
          {{ book.title }}</a> ({{book.author}})</p>
      <hr>
   {% empty %}
      <h2 class="text-warning"> No book was found </h2>
   {% endfor %}
{% endblock %}
```

# Versione XI: SQL e filtri

- Passo 5: definizione dello URL mapping per la pagina di ricerca

```
urlpatterns =
    [...,
     path('search_book_librarian',
          views.search_book_librarian,
          name='search-book-librarian') ]
```

# Versione XI: SQL e filtri

- Passo 6: estensione di `base_generic.html` per linkare ricerca:

```
{% block sidebar %}
 ....
{% if user.is_authenticated %}
  ....
  {% if perms.catalog.can_mark_returned %}
   <li><a href="{% url 'all-loaned-books' %}">Loaned books</a></li>
   <li><a href="{% url 'search-book-librarian' %}">Search</a></li>
  {% endif %}
 ...
{% endblock %}
```

# Test!

- Ora:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Ora agendo da addetti possiamo cercare libri

# Versione XI: SQL e filtri (continued...)

- ...NB: Django gestisce anche le ricerche del tipo '`LIKE` **`title`**`%`' !
- Si ricordi che Django fornisce molte opzioni per esprimere ricerche:

  https://docs.djangoproject.com/en/5.1/**ref/models/querysets/**

- Si usa un filtro (`filter`) con '**`title`**`__startswith`'

```
found_books = Book.objects.filter(title__startswith = s)
```

- tuttavia si e' voluto mostrare come usare direttamente SQL

```
....

if form.is_valid(): # Form is valid: search, and render
  s = form.cleaned_data['title']
  # Raw SQL formulation of query
  search_SQL   = s + '%'
  found_books  = Book.objects.raw( \
    'SELECT * FROM catalog_book WHERE title LIKE %s',
    [search_SQL])
  # Equivalent filter-based formulation
  found_books = Book.objects.filter(title__startswith = s)
  ctx = { 'search_title': s,
          'found_books' : found_books   }
  return render(request,
                'catalog/searched_book_librarian.html', ctx)

....
```

# Test!

- Ri-testiamo:

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser
Apri la pagina `http://127.0.0.1:8000`

- Ora agendo da addetti possiamo cercare libri

# Versione XI: SQL e filtri (recap e esercizio)

- Django permette, attraverso i molti metodi dell'ORM, di accedere al DB senza l'uso esplicito di SQL. Questo copre moltissimi casi.
- Di fatto le primitive di ORM sono convertite in comandi SQL
- In alcuni casi pero' puo' essere comodo usare raw SQL...
- Esempio/esercizio: modificare la pagina di ricerca libri in modo che l'utente possa, volendo, determinare la search col carattere di wild-card ('*'), es. cercando `Prom*` oppure `*essi*`

# Versione XI: SQL e filtri (continued...)

- Semplice soluzione usando raw SQL: rimpiazzare la wildcard

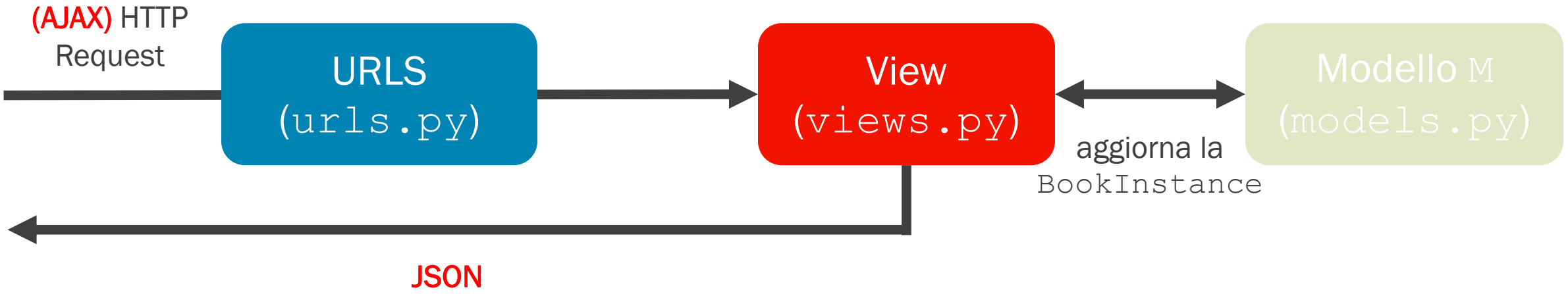
```
form = SearchBookTitleForm(request.POST)
if form.is_valid(): # Form is valid: search, and render
  s = form.cleaned_data['title']
  # Raw SQL formulation of query
  search_SQL   = s + '%'
  search_SQL   = s.replace("*","%")
  found_books  = Book.objects.raw( \
    'SELECT * FROM catalog_book WHERE title LIKE %s',
    [search_SQL])
  ...
```

- NB: esiste il metodo ORM `contains` , ma e' soluzione non flessibile

# Versione XII: Javascript e AJAX

- Obiettivo: aggiungere, sulla lista delle copie dei libri, bottoni che permettano all'addetto di porre ogni copia disponibile in manutenzione
- Concettualmente:
  - Lo vogliamo fare "senza creazione di pagine ex-novo"
  - Soluzione: Javascript asincrono (AJAX) ed eventi DOM
  - Il bottone causera' una richiesta asincrona HTTP verso una URL
  - La view collegata alla URL non fara' rendering ma agira' sul DB
  - La sua risposta sara' usata per modificare (il DOM del)la pagina

# Django :


(AJAX) HTTP Request
URLS (urls.py)
View (views.py)
Modello M (models.py)
aggiorna la BookInstance
JSON

# Versione XII: Javascript e AJAX

- Passo 1: modificare il template `book_detail.html`
  - Nella lista copie aggiungiamo gli HTML `button`s, solo per le istanze di libro in stato di disponibilita'
  - Diamo attributi `id` agli elementi, per poterli poi accedere
  - I bottoni chiamano uno script passando l'ID della copia libro

```
...
{% for copy in book.bookinstance_set.all %}
   <hr>
   <p id="para-{{copy.id}}"
      class="{% if copy.status == 'a' %}text-success
             {% elif copy.is_overdue %}text-danger
             {% else %}text-warning{% endif %}">
      {{ copy.get_status_display }}
      {% if perms.catalog.can_mark_returned %}
         {% if copy.status == 'a' %}
            <span id="span-{{copy.id}}">
               <button
                  onclick="setOnMaintenance('{{ copy.id }}')">
                  Put on maintenance
               </button>
            </span>
         {% endif %}
      {% endif %}
   </p>
   ...
```

# Versione XII: Javascript e AJAX

- Passo 2: nell'head di `base_generic.html` carico il Javascript

```
<head>
  ....
  {% load static %}
  <link rel="stylesheet" href="{% static 'css/styles.css' %}">
  <link rel="stylesheet" href="{% static 'css/home_layout.css' %}">
  <link rel="shortcut icon" type="image/png"
        href="{% static 'images/favicon.png' %}" >
  <script src="{% static 'js/dynamic.js' %}"></script>
</head>
```

# Versione XII: Javascript e AJAX

- Passo 3: il codice Javascript nel file `static/js/dynamic.js`:
  - Di fatto: una chiamata AJAX con relativa gestione risposta
  - Noi useremo l'interfaccia nativa `XMLHttpRequest`:
    - Creiamo un oggetto `XMLHttpRequest`
    - Definiamo la funzione di ritorno `onload`
      - Aspettera' e parsera' un formato JSON; campi attesi: `new_button`, `new_status` , `new_style`
      - I contenuti saranno usati per modificare il DOM
    - Apriamo e inviamo una HTTP `GET` , incorporando nella URL l'ID della copia del libro (per poi passarla al template)

```
function setOnMaintenance (id)
{
  const xhttp = new XMLHttpRequest();
  xhttp.onload = function()
   {
    obj = JSON.parse(this.responseText);
    document.getElementById("span-" + id).innerHTML = obj.new_button;
    document.getElementById("para-" + id).innerHTML = obj.new_status;
    document.getElementById("para-" + id).className = obj.new_style;
   };
  xhttp.open("GET", "/catalog/set_book_on_maintenance/" + id);
  xhttp.send();
}
```

# Versione XII: Javascript e AJAX

- Passo 4: definizione dello URL mapping per la invocazione AJAX

```
urlpatterns = [...,
               path('set_book_on_maintenance/<uuid:pk>',
                    views.set_book_on_maintenance,
                    name='set-book-on-maintenance')
              ]
```

# Versione XII: Javascript e AJAX

- Passo 5: function-based view di gestione call AJAX
  - Usa il dato passato dall'URL mapping (ID del libro)
  - Setta e salva lo stato del libro
  - Ritorna una struttura JSON attraverso il modulo `JsonResponse`

```
from django.http import JsonResponse

def set_book_on_maintenance (request,pk):
    book_instance = get_object_or_404(BookInstance, pk=pk)
    book_instance.status = 'd'
    book_instance.save()
    return JsonResponse({'outcome'    : 'OK',
                         'new_status': 'Maintenance',
                         'new_style' : 'text-warning',
                         'new_button': ''})
```

# Test!

```
python3 manage.py makemigrations
python3 manage.py migrate
python3 manage.py runserver
```

Apri un browser

Apri la pagina `http://127.0.0.1:8000`

NB: forza il reload dei files statici (shift-F5)

- Ora agendo da addetti possiamo porre libri in revisione

# Versione XII: Javascript e AJAX

- Note: esistono diverse alternative a `XMLHttpRequest`:
  - `Fetch API`
    - Interfaccia nativa in ECMAScript 6
    - Semplifica alcuni aspetti
    - Tuttavia richiede di conoscere le `promises`: transeat
  - `jQuery`
    - Notissima libreria esterna
    - Interfaccia molto semplice
    - Evitiamo di trattare una altra libreria esterna: transeat

# Versione XIII: testing aggiorna con settings.py 21 maggio 2025!!!

# Versione XIII: testing

- Obiettivo: corredare il progetto con una suite di testing
- Django supporta la scrittura e la esecuzione di test automatici
- Rif.: https://docs.djangoproject.com/en/5.1/**topics/testing/**

# Versione XIII: testing

- Concettualmente:
  - Si aggiungono classi che meccanizzano passi di test
  - Ci si basa su classi di supporto
  - I test vanno strutturati in classi e metodi di test
  - Ogni classe consiste di:
    - Set-up (tipicamente del  DB)
    - Metodi di test, ognuno con asserzioni
    - Chiusura (se necessaria)

# Versione XIII: testing

- Suites di test Django:
  - Ogni file di app `test*` e' identificato come suite di test
  - Uno scheletro e' generato in `tests.py`

- Organizziamo una struttura piu' limpida:
  - Dividiamo in moduli di test per views, modelli e forms
  - NB: non c'e' supporto al "test del rendering" !
  - Ogni modulo conterra' classi, e ogni classe metodi di test

# Versione XIII: testing

- Eliminiamo il file di scheletro vuoto `tests.py`
- Generiamo la struttura ex-novo a livello di app, con files vuoti:

```
tests
├── __init__.py
├── tests_models.py
├── tests_views.py
└── tests_forms.py
```

# Versione XIII: testing

- Per le suites di test, Django (via `manage.py`) permette diversi usi:

```
test                                    : Tutti i test nella app
test catalog.tests                      : Tutti i test nel folder tests
test catalog.tests.test_models                          : Un modulo
test catalog.tests.test_models.myClass                  : Una classe
test catalog.tests.test_models.myClass.myTest           : Un metodo
```

- Opzione importante: `--verbosity` **`N`** : 0,1,2,3, man mano piu' info

# Versione XIII: testing

- Tipologie di test:
  - Unit test : test su singoli moduli di codice
  - Integration test : test su integrazione di moduli di codice
  - Regression test : test comparativo su scenari di riferimento
  - …e molte altre…
- **Noi ci concentreremo su unit testing!**

# Versione XIII: testing

- Tests sul modello, in `tests_models.py`:
  - La classe di supporto di riferimento e' `TestCase` ; prevede
    - Un class method di setup `setUpTestData` (annotato con `@classmethod`) in cui si crea il DB di test
    - Metodi di test, ognuno con le sue asserzioni:
      - `assertEqual`
      - `assertTrue`
      - `assertFalse`
      - ...

- Esempio: test sulla lunghezza di un campo del modello `Author`

```
from django.test import TestCase
from catalog.models import Author

class AuthorModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up objects
        Author.objects.create(first_name='Big',last_name='Bob')

    def test_first_name_max_length(self):
        author = Author.objects.get(id=1)
        max_length =
            author._meta.get_field('first_name').max_length
        self.assertEqual(max_length, 100)
```

# Test!

```
python3 manage.py test
```

(od anche, indirizza la classe specifica)

Sperimentiamo le diverse `verbosity`

```
python3 manage.py test --verbosity 1
python3 manage.py test --verbosity 2
python3 manage.py test --verbosity 3
```

# Versione XIII: testing

- I test su forms, in `tests_forms.py`, sono analoghi a quelli sui dati
- Esempio: un test che crea una form errata, e valida che lo sia

```
import datetime
from django.test import TestCase
from catalog.forms import RenewBookForm

class RenewBookFormTest(TestCase):

    def test_renew_form_date_in_past(self):
        date = datetime.date.today() -
               datetime.timedelta(days=1)
        form = RenewBookForm(data={'renewal_date': date})
        self.assertFalse(form.is_valid())
```

# Versione XIII: testing

- Due ulteriori metodi di test in `RenewBookFormTest`:

```
    def test_renew_form_date_in_future_too_far(self):
        date = datetime.date.today() +
               datetime.timedelta(days=90)
        form = RenewBookForm(data={'renewal_date': date})
        self.assertFalse(form.is_valid())

    def test_renew_form_date_ok(self):
        date = datetime.date.today() +
               datetime.timedelta(days=7)
        form = RenewBookForm(data={'renewal_date': date})
        self.assertTrue(form.is_valid())
```

# Test!

```
python3 manage.py test
```

(od anche, indirizza la classe specifica)

Sperimentiamo le diverse `verbosity`

```
python3 manage.py test --verbosity 1
python3 manage.py test --verbosity 2
python3 manage.py test --verbosity 3
```

# Versione XIII: testing

- Tests sulle views, in `tests_views.py`:
  - per poter "attivare" le views, bisogna simulare azioni dal client
  - Per questo, in `TestCase` e' a disposizione un oggetto `client`
  - `client` permette di simulare comandi HTTP
  - `client` permette di simulare azioni utente di login/logout
  - Spesso servira' creare utenti in fase di setup (`create_user`)

# Versione XIII: testing

- Esempio semplice: verifichiamo di poter accedere alla home:

```
from django.test import TestCase

class HomeViewTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        pass

    def test_view_url_exists_at_desired_location(self):
        response = self.client.get('/catalog/')
        self.assertEqual(response.status_code, 200)
```

# Test!

```
python3 manage.py test
```

(od anche, indirizza la classe specifica)

Sperimentiamo le diverse `verbosity`

```
python3 manage.py test --verbosity 1
python3 manage.py test --verbosity 2
python3 manage.py test --verbosity 3
```

# Versione XIII: testing

- Esempio: verifica di accesso; setup...:

```
from catalog.models import User

class AuthorListViewTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        # Create user, since page is login-restricted
        test_user1 = User.objects.create_user
            (username='user1', password='1X<ISRUkw+tuK')
        test_user1.save()
```

# Versione XIII: testing

- Metodo per test di accesso con successo alla pagina:

```
def test_view_url_exists_at_desired_location(self):
    self.client.login (username='user1',
                       password='1X<ISRUkw+tuK')
    response = self.client.get('/catalog/authors/')
    self.assertEqual(str(response.context['user']),'user1')
    self.assertEqual(response.status_code, 200)
```

# Versione XIII: testing

- Metodo per test di accesso fallito alla pagina:

```
def test_view_url_exists_at_desired_location_nologin(self):
    # Notice: no login, so page access shall be refused.
    response = self.client.get('/catalog/authors/')
    self.assertFalse(response.status_code == 200)
```

# Test!

```
python3 manage.py test
```

(od anche, indirizza la classe specifica)

Sperimentiamo le diverse `verbosity`

```
python3 manage.py test --verbosity 1
python3 manage.py test --verbosity 2
python3 manage.py test --verbosity 3
```

# Versione XIII: testing

Altri due metodi di test, ma basati su istanza di libro:

- Verifica di non accedere a libro prestato senza autenticazione
- Verifica di accede a libro prestato se autenticati
- Altri import iniziali...

```
import datetime
from django.urls import reverse
from catalog.models import Author, BookInstance, Book,
                           Genre, Language
from django.contrib.auth.models import Permission
```

```
class RenewBookInstancesViewTest(TestCase):
    # Setup, test-based
    def setUp(self):
        # Create 2 users, one with permission and one without
        test_user1 = User.objects.create_user(
              username='user1', password='1X<ISRUkw+tuK')
        test_user1.save()
        test_user2 = User.objects.create_user(
              username='user2', password='2HJ1vRV0Z&3iD')
        test_user2.save()

        permission = Permission.objects.get(name=
                                'Set book as returned')
        test_user2.user_permissions.add(permission)
        test_user2.save()
```

```
# Create a book
test_author = Author.objects.create
        (first_name='John', last_name='Smith')
test_genre = Genre.objects.create(name='Fantasy')
test_language = Language.objects.create(name='English')
test_book = Book.objects.create(
                title='Book Title',
                summary='My book summary',
                isbn='ABCDEFG',
                author=test_author,
                language=test_language,
                genre=test_genre)
test_book.save()
```

```
        # Create two BookInstance objects
        return_date = datetime.date.today() +
                      datetime.timedelta(days=5)
        self.test_bookinstance1 = BookInstance.objects.create(
                                      book=test_book,
                                      imprint='Imprint, 2016',
                                      due_back=return_date,
                                      borrower=test_user1,
                                      status='o')
        self.test_bookinstance2 = BookInstance.objects.create(
                                      book=test_book,
                                      imprint='Imprint, 2019',
                                      due_back=return_date,
                                      borrower=test_user2,
                                      status='o')
```

## Test: `user_1` e' autenticato ma non ha il permesso di rinnovare il prestito.

```
def test_403_if_logged_in_but_no_permission(self):
    self.client.login (username='user1',
                       password='1X<ISRUkw+tuK')
    response = self.client.get (
               reverse('renew-book-librarian',
               kwargs={'pk': self.test_bookinstance1.pk}))
    self.assertEqual(response.status_code, 403)
```

# Test: `user_2` e' autenticato ed e' abilitato a rinnovare il prestito.

```
def test_ok_logged_in_librarian_borrowed_book(self):
  self.client.login (username='user2',
                     password='2HJ1vRV0Z&3iD')
  response = self.client.get (
              reverse('renew-book-librarian',
              kwargs={'pk': self.test_bookinstance2.pk}))
  self.assertEqual(response.status_code, 200)
```

# Test!

```
python3 manage.py test
```

(od anche, indirizza la classe specifica)

Sperimentiamo le diverse `verbosity`

```
python3 manage.py test --verbosity 1
python3 manage.py test --verbosity 2
python3 manage.py test --verbosity 3
```

# Note sul deploy

- Tipicamente un ambiente di sviluppo non e' ok per produzione:
  - Messa in produzione : agisce da server
    - Deve essere sempre attivo
    - Deve essere sempre connesso a Internet, con IP statico
  - Tipicamente, ambiente di sviluppo:
    - Non sempre attivo
    - Non sempre connesso
    - Connessione con DHCP

# Note sul deploy

- Deploy in produzione:
  - Identificare un ambiente adatto (sempre connesso, IP statico)
  - Installarci quanto serve (Django, files di sviluppo, ...)
  - **Ma richiede diverse modifiche anche a livello di settaggi!**

- Ambienti di produzione:
  - Tipicamente non si crea "da zero"
  - Infrastrutture a pagamento (o gratis in casi base)
  - Si categorizzano in IaaS e PaaS

# Note sul deploy

- IaaS: Infrastructure As A Service
  - Forniscono solo infrastruttura:
    - Sistema operativo
    - Web server, application server
    - Database
    - Linguaggi di programmazione
  - Massima flessibilita' in cio' che si installa (fwk o altro)
  - Richiede molto know-how

# Note sul deploy

- PaaS: Platform As A Service
  - Forniscono infrastruttura piu' fwk di sviluppo (es. Django)
  - Meno flessibili
  - Piu' semplici... ma comunque servono diversi interventi!

  - Da qui, ci riferiremo a un deploy su una PaaS generica

# Note sul deploy

Passi generali:

1. Interventi sui settings Django (test <> produzione)
2. Creazione del contenitore dei dati nella PaaS
3. Creazione dei settaggi nella PaaS (dipendenze, startup, etc.)
4. Passaggio dei dati nel contenitore PaaS
5. Creazione della struttura DB lato PaaS (test <> produzione)
6. (Ev.) installazione di una CLI remota
7. Creazione di utenti e dati via CLI / pagine admin

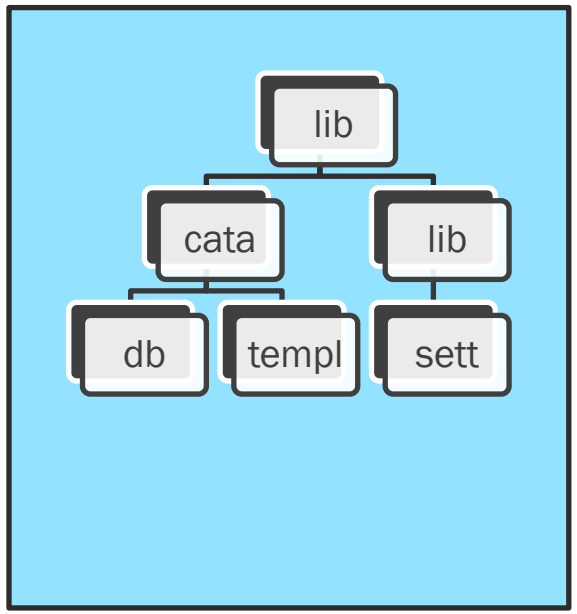
# Note sul deploy

Alcune PaaS di esempio: `Railway`, `PythonAnywhere`

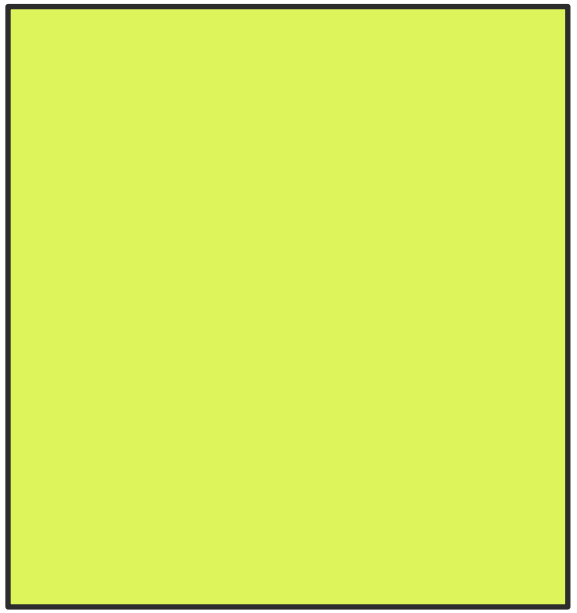
1. Supportano Django
2. Forniscono un contenitore virtualizzato per webapp
3. Hanno forte integrazione con GIT e GITHub
   - GIT: sistema di version control per progetti software
   - GITHub: servizio di cloud repository per progetti GIT
4. A maggio 2025, `PythonAnywhere` ha un piano gratuito che permette di deployare una singola web application.

NB: la gratuità dei piani è dinamica e va man mano confermata.

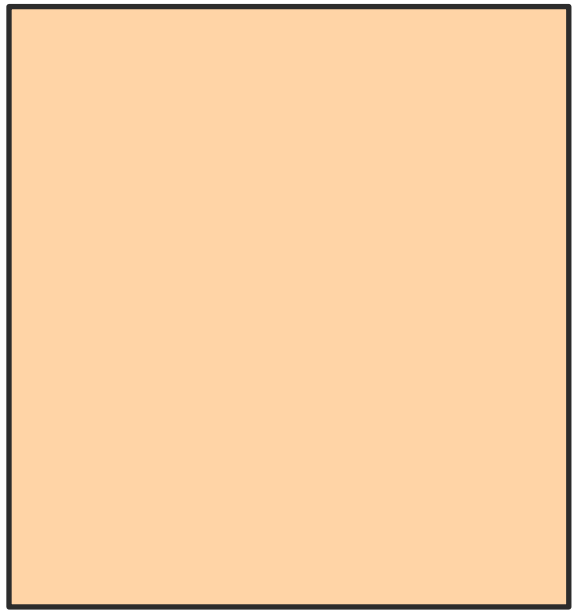
# Processo di deploy, ad alto livello




Locale

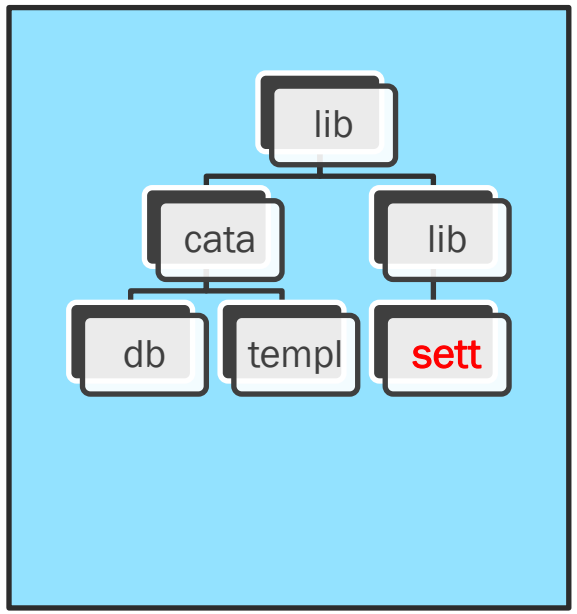


GitHub

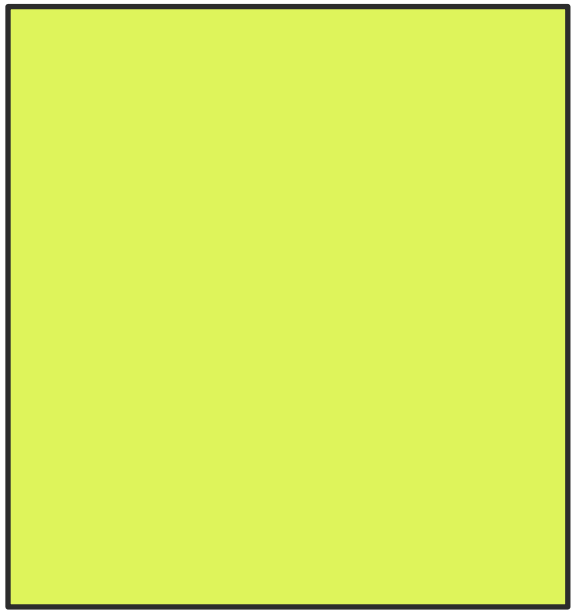


Railway PaaS

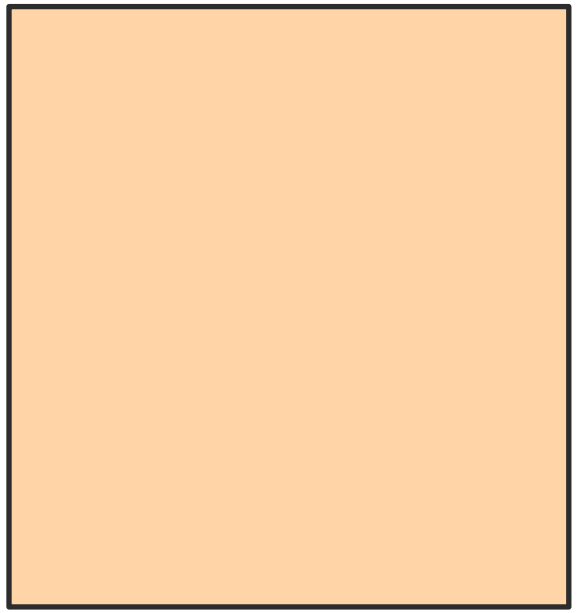
# Processo di deploy, ad alto livello




Locale

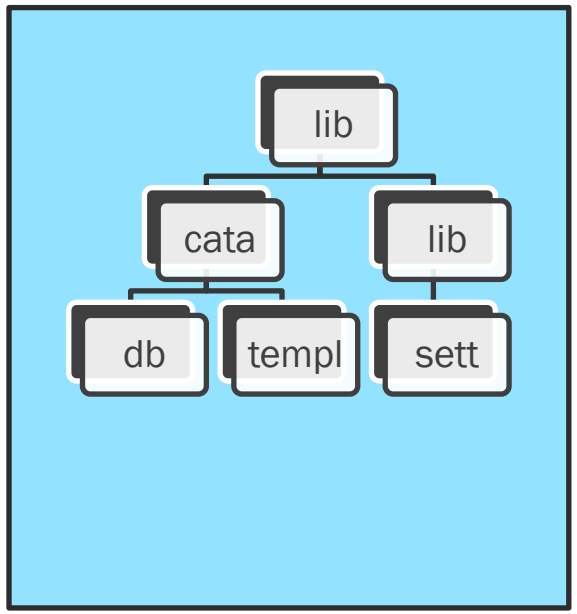


GitHub

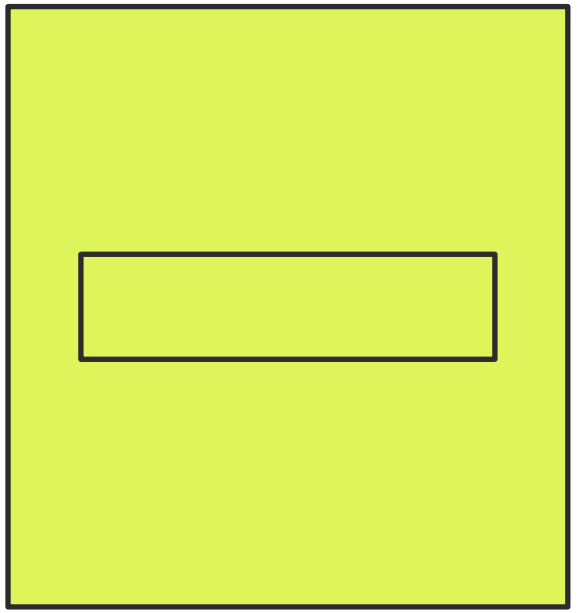


Railway PaaS

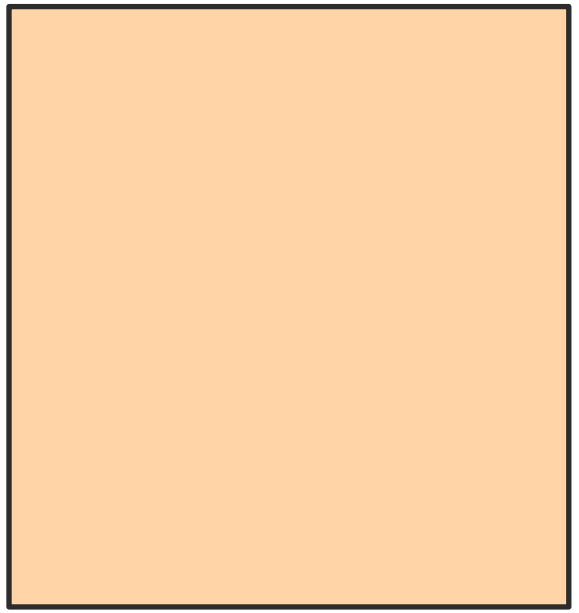
# Processo di deploy, ad alto livello




Locale

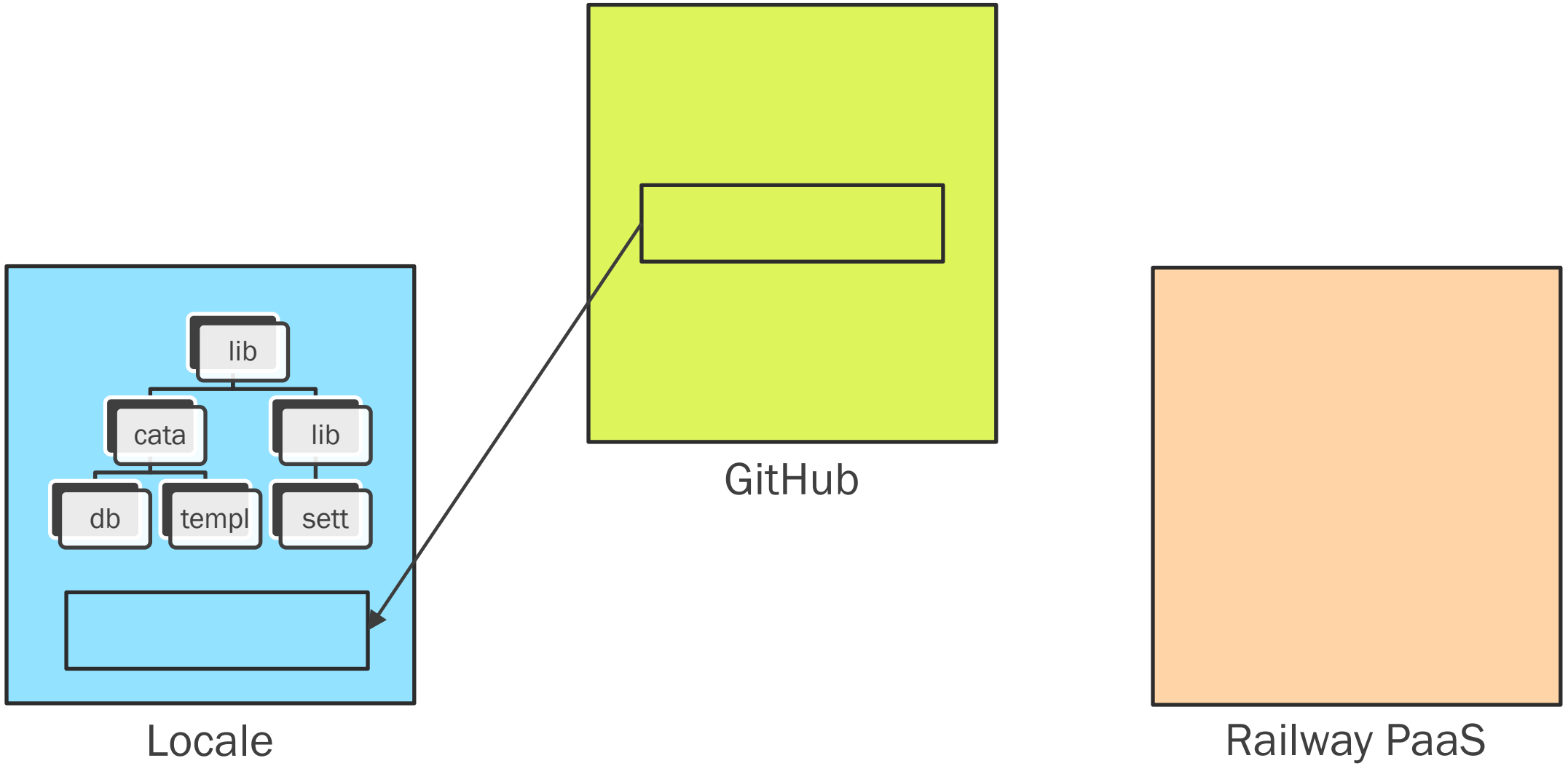


GitHub

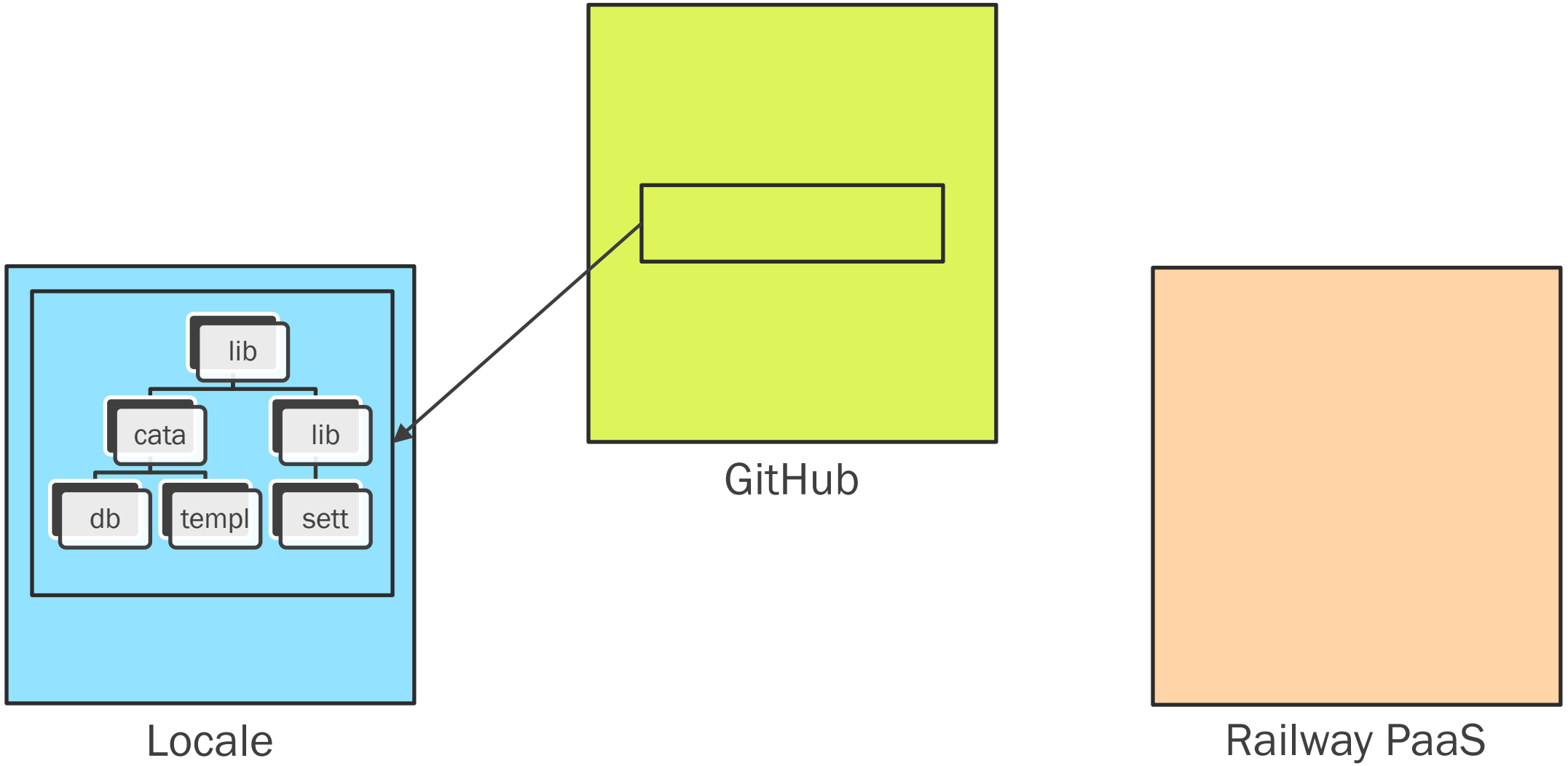


Railway PaaS

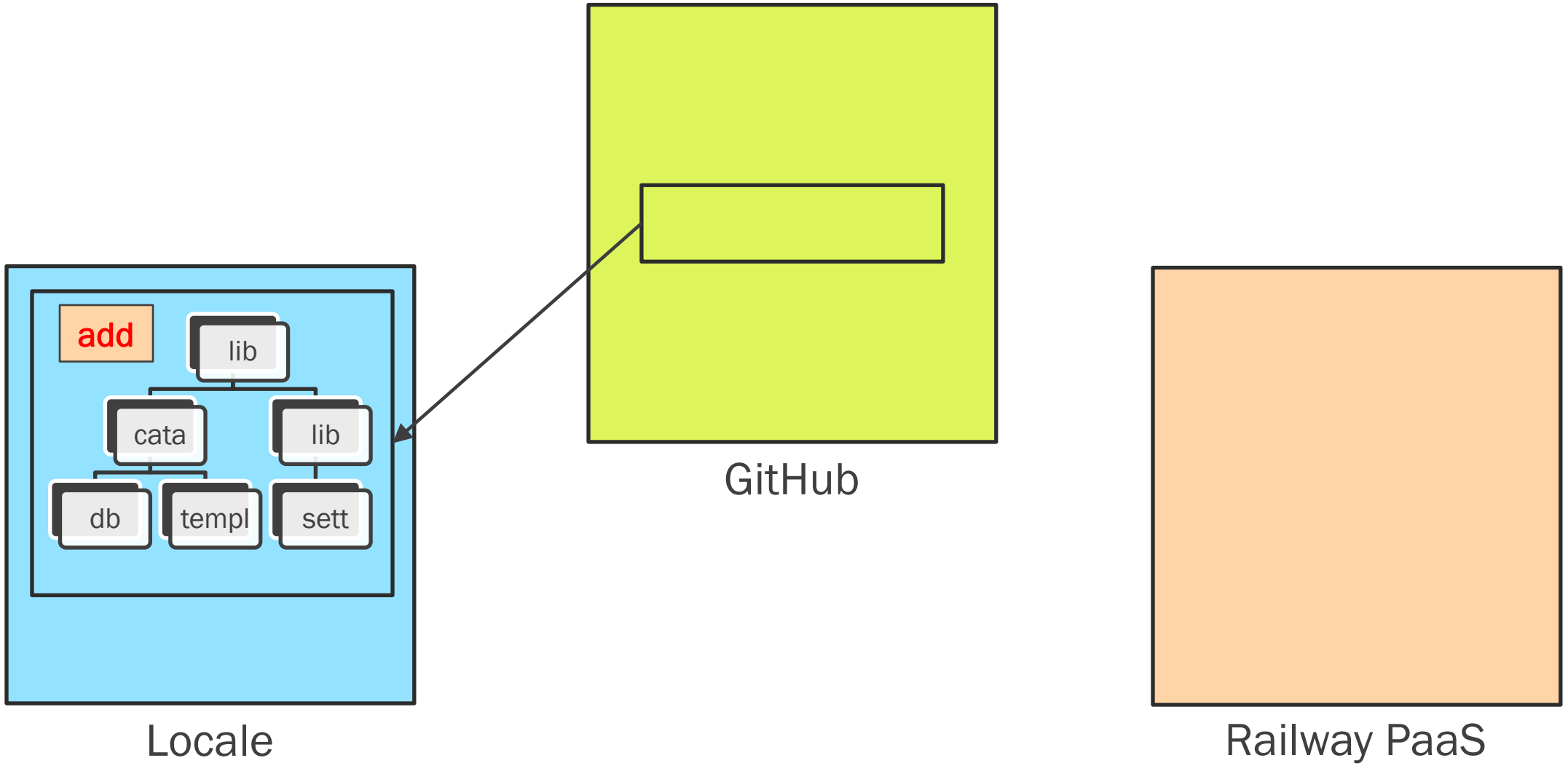
# Processo di deploy, ad alto livello



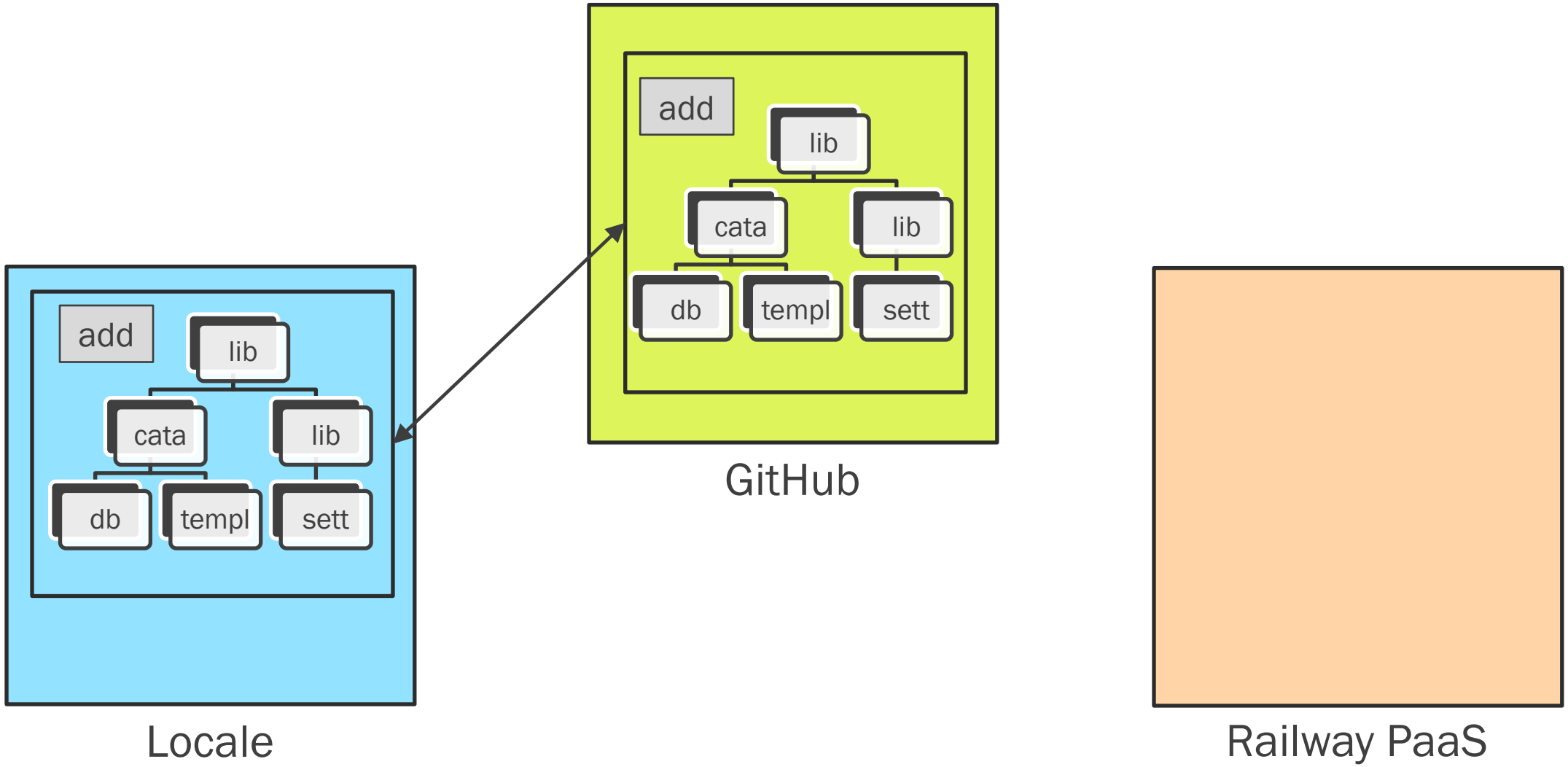
# Processo di deploy, ad alto livello



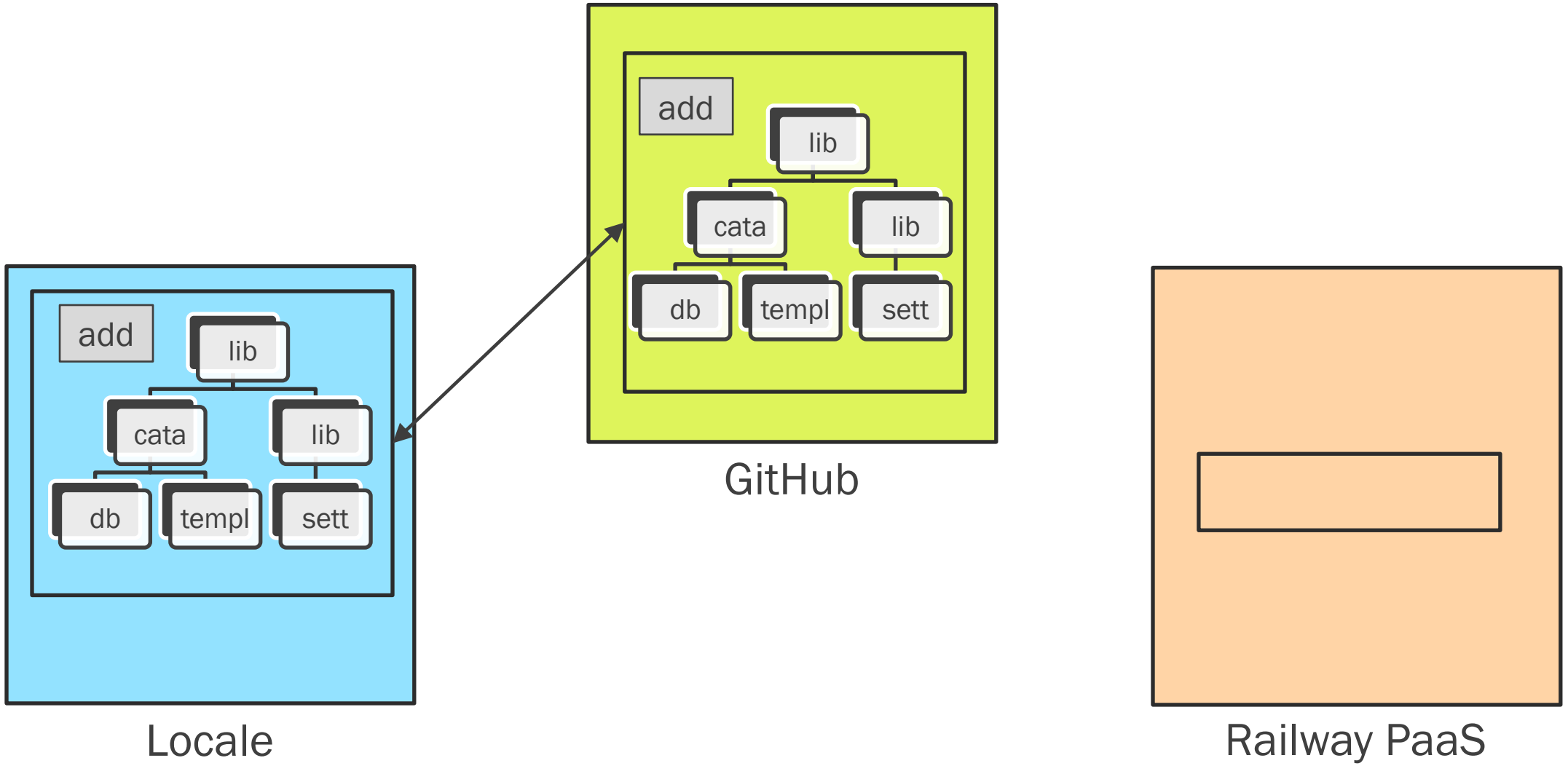
# Processo di deploy, ad alto livello



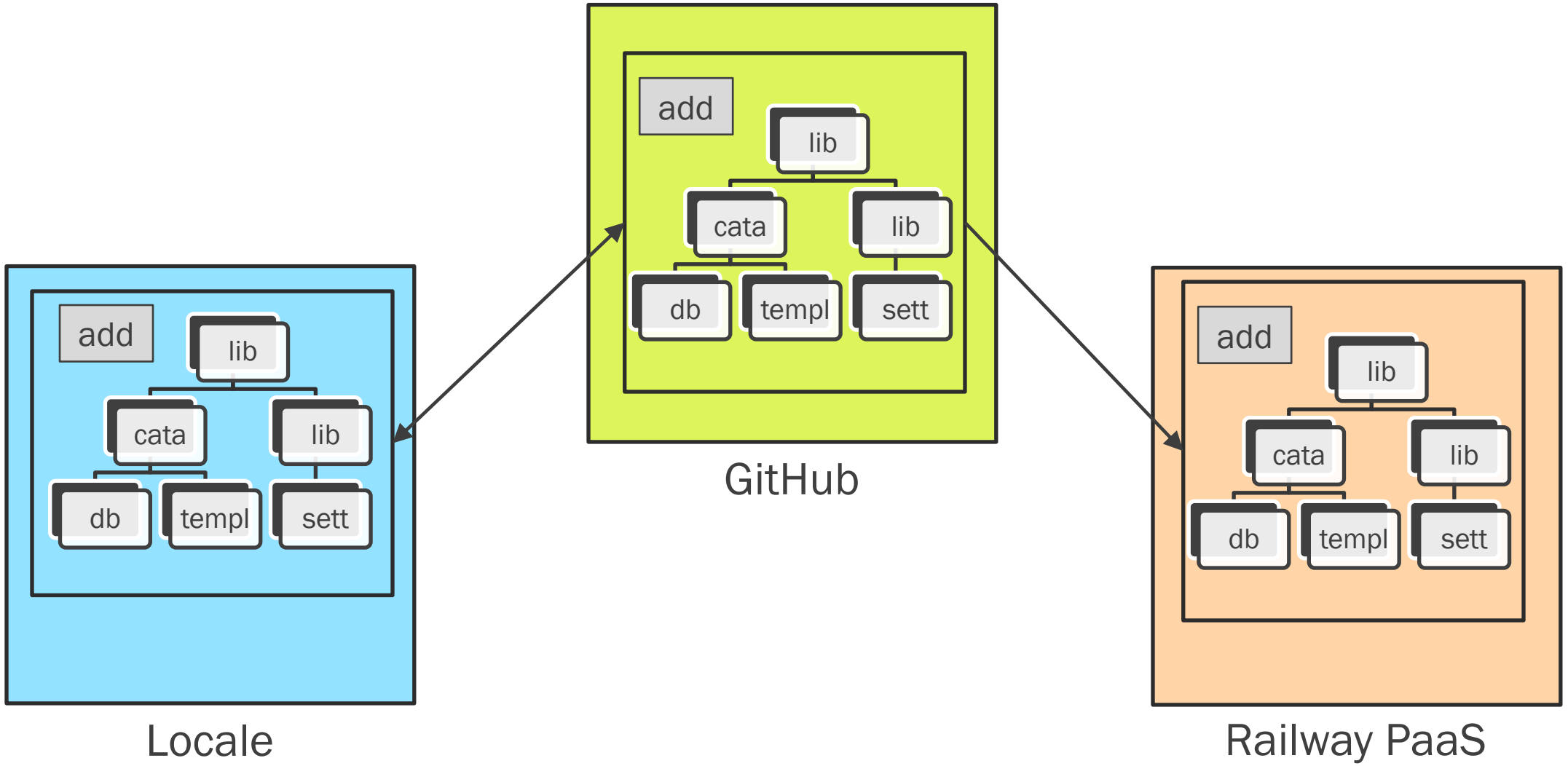
# Processo di deploy, ad alto livello



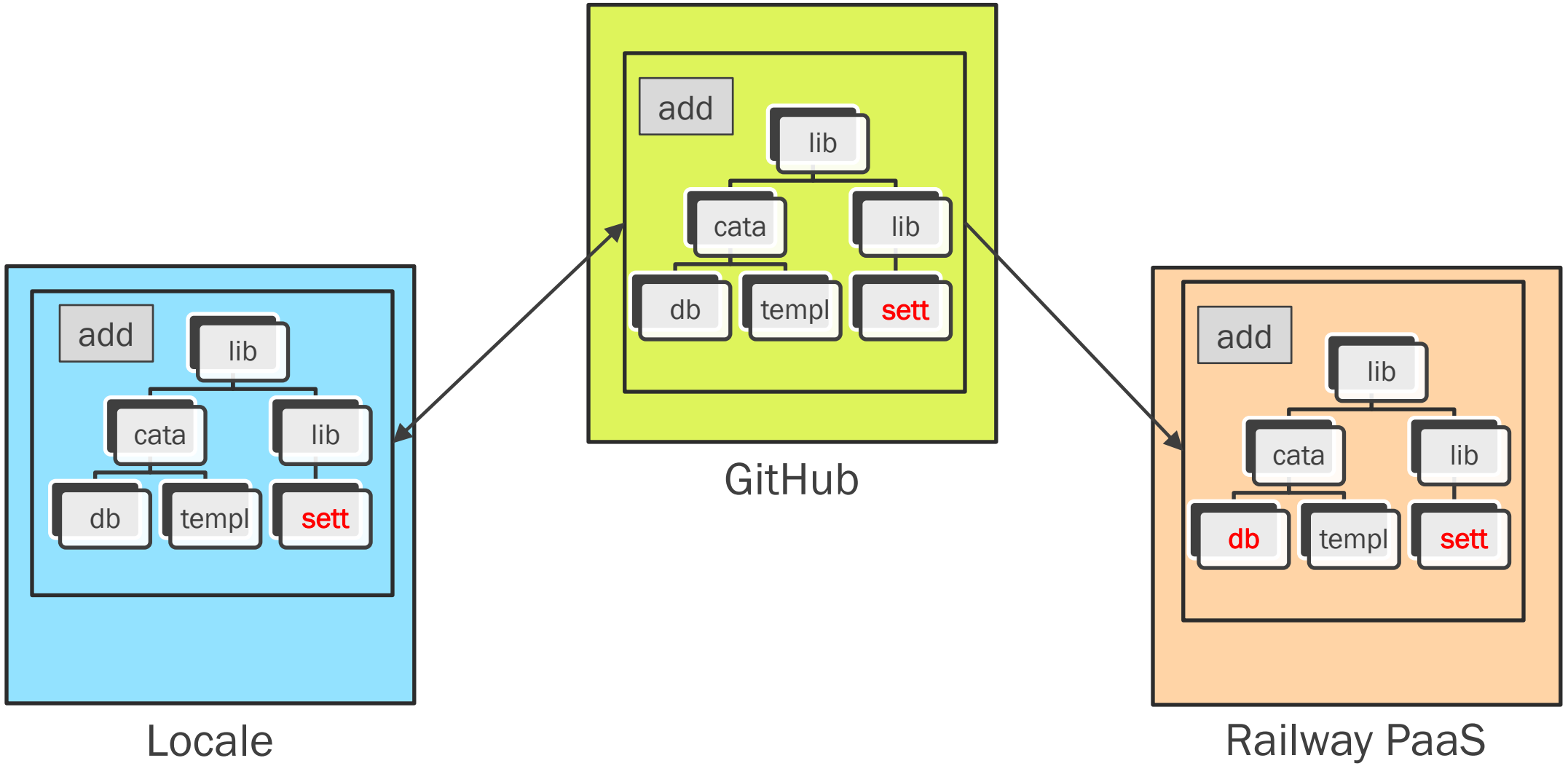
# Processo di deploy, ad alto livello



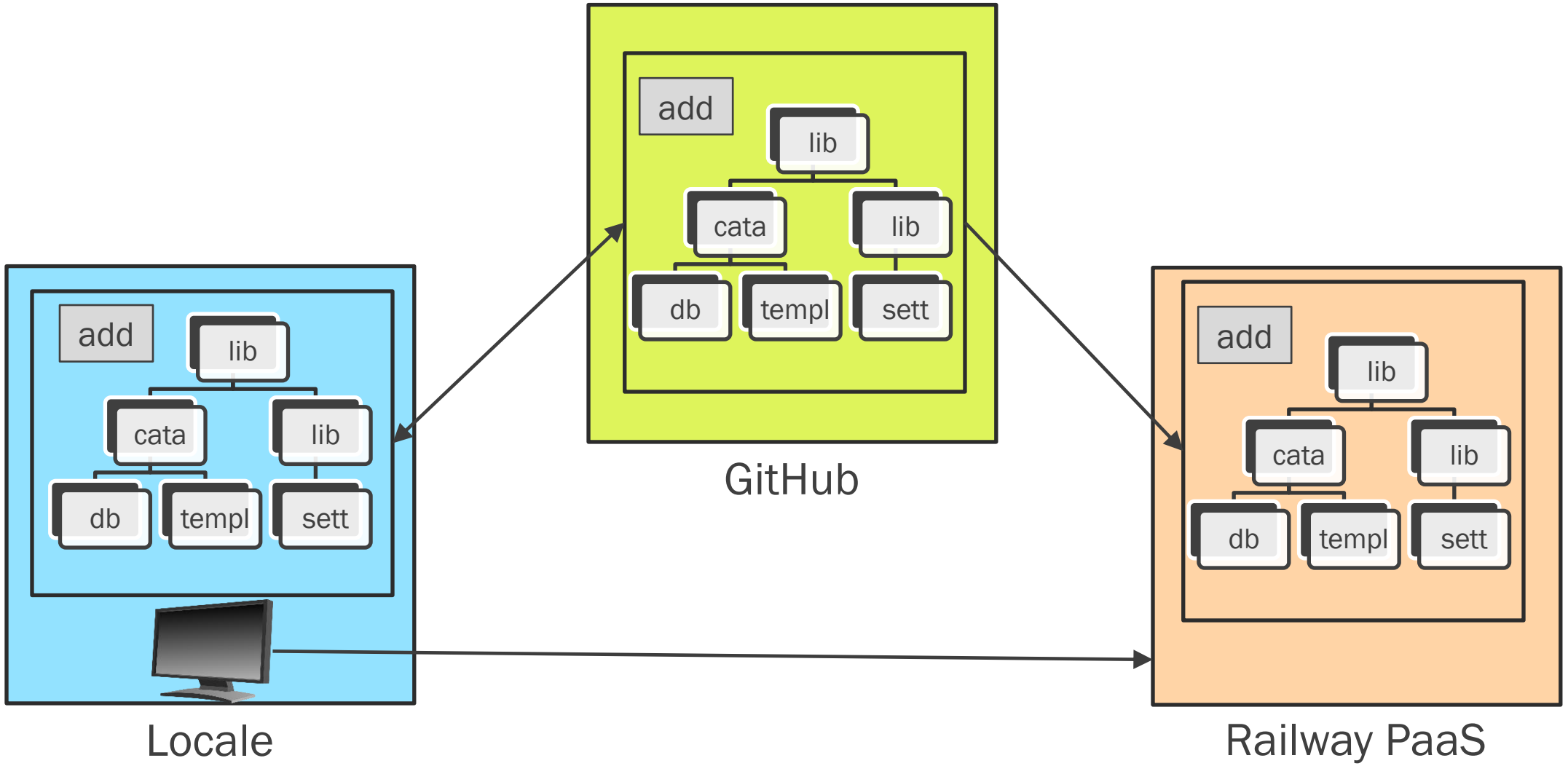
# Processo di deploy, ad alto livello



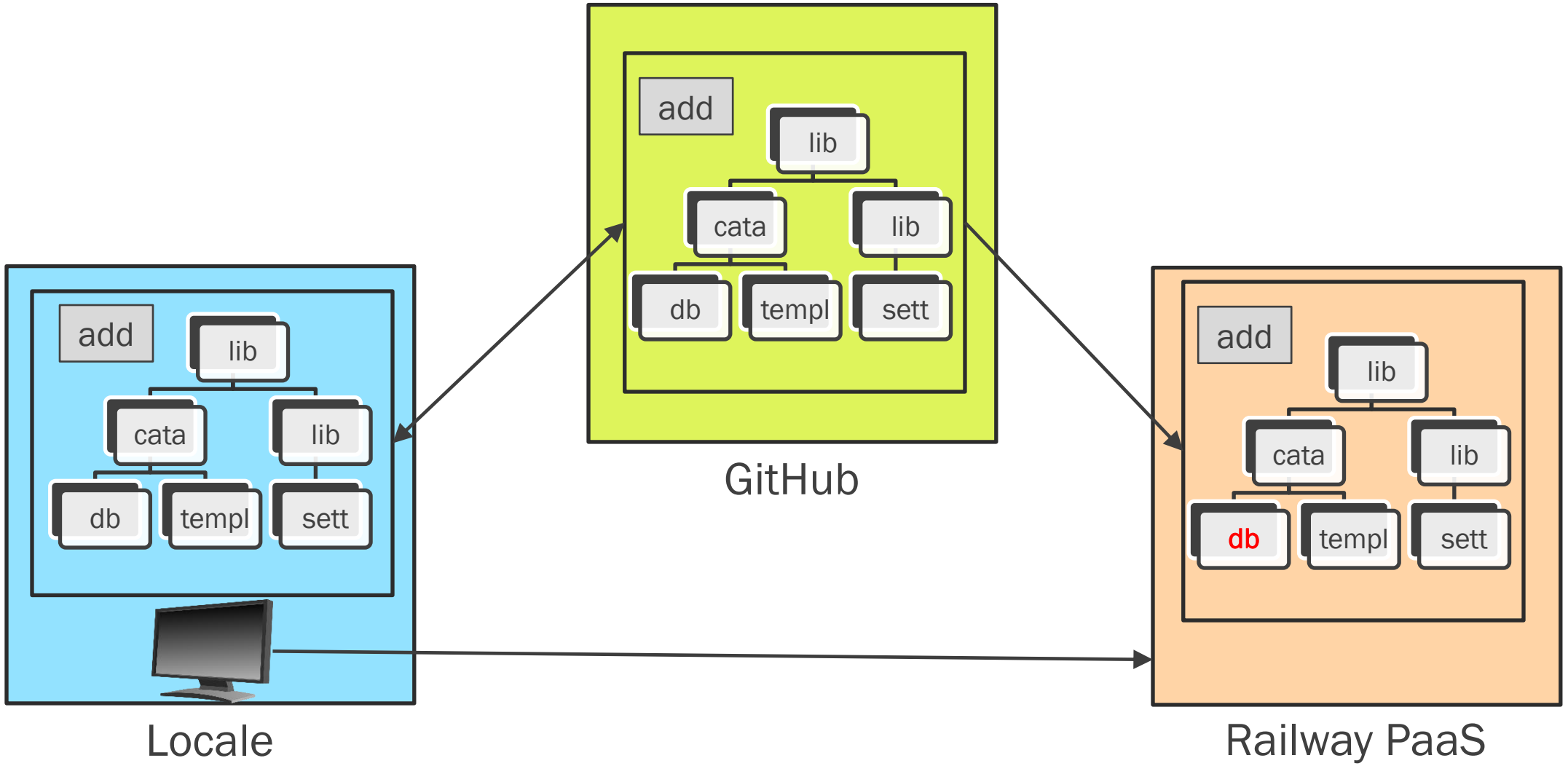
# Processo di deploy, ad alto livello



# Processo di deploy, ad alto livello



# Processo di deploy, ad alto livello



# Note sul deploy

- Step 1: interventi iniziali sui settings:
  - Servono a preparare `settings.py` a quanto servirà nella PaaS
  - Conviene tenersi anche una copia del file "da sviluppo locale"
  - Settaggi (generali) relativi alla sicurezza
  - Settaggi (tecnici e specifici) relativi al DB
  - Settaggi relativi alla gestione dei files statici
  - Per una lista (non del tutto esaustiva) di interventi, vedi:

https://docs.djangoproject.com/en/5.1/howto/deployment/checklist/

# Note sul deploy

- Step 2: installazione in locale delle app necessarie in produzione:
  - Questo serve per poter creare (con `pip`) un file delle dipendenze
  - Il file delle dipendenze, una volta deployato, porterà a poter riprodurre l'ambiente corretto nella PaaS
  - Le app tipicamente contengono quanto serve per gestire i files statici, per interfacciare Django con DB diversi da SQLite, etc.

# Note sul deploy

- Step 3: creazione del repository di progetto:
  - Va creato un account GIT ed un token di accesso
  - Va creato un repository (vuoto) di progetto `GIT`
  - Va clonato localmente il repository vuoto
  - Va riempito con i files della nostra app
  - Vanno configurati i files di non-interesse per `GIT`
  - Va allineato il repository (add + commit + push)

# Note sul deploy

- Step 4: preparazione dei setup della PaaS:
  - Passi tecnici di configurazione dipendenti dalla PaaS
  - Tipicamente richiedono di descrivere, in un file e/o via interfaccia web, il runtime `python`, i settaggi dei folder e la procedura di attivazione della web application
- Step 5: deploy
  - Trasferimento (via `GIT`) dei files sulla PaaS
  - Ricostruzione dell'ambiente attraverso il file di dipendenze
  - Allineamento/completamento dei setup della PaaS

# Note sul deploy

- Step 6: aggiornamento ulteriore di settaggi ed ambiente:
  - Tipicamente, solo a questo punto si hanno i dati per poter aggiornare `settings.py` perchè Django abiliti correttamente gli accessi al sito (variabili `ALLOWED_HOSTS` e `SET_CSRF_TRUSTED_ORIGINS` )
  - Si creano, laddove necessarie, le variabili di ambiente riferite da `settings.py` sulla PaaS
  - Si crea, se necessario, le utenze ed i dati sul DB di produzione

**Finalmente la app e' deployata!**

# Note sul deploy: recap

- Ambiente di produzione ben diverso da quello di sviluppo
- Infrastrutture di produzione: IaaS e PaaS
- Anche su quelle piu' semplici (PaaS):
  - Procedura non banale
  - Richiede interventi specifici della PaaS prescelta
  - Richiede settaggi anche sui files in Django

# Server: recap

- Inquadramento dello sviluppo lato server:
  - Overview dei linguaggi piu' diffusi
  - Overview degli approcci e degli strumenti di supporto (fwks)
  - Focus su Django
- Sviluppo con Django
  - Recap prerequisiti (focus su OOP Python)
  - Metodologia incrementale basata su esempio

Tutti gli argomenti sono passibili di approfondimento, ma si e' mirato ad una copertura ampia nel contesto dei tempi dati