Web Application Architecture

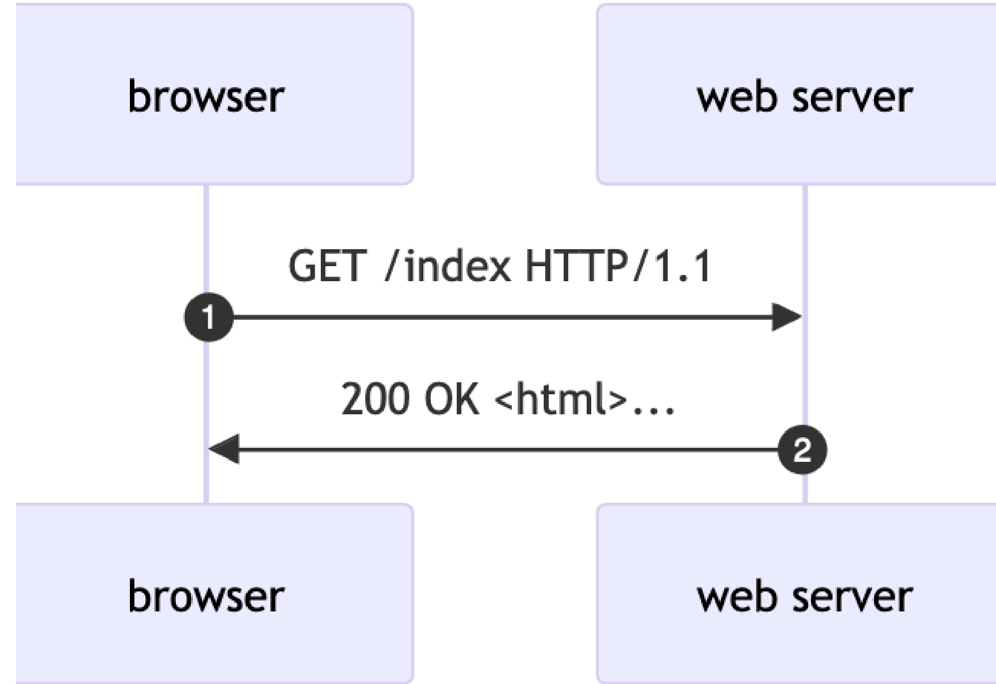
SECT.07.2026

- How does a web application work?
- What *architectures* are in common use in web apps?

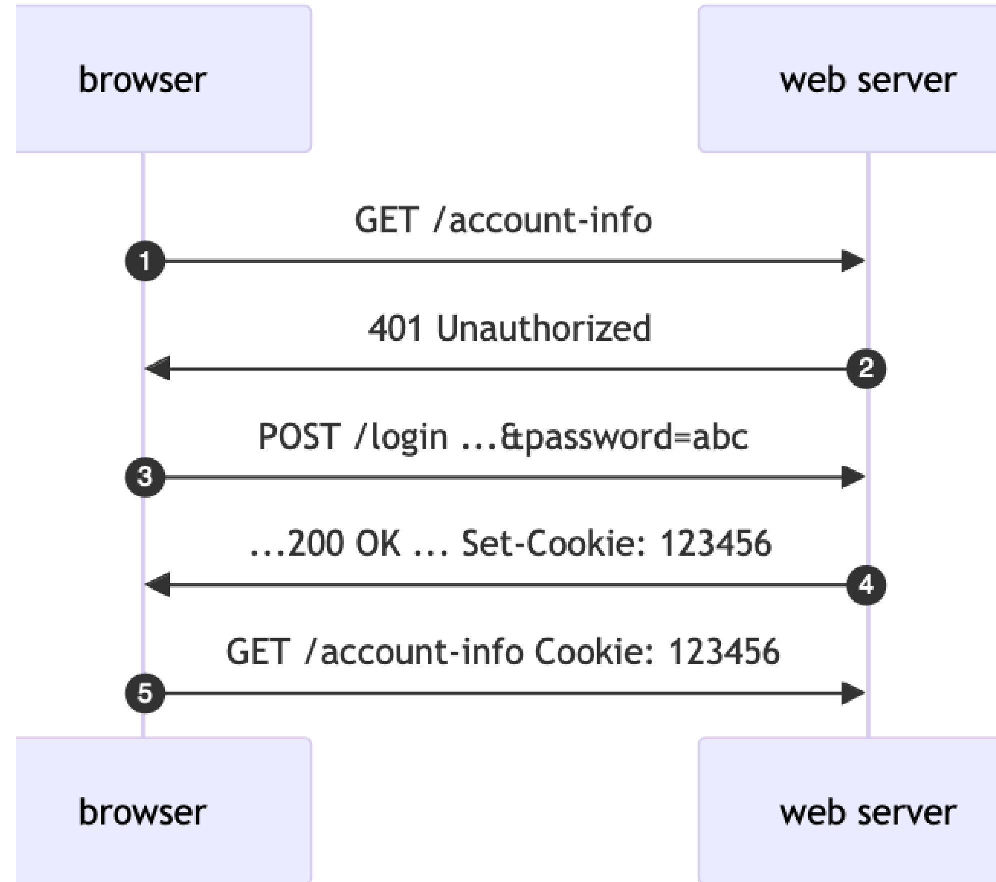
Reminder: HTTP is stateless

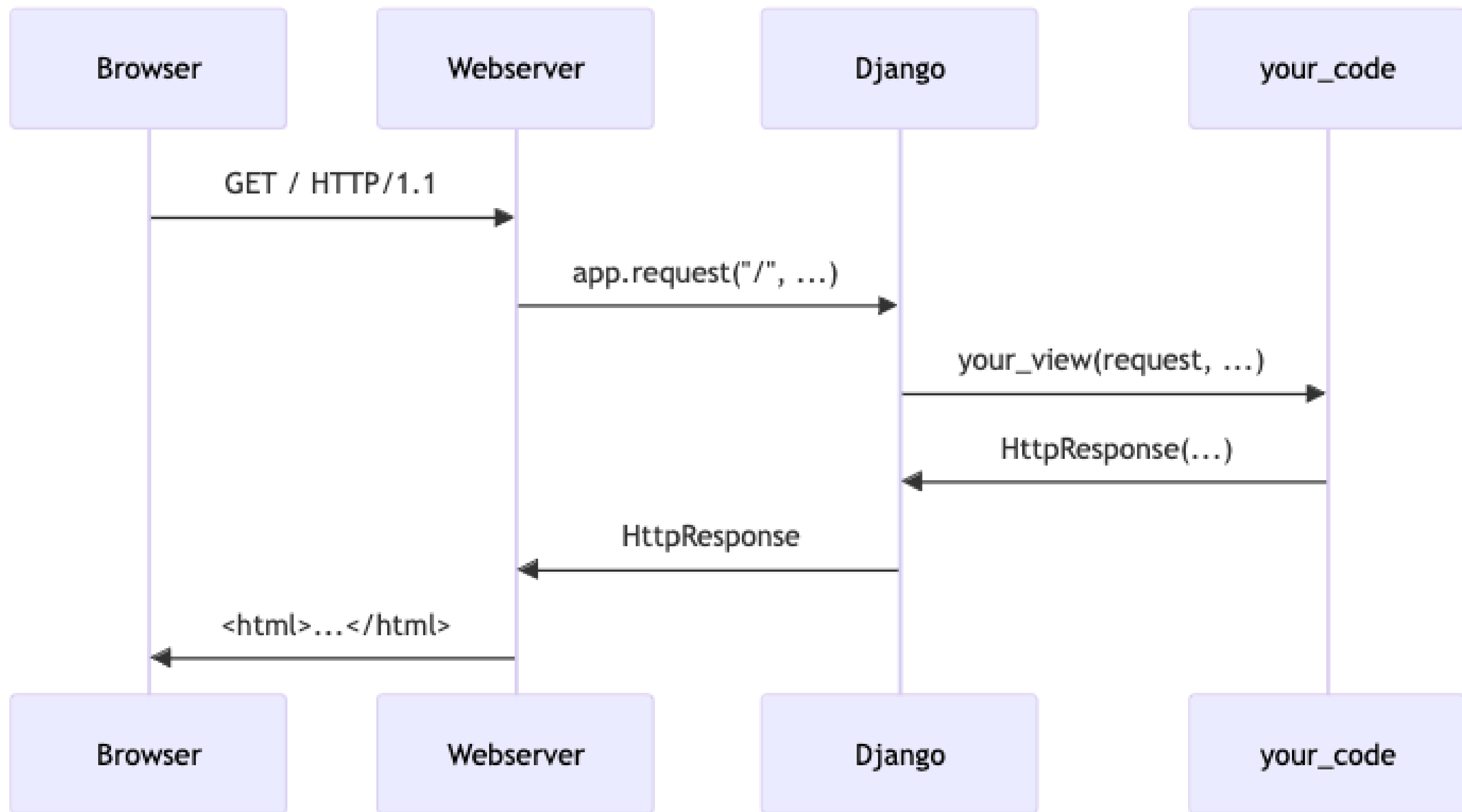
1 request -> 1 "run" of the program

We re-establish all basic facts on each & every request: "who are you, what browser, what protocol, what language, ..."



Regaining state with cookies




Browser
Webserver
Django
your_code
GET / HTTP/1.1
app.request("/", ...)
your_view(request, ...)
HttpResponse(...)
HttpResponse
<html>...</html>
Browser
Webserver
Django
your_code

1 Request : 1 Function

```
f(
  url: str,
  method: "GET" | "POST" | "PUT" | "DELETE",
  params: dict[str, str],
  headers: dict[str, str]) →
    tuple[status_code: int, headers: dict[str, str], content: str]
```

```
f("/cat.jpg", "GET" ...) →
  ("200 OK", {"content-type": "image/jpeg"}, Image("cat.jpg))
```

```
f("/user/login/", "POST", ...) →
  ("200 OK", {"content-type": "text/html"}, ...)
```

In Practice

`WSGI: Web Server Gateway Interface`

```
def application(environ, start_response):
  # params in environ dict
  # start_response is a function
  response_headers = [
        ('Content-Type', 'text/plain'),
        ('Content-Length', str(len( ... )))
    ]
  start_response('200 OK', response_headers)
  return ["<html> ... "]
```

Flask

```
from flask import Flask

app = Flask(__name__)

@app.route("/hello")
def hello_world():
    return "<p>Hello, World!</p>"
```

In Django: a view function or class associated with a URL via urls.py

What else do we need?

Ability to persist data on backend between calls

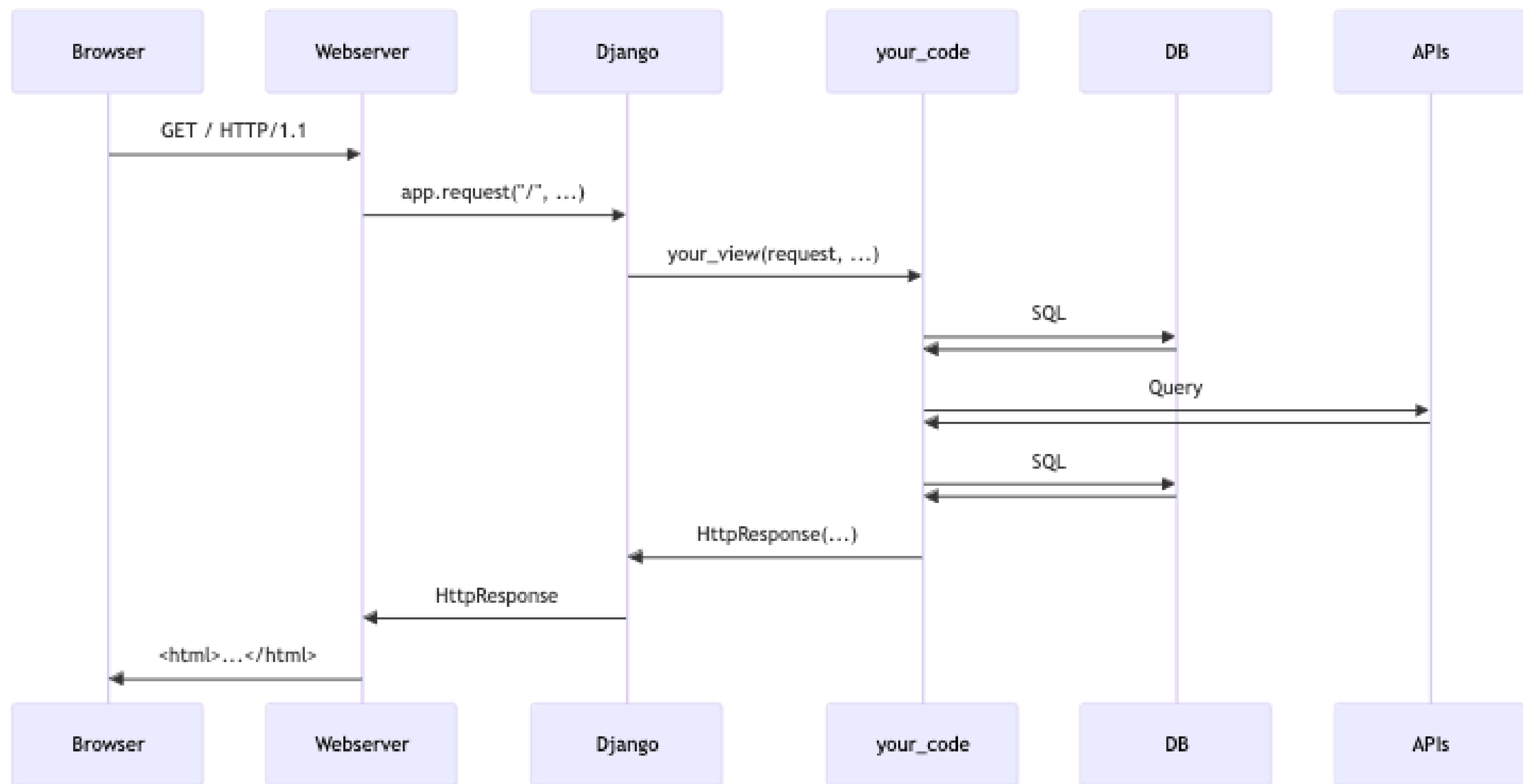
Restoring State

We can’t keep state, but we can communicate with program(s) that can.

database - Manage core data of the application.

cache - Often a key-value store like Redis or memcached. Can be used for traditional caching or as a simplistic DB. (A persistent Python dictionary.)

queue - Can help manage long-running tasks behind the scenes while adhering to sub-second expectations.


Browser
Webserver
Django
your_code
DB
APIs
GET / HTTP/1.1
app.request("/", ...)
your_view(request, ...)
SQL
Query
SQL
HttpResponse(...)
HttpResponse
<html>...</html>
Browser
Webserver
Django
your_code
DB
APIs

Timing

Page load: from pressing enter to full page display

1-3s (but typically will include images & other resources)

Initial HTTP Response

100-300ms

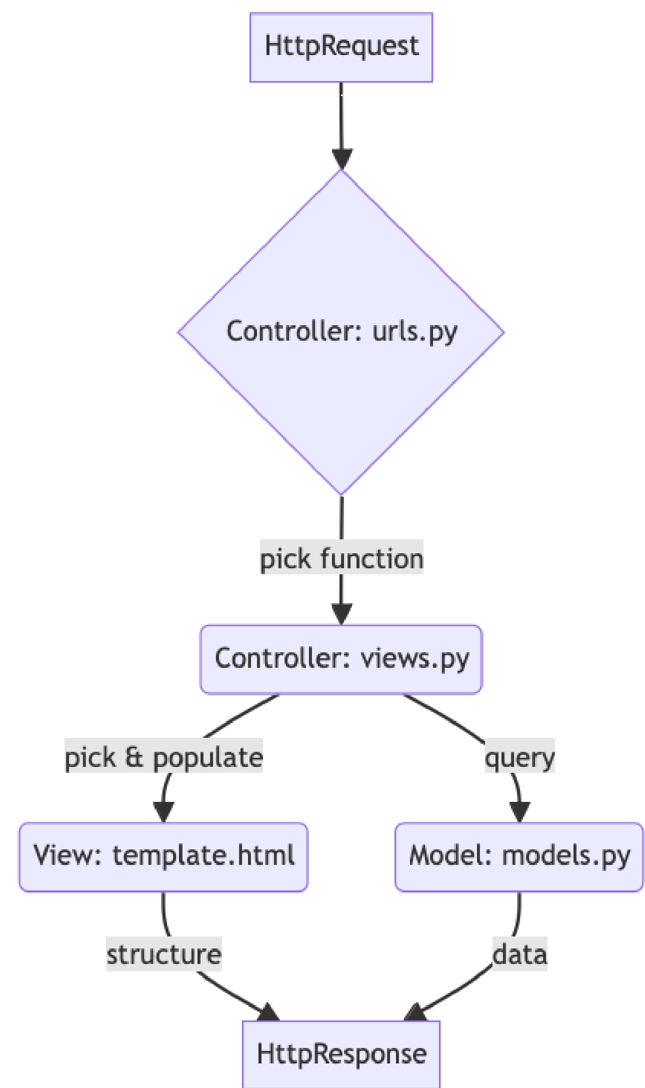
Interaction Response

50-100ms - will need to happen *client side*, can't rely on a full server request

Model-View-Controller

Object-oriented approach to division of labor within a web application.

Django uses a modified version of this: Model-Template-View.



Model

A layer that provides interface between code & the data layer.

Functions and/or classes that allow interacting with data:

```
# model.py
create_user(username: str, password: str)
update_user_password(password: str)
save_bookmark(user_id: int, url: str)
delete_bookmark(user_id: int, url: str)
get_bookmarks(user_id: int) → list[str]
```

Django: ORM (Object-Relational-Mapper) Models

Django Models

```
class Bookmark(models.Model):
    user = ForeignKey(User)
    title = CharField(max_length=200)
    url = URLField()
    description = TextField()
    created_at = DateTimeField( ... )
     ...
```

Using Models (from Django Views)

```
Bookmark.objects.all()

Bookmark.objects.create( ... )

Bookmark.objects.filter(
  tags__includes=["#recipe"], user__name="Stu",
)

Bookmark.objects.trending_today() # user-written
```

Can drop down to raw SQL when needed, but provides a more Pythonic interface & hides leaky abstraction of different SQL implementations.

View

User Interface / Presentation of Data

Django: HTML Template

Django Templates

```
{% extends 'base.html' %}

{% block body %]

  <h1>Welcome {{ request.user }}</h1>
  {% for bookmark in bookmarks %}
    {% include "_bookmark_div.html" %}
  {% endfor %}

{% endblock %}
```

Typical approach for those new to Django: Write HTML with placeholders first, then convert to templates.

```
<!-- Step 3: broken into files -->
{% extends base.html %}

{% block body %}
  <h1>Welcome {{ user }}!</h1>
    <ul>
    {% for bookmark in user.bookmarks %}
      <li><a href="{{ bookmark.url }}"{{ bookmark.name }}</a></li>
    {% endfor %}
    </ul>
{% endblock %}
```

Controller

Interfaces between model & view.

Based on URL, user input, etc. selects an appropriate **view** and populates with data from **models** and populates with data from **models**.

Django: views.py (sorry) & urls.py

Django URLs

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

urlpatterns = [
    path('', views.bookmark_list, name='bookmark_list'),
    path('add/', views.bookmark_create),
    path('<int:pk>/', views.bookmark_detail),
    path('<int:pk>/update/', views.bookmark_update),
    path('<int:pk>/delete/', views.bookmark_delete),
]
```

Django Views

```
def bookmark_list(request):
  if not request.user.is_authenticated():
    raise Http403()
  bookmarks = Bookmark.get_for_user(request.user)
  trending = Bookmark.get_trending()

  return render(
    "bookmark.html",
    {"bookmarks": bookmarks,
     "trending": trending,
     "username": request.user.username}
  )
```

Django Specifics

Apps (modules)

Encourages code organization. Function somewhat like **plugin architecture**.

Forms

Helper classes/functions that map between models & HTML.

Auth

Pre-existing models & views for authentication.

Admin

Pre-existing views & templates for managing data in database.

CRUD: Create, Read, Update, Delete Views

Migrations

Help manage changes to models.

Non-Django Libraries

`Flask` & `FastAPI` can support MVC architecture, but are primarily the **controller**.

Model Layers

Other ORMs: SQLAlchemy, TortoiseORM, Peewee

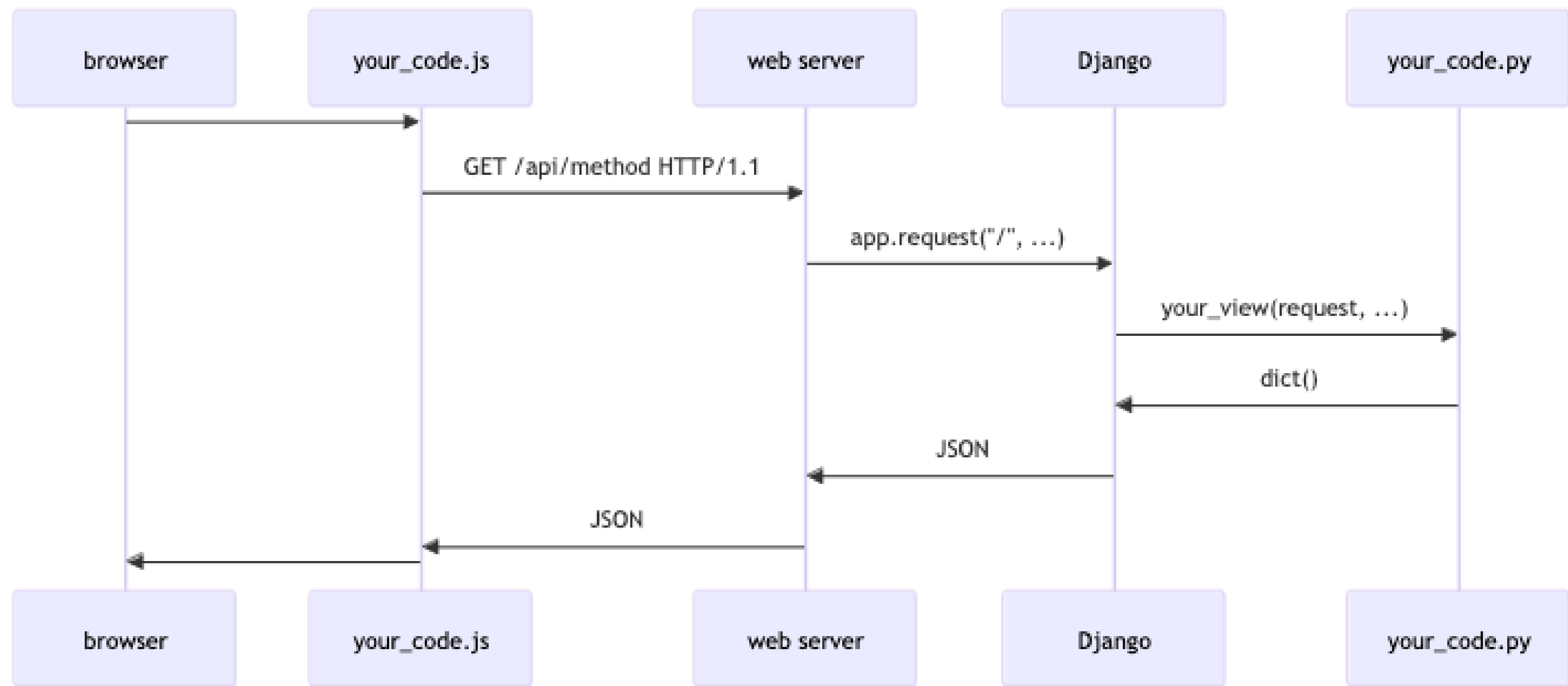
SQL directly, `prql`, MongoDB, etc.

View Layers

`jinja` - The canonical Python HTML template library, similar to Django.

(Some prefer it to Django's, easy to swap out this layer.)

Or: send JSON to browser, and render content with JavaScript. JavaScript is view behavior.


browser
your_code.js
web server
Django
your_code.py
GET /api/method HTTP/1.1
app.request("/", ...)
your_view(request, ...)
dict()
JSON
JSON
browser
your_code.js
web server
Django
your_code.py

Stateful Design

Associate user with a "state" object stored in backend.

Rely very heavily on framework, mostly ignore HTTP.

Often breaks back button & other browser features.

Common in enterprise apps. You’ll recognize it when an application doesn’t update the URL at all as you use it and/or breaks back button.

Full Application