# Django for APIs

- Django's "batteries-included" approach masks much of the underlying complexity, allowing for rapid and secure development
- **API** (Application Programming Interface): describe how two computers communicate directly with one another
- **web APIs** dominant architectural pattern: **REST** (REpresentational State Transfer)
- **API-first** approach of formally separating back-end from front-end

# Chapter 1: Initial Set Up

- *$ python -m pip install djangorestframework~=3.13.0*
- *$ pip freeze*: outputs current VE's content
- Use below command to keep track of installed packages & lets other developers recreate VE on different computers: *$ pip freeze > requirements.txt*

# Chapter 2: Web APIs

- **World Wide Web**
    - Internet: system of interconnected computer networks since at least the 1960
    - By the 1980s, many research institutes used internet to share data
    - In 1989 when a research scientist at CERN, **Tim Berners-Lee**, invented HTTP (Hypertext Transfer Protocol) the first standard, universal way to share docs over internet
- **URLs**
    - Uniform Resource Locator: resrc addr on internet
    - Web communication occurs via HTTP, known more formally as HTTP req & HTTP resp
    - Url parts:
        - **scheme:** like https, tells browser how to access resrc at the location
        - **hostname**: like www.google.com
        - **optional path**: /about/
    - When user types https://www.google.com into browser:
        - First browser needs to find the desired server, uses DNS to translate domain name into an IP
        - After that it needs a way to set up a consistent connection with server, via Transmission Control Protocol (TCP) which provides reliable, ordered, and error-checked delivery of bytes between two apps
        - Three-way handshake:
            1. client sends SYN asking to establish connection
            2. server responds with SYN-ACK acknowledging req and passing connection parameter
            3. client sends ACK back to server confirming connection
- Every webpage contains both addr (URL) as well as a list of approved actions known as **HTTP verbs:** CRUD: **POST, GET, UPDATE, DELETE**
- **endpoint** contains data, typically in the JSON format, & list of available actions

- **collection:** type of endpoint which returns multiple data resrc
- **HTTP**
    - request-response protocol between two comps that have an existing TCP connection
    - Every HTTP msg has **status line**, **headers**, **optional body data**:

      Response/request line
      Headers...
      (optional) Body
    - **request line** specifies HTTP method to use (GET), path (/), specific version of HTTP to use (HTTP/1.1): *GET / HTTP/1.1*
    - HTTP headers: Host is domain name and Accept_Language is lang to use: *Host: google.com*     *Accept_Language: en-US*
    - **response line:** specifies that we are using HTTP/1.1, status code 200 OK indicates: *HTTP/1.1 200 OK*
- **Status Codes**
    - **2xx Success**: action requested by client was received, understood, accepted
    - **3xx Redirection**: requested URL has moved
    - **4xx Client Error**: there was an error, typically a bad URL req by client
    - **5xx Server Error**: server failed to resolve req
- **REST**
    - REpresentational State Transfer: architecture first proposed in 2000 by Roy Fielding, approach to building APIs on top of HTTP protocol
    - Every RESTful API:
        - stateless
        - supports common HTTP verbs (GET, POST, PUT, DELETE, etc.)
        - returns data in either the JSON or XML format
- Web API is a collection of endpoints that expose certain parts of underlying DB, As developers we control URLs for each endpoint, what underlying data is available, what actions are possible via HTTP verbs, By using HTTP headers we can set various levels of authentication & permission

# Chapter 3: Library Website

- ***views.py*** controls how DB model content is displayed

# Chapter 4: Library API

- Add ***"rest_framework"*** to ***INSTALLED_APPS*** in **settings.py**
- In traditional Django views used to customize what data to send to temps, Django REST Framework views similar except end result is serialized data in **JSON** format, not the content for a web page
- use ***ListAPIView*** from rest_framework to create a read-only endpoint for all model instances
- **serializer** translates complex data like querysets & model instances into typically JSON, **deserialize** data is the same process in reverse, whereby JSON data is validated & then transformed into a dictionary

- Classes that extend ***serializers.ModelSerializer***, need a ***Meta*** class that needs ***model*** & ***fields***
- Django REST Framework provides several additional helper classes that extend Django's existing test framework like ***APIClient***, extension of Django's default Client to test retrieving API data from DB

## Deployment

# Chapter 5: Todo API

- **Single Page Apps (SPAs):** required for mobile apps & dominant pattern for web apps that want to take advantage of front-end frameworks, allows for using testing & build tools suitable to the task at hand since building, testing, deploying a Django project is quite different than doing the same for a JS one like React.
- forced separation removes the risk of coupling; not possible for front-end changes to break the back-end.
- The main risk of separating the back-end and the front-end is that it requires domain knowledge in both areas.
- **First step** for any **Django API**: install Django & Django REST Framework on top of it
- If update models in two different apps and then run ***python manage.py makemigrations*** the resulting single migration file would contain data on both apps, makes debugging harder, keep migrations as small as possible
- Configure Django REST Framework specific settings use ***REST_FRAMEWORK***:
    - *$ REST_FRAMEWORK = {*
        - *$ "DEFAULT_PERMISSION_CLASSES": [*
        - *$ "rest_framework.permissions.AllowAny",*
        - *$ ],*
    - *$ }*
- default Django REST Framework settings not appropriate for production
- unnecessary to create a separate apis app for API-first by design project
- **id** vs **pk**: both refer to a field automatically added to Django models by the ORM, **id** is a built-in function from Python standard lib, **pk** comes from Django itself, Generic class-based views like DetailView in Django expect to be passed a param named pk while on model fields it is often common to simply refer to id
- API endpoint refers to URL used to make req, collection is an endpoint with multiple items while resource has a single item, terms endpoint & resource often used interchangeably by developers but they mean different things
- **CORS:** Cross-Origin Resource Sharing refers to the fact that whenever a client interacts with an API hosted on a different domain there are potential security issues, CORS requires the web server to include specific HTTP headers that allow for the client to determine if and when cross-domain requests should be allowed, Because we are using **SPA** architecture the front-end will be on a different local port during development and a completely different domain once deployed! So use middleware that will automatically include the appropriate HTTP headers based on our settings:

- *$ python -m pip install django-cors-headers*, do below tasks:
- add ***corsheaders*** to the ***INSTALLED_APPS***
- add ***CorsMiddleware*** above ***CommonMiddleWare*** in ***MIDDLEWARE***
- create a ***CORS_ALLOWED_ORIGINS*** config at the bottom of the file:
    - *$ CORS_ALLOWED_ORIGINS = (*
        - *"http://localhost:3000",*
        - 
        - *"http://localhost:8000",*
    - *$ )*
- Django comes with robust CSRF protection that should be added to forms in any Django template, but with a dedicated React front-end setup this protection isn't inherently available. Fortunately, we can allow specific cross-domain requests from our frontend by setting CSRF_TRUSTED_ORIGINS: *$ CSRF_TRUSTED_ORIGINS = ["localhost:3000"]*

**Deployment**

# Chapter 6: Blog API

- Customize admin.py to display new custom user model and create forms.py that sets CustomUser to be used when creating/changing users
- extend ***UserCreationForm*** & ***UserChangeForm*** which are used for creating/updating user
- Django REST Framework takes care of transforming DB models into a RESTful API, three main steps to this process:
    - **urls.py** file for the URL routes
    - **serializers.py** file to transform the data into JSON
    - **views.py** file to apply logic to each API endpoint
- By default Django REST Framework is configured to AllowAny to enable ease of us in local development however this is far from secure
- good practice to always version your APIs since when you make a large change there may be some lag time before various consumers of the API can also update. That way you can support a v1 of an API for a period of time while also launching a new, updated v2 and avoid breaking other apps that rely on your API back-end.
- ***ListAPIView*** creates a read-only endpoint collection, essentially a list of all model instances
- ***RetrieveAPIView*** for a read-only single endpoint, analogous to detail view in traditional Django
- ***ListCreateAPIView***, read-write endpoint, allows POST requests
- ***RetrieveUpdateDestroyAPIView*** for read, update, or delete
- **CORS:** Since it is likely our API will be consumed on another domain we should configure CORS and set which domains will have access.

# Chapter 7: Permissions

- DRF has several out-of-the-box permissions settings that we can use to secure the API, These can be applied at a project-level, a view-level, or at any individual model level
- uilt-in project-level permissions settings:
    - **AllowAny**: any user, authenticated or not, has full access
    - **IsAuthenticated**: only authenticated, registered users have access
    - **IsAdminUser**: only admins/superusers have access
    - **IsAuthenticatedOrReadOnly**: unauthorized users can view any page, but only authenticated users have write, edit, or delete privileges
- Use ***permissions*** of rest_framework in view classes and set permission_classes field to limit the access to that view api
- DRF relies on ***BasePermission*** class from which all other permission classes inherit
- For a custom permission class you can override one or both of these methods. ***has_permission*** works on list views while detail views execute both: first ***has_permission*** and then, if that passes, ***has_object_permission***, strongly advised to always set both methods explicitly because each defaults to True, meaning they will allow access implicitly unless set explicitly.

# Chapter 8: User Authentication

- HTTP is a stateless protocol, no built-in way to remember if user is authenticated from one req to the next. Each time a user requests a restricted resource it must verify itself
- Solution: pass along a **unique identifier** with each HTTP req, no universally agreed-upon approach for the form of this identifier and it can take multiple forms
- DRF ships with four different built-in authentication options:
    - **Basic:** basic authentication should only be used via HTTPS
        1. Client makes an HTTP req
        2. Server responds with an HTTP resp containing a 401 (Unauthorized) status code and WWW-Authenticate HTTP header with details on how to authorize
        3. Client sends credentials back via the Authorization HTTP header
        4. Server checks credentials and responds with either 200 OK or 403 Forbidden status code
        5. Once approved, the client sends all future reqs with the Authorization HTTP header credentials
           Downside of this approach:
            1. on every single req, server must look up and verify username & password
            2. user credentials are being passed in clear text
    - **Session**: more secure since user credentials are only sent once
        1. user enters their login credentials
        2. server verifies the credentials are correct, generates a session obj stored in DB
        3. server sends the client a session ID —not the session object itself— stored as a cookie on the browser

4. On all future reqs the session ID is included as an HTTP header and, if verified DB, the req proceeds
5. Once a user logs out of app, the session ID is destroyed by both client & server
6. if the user later logs in again, a new session ID is generated and stored as a cookie on the client
   Downside of this approach:
   1. session ID is only valid within the browser where log in was performed; it will not work across multiple domains. This is an obvious problem when an API needs to support multiple front-ends such as a website and a mobile app
   2. session obj must be kept up-to-date which can be challenging in large sites with multiple servers
   3. cookie is sent out for every single req, even those that don't require authentication, which is inefficient.
      generally not advised to use a session-based auth scheme for any API that will have multiple front-ends

- **Token:**
  - stateless, once a client sends the initial user credentials to server, unique token is generated and then stored by the client as either a cookie or in local storage. This token is then passed in the header of each incoming HTTP req and the server uses it to verify that a user is authenticated. The server itself does not keep a record of the user, just whether a token is valid or not, best practice is to store tokens in a cookie with the **httpOnly** & **Secure** cookie flags
  - Since tokens are stored on the client, they can be shared amongst multiple front-ends: the same token can represent a user on the website and the same user on a mobile app
  - A potential downside is that tokens can grow quite large. A token contains all user information, Since the token is sent on every req, managing its size can become a performance issue.
  - JSON Web Tokens (JWTs) are a newer form of token containing cryptographically signed JSON data
- **default**, and many more third-party packages that offer additional features like JSON Web Tokens (JWTs)

- **Implementing Token Auth:**
  - Add below code to ***REST_FRAMEWORK***:
    *$ "DEFAULT_AUTHENTICATION_CLASSES": [*
    *$ "rest_framework.authentication.SessionAuthentication",*
    *$ "rest_framework.authentication.TokenAuthentication", # new*
    *$ ],*
  - Add ***rest_framework.authtoken*** to ***INSTALLED_APPS***
  - Use ***dj-rest-auth*** in combination with ***django-allauth***

- **User Registration**
    - Traditional Django does not ship with built-in views or URLs for user registration and neither does Django REST Framework
    - A popular approach is to use the third-party package **django-allauth** which comes with user registration as well as a number of additional features to the Django auth system such as social authentication via Facebook, Google, etc: *$ python -m pip install django-allauth*
      Add below apps for allauth: django.contrib.sites, (3rd party apss) allauth, allauth.account, allauth.socialaccount, dj_rest_auth.registration
    - If we add dj_rest_auth.registration from the **dj-rest-auth** package then we have user registration endpoints too!
    - django-allauth needs to be added to the ***TEMPLATES*** configuration after existing context processors as well as setting the ***EMAIL_BACKEND*** to console and adding a ***SITE_ID*** of 1.
    - The email back-end config is needed since by default an email will be sent when a new user is registered, asking them to confirm their account. Rather than also set up an email server, we will output the emails to the console with the ***console.EmailBackend*** setting.
    - ***SITE_ID*** is part of the built-in Django "**sites**" framework, which is a way to host multiple websites from the same Django project. We only have one site we are working on here but django-allauth uses the sites framework, so we must specify a default setting.
    - In our front-end framework, we would need to capture and store this token. Traditionally this happens on the client, either in **localStorage**/**cookie**, and then all future requests include the token in the header as a way to authenticate the user. Note that there are additional security concerns on this topic so you should take care to implement the best practices of your front-end framework of choice.

# Chapter 9: Viewsets and Routers

- **Viewsets** & **routers** are tools within DRF that can speed-up API development, additional layer of abstraction on top of views and URLs
- A single viewset can replace multiple related views, a router can automatically generate URLs for the developer
- By using ***get_user_model*** we ensure that we are referring to the correct user model, whether it is the default User or a custom user model like ***CustomUser*** in our case.
- **Viewsets:**
    - A viewset is a way to combine the logic for multiple related views into a single class, one viewset can replace multiple views
    - Then we are using ***ModelViewSet*** which provides both a list view and a detail view for us.
- **Routers**
    - **Routers** work directly with **viewsets** to **automatically** generate URL patterns for us: DRF has two default routers: ***SimpleRouter*** & ***DefaultRouter***

# Chapter 10: Schemas and Documentation

- **Schema**: machine-readable doc that outlines all available API endpoints, URLs, and the HTTP verbs (GET, POST, PUT, DELETE, etc.)
- The **OpenAPI** specification is the current default way to document an API. It describes common rules around format for available endpoints, inputs, authentication methods, contact information, and more.
- **drf-spectacular** is the recommended third-party package for generating an OpenAPI 3 schema for DRF: *$ python -m pip install drf-spectacular*
- Then register **drf-spectacular** within the ***REST_FRAMEWORK*** section of the django_project/settings.py file, also add ***SPECTACULAR_SETTINGS***
- *$ python manage.py spectacular --file schema.yml*
- **Dynamic Schema: s**erve the schema directly from our API as a URL route, import ***SpectacularAPIView*** and then add a new URL path at api/schema/ to display it.
- **Documentation:** API doc tools supported by drf-spectacular: **Redoc** & **SwaggerUI**

# Chapter 11: Production Deployment

- **Environment Variables**
    - Env vars can be loaded into a codebase at runtime yet not stored in the source code, ideal way to toggle between local & production settings, a good place to store sensitive info that should not be present in source control
    - Use **environs** package: *$ python -m pip install 'environs[django]'*
    - Add *.env* file
    - Django's default *settings.py*: local production settings
    - ***DEBUG = True:*** generates a very detailed error page and stack trace
    - DEBUG = True for local development yet False for production
    - ***SECRET_KEY*** a random 50 char str generated each time startproject is run
    - Generate new ***SECRET_KEY***: *$ python -c "import secrets; print(secrets.token_urlsafe())"*
    - ***ALLOWED_HOSTS*** represents the host/domain names our project can serve.
    - Change default of ***DATABASES*** to *"default": env.dj_db_url("DATABASE_URL")* to read urls from .env
    - For static files add *WhiteNoise* package:
        - whitenoise above django.contrib.staticfiles in ***INSTALLED_APPS***
        - WhiteNoiseMiddleware above ***CommonMiddleware***
        - ***STATICFILES_STORAGE*** configuration pointing to WhiteNoise
    - run *$ python manage.py collectstatic* so that all static dirs & files are compiled into one location for deployment purposes
    - two final packages for a proper production env:
        - **Psycopg** is a DB adapter that lets Python apps talk to PostgreSQL DBs: *$ python -m pip install psycopg2*
        - **Gunicorn** is a production web server and must be installed as well to replace the current Django web server which is only suitable for local development: *$ python -m pip install gunicorn*

- *requirements.txt* needed which lists all the packages installed in our local virtual env: *$ python -m pip freeze > requirements.txt*
- **Deployment checklist:**
    - dd environment variables via environs[django]
    - set ***DEBUG*** to False
    - set ***ALLOWED_HOSTS***
    - use environment variable for ***SECRET_KEY***
    - update ***DATABASES*** to use SQLite locally and PostgreSQL in production configure static files and install *whitenoise*
    - install *gunicorn* for a production web server
    - create *requirements.txt*
    - create *Procfile* for Heroku
    - create *runtime.txt* to set the Python version on Heroku
- **Advanced Topics in DRF:**
    - **Pagination**: helpful way to control how data is displayed on individual API endpoints
    - **Filtering** also becomes necessary in many projects especially in conjunction with the excellent django-filter library
    - **Throttling** necessary on APIs as a more advanced form of permissions
    - **caching** of the API for performance reasons