



جنگو

برای

ها API

ساخت web api ها با پایتون و جنگو

ترجمه آزاد کتاب

ویلیام . اس وینسنت

مقدمه

اینترنت بوسیله RESTful API ها قدرت گرفته. حتی در پشت ساده‌ترین تعاملات بین چند کامپیوتر.

در یک تعریف رسمی Application Programming Interface (API) روشی برای تعریف چگونگی تعامل و ارتباط مستقیم بین یک رایانه با دیگر است. و در حالیکه روش های مختلفی برای ساخت یک API وجود دارد که اجازه انتقال داده ها را در سطح وب میدهد با این حال وب به شکل قاطعی بر الگوی RESTful (REpresentational State Transfer) ساخت یافته است.

در این کتاب، نحوه ساخت چندین API وب RESTful با پیچیدگی فراوانی از ابتدا با استفاده از Django و REST Framework توضیح داده شده است، یکی از محبوب‌ترین و قابل تنظیم‌ترین راه‌ها برای ساخت API‌های وب، که توسط بسیاری از بزرگترین شرکت‌های فناوری در جهان استفاده می‌شود، آموزش داده خواهد شد. از جمله اینستاگرام، موزیلا، پینترست، و بیت باکت است. این رویکرد همچنین برای مبتدیان کاملاً مناسب است، زیرا رویکرد «batteries-included» جنگو بسیاری از پیچیدگی‌های اساسی و خطرات امنیتی موجود در ایجاد هر API وب را پنهان می‌کند.

پیش‌نیازها

اگر در توسعه وب با Django کاملاً تازه کار هستید، توصیه می‌کنم ابتدا کتاب قبلی من Django for Beginners را بخوانید. چندین فصل اول به صورت آنلاین رایگان در دسترس هستند و شامل راه اندازی مناسب، یک برنامه Hello World و یک برنامه Pages است. نسخه کامل کتاب عمیق‌تر می‌شود و یک وب‌سایت بلاگ را با فرم‌ها و حساب‌های کاربری و همچنین یک سایت روزنامه آماده تولید را پوشش می‌دهد که دارای یک مدل کاربر سفارشی، جریان احرار، هویت کامل کاربر، ایمیل‌ها، مجوزها، استقرار، متغیرهای محیطی و موارد دیگر است.

این پیشینه در جنگو سنتی مهم است زیرا چارچوب REST عمداً بسیاری از قراردادهای جنگو را تقلید می‌کند.

همچنین توصیه می‌شود که خوانندگان دانش اولیه پایتون را داشته باشند. تسلط واقعی بر پایتون سال‌ها طول می‌کشد، اما فقط با کمی دانش، می‌توانید درست در آن شیرجه بزنید و شروع به ساختن کنید.

چرا API ها

جنگو برای اولین بار در سال 2005 منتشر شد و در آن زمان بیشتر وب سایت‌ها از یک پایگاه که یک پارچه بزرگ تشکیل شده بودند. «بک‌اند» شامل مدل‌های پایگاه داده، URL‌ها و نماهایی بود که با قالب‌های HTML، CSS و جاوا اسکریپت که طرح انیمیشنی هر صفحه وب را که «front-end» کنترل می‌کرند، تعامل داشتند.

با این حال، در سال‌های اخیر، رویکرد اول "API" به عنوان الگوی غالب در توسعه وب ظاهر شد. این رویکرد شامل جداسازی رسمی قسمت بک‌اند از قسمت فرانت‌اند است. این بدان معناست که جنگو به جای یک چارچوب وب سایت، به یک پایگاه داده و API قادرمند تبدیل می‌شود. امروزه جنگو بیشتر به عنوان یک API پشتیبان به جای یک راه حل کامل وب سایت یکپارچه در شرکت‌های بزرگ استفاده می‌شود!

یک سوال واضح در این مرحله این است: "چرا زحمت بکشیم؟" جنگو سنتی به تنهایی کار می کند و تبدیل شدن یک سایت جنگو به یک وب API کار اضافی به نظر می رسد. به علاوه، به عنوان یک توسعه دهنده، باید یک front-end اختصاصی به زبان برنامه نویسی دیگری هم بنویسید. این رویکرد تقسیم خدمات به اجزای مختلف، به هر حال، به طور گسترده به عنوان [معماری سرویس گرا](#) شناخته می شود.

با این حال، به نظر می رسد که مزایای متعددی برای جداسازی قسمت فرانت‌اند از قسمت بکاند وجود دارد. اولاً، مسلماً « مقاوم‌تر در آینده » است زیرا یک API بکاند را می‌توان توسط هر فرانت‌اند جاوا اسکریپت استفاده کرد. با توجه به رشد سریع تغییر در کتابخانه‌های فرانت‌اند - [React](#) تنها در سال 2013 و [Vue](#) در سال 2014 منتشر شد! این بسیار ارزشمند است وقتی که فریم‌ورک‌های فرانت‌اند فعلی در سال‌های آینده با فریم‌ورک‌های جدیدتر جایگزین شوند، اما API بکاند می‌تواند ثابت بماند و نیازی به بازنویسی عدمه ندارند.

دوم، یک API می‌تواند از چندین فرانت‌اند نوشته شده در زبان‌ها و فریم‌ورک‌های مختلف پشتیبانی کند. در نظر بگیرید که جاوا اسکریپت برای فرانت‌اندهای وب استفاده می‌شود، در حالی که برنامه‌های اندروید به زبان برنامه نویسی جاوا و برنامه‌های iOS به زبان برنامه نویسی Swift نیاز دارند. با رویکرد سنتی یکپارچه، یک وب سایت جنگو نمی‌تواند از این بخش‌های مختلف پشتیبانی کند. اما با یک API داخلی، هر سه می‌توانند با یک پایگاه داده اصلی ارتباط برقرار کنند!

سوم، یک رویکرد API-first می‌تواند هم به صورت داخلی و هم خارجی استفاده شود. زمانی که در سال 2010 در [Quizlet](#) کار می‌کردم، منابع لازم برای توسعه برنامه‌های iOS یا اندروید را نداشتیم. اما ما یک API خارجی در دسترس داشتیم که بیش از 30 توسعه دهنده از آن برای ایجاد برنامه‌های فلش کارت خود با پشتیبانی از پایگاه داده Quizlet استفاده می‌کردند. چندین مورد از این برنامه‌ها بیش از یک میلیون بار دانلود شدند و توسعه دهنگان را غنی تر کرد و همزمان دسترسی به Quizlet را افزایش داد. اکنون یکی از 20 وب سایت برتر در ایالات متحده در طول سال تحصیلی است.

نقطه ضعف اصلی رویکرد API-first این است که نسبت به یک برنامه سنتی جنگو به پیکربندی بیشتری نیاز دارد. با این حال، همانطور که در این کتاب خواهیم دید، کتابخانه فوق العاده فریم‌ورک جنگو REST بسیاری از این پیچیدگی‌ها را حذف می‌کند.

فریم‌ورک جنگو REST

صدها و صدها برنامه شخص ثالث در دسترس هستند که عملکردهای بیشتری را به جنگو اضافه می‌کنند. شما می‌توانید یک لیست کامل و قابل جستجو را در [Django Packages](#) و همچنین یک لیست انتخاب شده را در [awesome-Django repo](#) مشاهده کنید. با این حال، در میان تمام برنامه‌های شخص ثالث، [Django REST Framework](#) مسلماً برنامه‌ای که قاتل جنگو است. کامل، پر از ویژگی‌ها، قابل سفارشی سازی، آزمایش پذیر و مستندات کاملی دارد. همچنین به طور هدفمند بسیاری از قراردادهای سنتی جنگو را تقلید می‌کند که یادگیری آن را بسیار سریعتر می‌کند. و به زبان برنامه نویسی پایتون نوشته شده است، زبانی فوق العاده، محبوب و قابل دسترس برای همگان.

اگر از قبل جنگو را می‌شناسید، یادگیری جنگو REST Framework قدم منطقی بعدی است. با حداقل مقدار کد، می‌تواند هر برنامه جنگو موجود را به یک وب API تبدیل کند.

چرا این کتاب

من این کتاب را نوشتیم زیرا منابع کم و خوبی در دسترس برای توسعه دهنگان تازه وارد جنگو REST Framework وجود دارد. به نظر می‌رسد فرض بر این است که همه از قبل همه چیز را در مورد API‌ها، REST، HTTP و موارد مشابه می‌دانند. سفر خود من هم در یادگیری نحوه ساخت API‌های وب خسته کننده بود... و قبلًاً جنگو را به خوبی می‌شناختم که بتوانم کتابی در مورد آن بنویسم!

این کتاب راهنمای من است که آرزو می‌کردم هنگام شروع کار با Django REST Framework وجود داشته باشد. فصل 1 با معرفی مختصاتی از وب API و پروتکل HTTP آغاز می‌شود. در فصل 2، با ایجاد یک وب سایت کتابخانه ای و سپس افزودن یک API به آن، تفاوت‌های بین جنگو سنتی و چارچوب REST جنگو را بررسی می‌کنیم. سپس در فصل 4-3 یک Todo API می‌سازیم و آن را به یک React front-end متصل می‌کنیم. از همین فرآیند می‌توان برای اتصال هر فرانت‌اند اختصاصی (وب، iOS، اندروید، دسکتاپ یا موارد دیگر) به یک بک‌اند API وب استفاده کرد.

در فصل‌های 9-5 ما یک API و بلاگ تولید محتوا می‌سازیم که شامل عملکرد کامل CRUD است. همچنین مجوزهای عمیق، احراز هویت کاربر، مجموعه‌های نمایشی، روت‌ها، اسناد و موارد دیگر را پوشش می‌دهیم.

همه کدها برای همه فصل‌ها را می‌توانید به [صورت آنلاین در Github](#) در فصل 10 ام پیدا کنید.

نتیجه گیری

درواقع Django and Django REST Framework یک راه قدرتمند و در دسترس برای ساخت API‌های وب است. در پایان این کتاب، شما می‌توانید API‌های وب خود را از ابتدا به درستی با استفاده از بهترین شیوه‌های مدرن بسازید. و می‌توانید هر وب سایت جنگو موجود را با کمترین مقدار کد به یک API وب گسترش دهید.

Web APIs

قبل از اینکه شروع به ساختن وب api خودمان کنیم مهم است که مرور کنیم صفحات وب واقعاً چگونه کار می‌کنند. گذشته از همه اینها یک "وب api" به معنای واقعی کلمه در بالای معماری حال حاضر شبکه جهانی وب قرار می‌گیرد و از مجموعه‌ای از فناوری‌ها از جمله TCP/IP، HTTP و غیره استفاده می‌کند.

در این فصل اصطلاحات اساسی وب API را بررسی خواهیم کرد: endpoints, resources, HTTP verbs, کدهای وضعیت HTTP و REST. حتی اگر با این موارد از قبل آشنایی دارید توصیه می‌کنم که این فصل را به طور کامل بخوانید.

شبکه جهانی وب

اینترنت سیستمی از شبکه‌های کامپیوتری به هم پیوسته است که حداقل از دهه 1960 وجود داشته است. به هر حال استفاده اولیه از اینترنت محدود به تعداد کمی از شبکه‌های مجزا، عمدتاً دولتی، نظامی یا علمی بود که اطلاعات را به صورت الکترونیکی رد و بدل می‌کردند.

در دهه 1980، بسیاری از مؤسسات تحقیقاتی و دانشگاه‌ها از اینترنت برای به اشتراک گذاری داده‌ها استفاده می‌کردند.

در اروپا، بزرگترین گره (Node) اینترنتی در CERN (سازمان اروپایی تحقیقات هسته‌ای) در ژنو سوئیس که بزرگترین آزمایشگاه فیزیک ذرات جهان را اداره می‌کند قرار داشت. این آزمایش‌ها مقادیر عظیمی از داده‌ها را تولید می‌کنند که باید از راه دور با دانشمندان در سراسر جهان به اشتراک گذاشته می‌شدند.

در مقایسه با امروز، استفاده کلی از اینترنت در دهه 1980 بسیار ناچیز بود. اکثر مردم به آن دسترسی نداشتند یا حتی دلیل اهمیت آن را درک نمی‌کردند. تعداد کمی از گره‌های اینترنتی تمام ترافیک را تأمین می‌کردند و رایانه‌هایی که از آن استفاده می‌کردند، عمدتاً در همان شبکه‌های کوچک بودند.

همه اینها در سال 1989 زمانی که یک دانشمند محقق در سرن، Tim Berners-Lee) HTTP را اختراع کرد، تغییر کرد و شبکه جهانی وب مدرن را آغاز کرد. نگرش او این بود که سیستم hypertext موجود، که در آن متن نمایش داده شده بر روی صفحه کامپیوتر حاوی پیوند (هایپرلینک) به اسناد دیگر است، می‌تواند به اینترنت منتقل شود.

اختراع او، پروتکل انتقال هایپرтекست (HTTP)، اولین راه استاندارد و جهانی برای به اشتراک گذاری اسناد از طریق اینترنت بود. این پروتکل این موارد را در قالب صفحات وب ارائه می‌دهد: اسناد گسته با URL پیوندها و منابعی مثل عکس‌ها صوت و ویدیو.

امروزه، زمانی که بیشتر مردم به «اینترنت» فکر می‌کنند، در واقع به شبکه جهانی وب فکر می‌کنند، که در حال حاضر راه اصلی برقراری ارتباط آنلاین میلیاردها نفر و رایانه است.

URLs

یک url (تعیین کننده محل منابع) آدرس یک منبع بر روی شبکه اینترنت می‌باشد. برای مثال صفحه اصلی گوگل در <https://www.google.com> می‌باشد.

وقتی که می خواهید به صفحه اصلی گوگل بروید شما url کامل آن را در مرورگر تایپ می کنید. مرورگر شما از طریق اینترنت یک درخواست می فرستد و یک سرور به درخواست شما برای نمایش دیتاهاي موردنیاز صفحه اصلی Google در مرورگر شما پاسخ می دهد(به زودی آنچه واقعاً اتفاق می افتد را پوشش خواهیم داد).

این الگوی درخواست و پاسخ اساس همه ارتباطات وب است. یک کلاینت (ممکن است یک مرورگر اما می تواند یک برنامه یا هر دستگاه متصل به اینترنت باشد) اطلاعاتی را درخواست می کند و سرور پاسخ می دهد.

از آنجایی که ارتباطات وب از طریق HTTP انجام می شود، اینها بیشتر به عنوان درخواست های HTTP و پاسخ های HTTP شناخته می شوند.

در یک URL داده شده نیز چندین مؤلفه مجزا وجود دارد. برای مثال صفحه اصلی گوگل را که در واقع شده است در نظر بگیرید. قسمت اول یا همان https://www.google.com به نوع درخواست استفاده شده اشاره دارد.

این بخش به مرورگر وب می گوید که چگونه به منابع موجود در آدرس دسترسی داشته باشد.

برای یک وبسایت، این معمولاً http یا https است، اما همچنین می تواند برای فایل ها ftp، برای ایمیل smtp و غیره باشد. بخش بعدی، www.google.com نام میزبان(host) یا نام واقعی سایت است. هر URL حاوی یک نوع درخواست و یک میزبان(host) است.

بسیاری از صفحات وب همچنین شامل یک مسیر اختیاری هستند. اگر به صفحه اصلی پایتون در /https://www.python.org/about بروید و روی پیوند صفحه «about» کلیک کنید، به https://www.python.org هدایت خواهید شد. /about/ یک بخش از آدرس صفحه است.

به طور خلاصه، هر URL مانند /https://python.org/about دارای سه بخش نهانی است:

- یک شمای کلی(نوع درخواست) - https
- نام میزبان - www.python.org
- و یک مسیر (اختیاری) - /about/

مجموعه پروتکل اینترنت

هنگامی که URL یک منبع را می دانیم، مجموعه کاملی از فناوری های دیگر باید به درستی (با هم) کار کنند تا کلاینت را به سرور متصل کرده و یک صفحه وب واقعی را بارگیری کند. این به طور گسترده به عنوان **مجموعه پروتکل اینترنت** نامیده می شود و کتاب های کاملی وجود دارد که فقط برای این موضوع نوشته شده است. با این حال، برای اهداف خود، می توانیم به اصول کلی پایبند باشیم.

هنگامی که کاربر https://www.google.com را در مرورگر وب خود تایپ می کند و دکمه اینتر را می زند، چندین اتفاق می افتد. ابتدا مرورگر باید سرور مورد نظر را در اینترنت پیدا کند. از یک سرویس نام دامنه (DNS) برای ترجمه نام دامنه "google.com" به آدرس IP استفاده می کند، که یک دنباله منحصر به فرد از اعداد است که هر دستگاه متصل را نشان می دهد. اینترنت از نام های دامنه استفاده می کند زیرا به خاطر سپردن نام دامنه برای انسان آسان تر است مانند «google.com» تا یک آدرس IP مانند «172.217.164.68».

پس از اینکه مرورگر آدرس IP یک دامنه معین را داشت، به راهی برای ایجاد یک ارتباط ثابت با سرور مورد نظر نیاز دارد. این از طریق پروتکل کنترل انتقال (TCP) اتفاق می افتد. که مطمئن، مرتب شده و عاری از خطأ انتقال بایت ها را بین دو برنامه ارائه می دهد.

برای ایجاد یک اتصال TCP بین دو کامپیوتر، یک "handshake" سه مسیره بین کلاینت و سرور رخ می دهد:

- کلاینت یک SYN(یک نوع پیام) می فرستد و درخواست برقراری اتصال می کند
 - سرور با تایید درخواست و ارسال یک پارامتر اتصال با یک SYN-ACK پاسخ می دهد
 - کلاینت یک ACK(نشاندهنده دریافت پیام SYN) را برای تایید اتصال به سرور ارسال می کند
- هنگامی که اتصال TCP برقرار شد، دو کامپیوتر می توانند از طریق HTTP ارتباط برقرار کنند.

HTTP Verbs

هر صفحه وب شامل یک آدرس (URL) و همچنین لیستی از اقدامات مورد تایید به نام HTTP verbs است. تاکنون عمدتاً در مورد دریافت یک صفحه وب صحبت کرده ایم، اما امکان ایجاد، ویرایش و حذف محتوا نیز وجود دارد. وبسایت فیسبوک را درنظر بگیرید.

بعد از لایک شما می توانید پست های تایملاینتان را بخوانید یک پست جدید ایجاد کنید یا پست های موجودتان را ویرایش یا حذف کنید. این چهار عمل (خواندن - ساختن - ویرایش کردن - حذف کردن) به صورت عامیانه به عنوان اعمال CRUD شناخته می شوند و اکثر کارهایی که به صورت آنلاین انجام می شود را نمایش می دهد.

پروتکل HTTP حاوی تعدادی متود درخواست است که می توان از آنها در هنگام درخواست اطلاعات از یک سرور استفاده کرد. چهار عمل رایج برای عملکرد CRUD. چهار عمل POST، GET، PUT و DELETE هستند.

CRUD	HTTP Verbs
---	---
Create	POST
Read	GET
Update	PUT
Delete	DELETE

برای ایجاد محتوا از POST، برای خواندن محتوا از GET، برای به روز رسانی آن PUT و برای حذف آن از DELETE استفاده می کنند.

Endpoints

یک وب سایت از صفحات وب با HTML، CSS، تصاویر، جاوا اسکریپت و موارد دیگر تشکیل شده است. اما یک وب API به جای آن اندپوینت دارد که url‌هایی با فهرستی از اعمال موجود (HTTP VERBS) هستند که داده‌ها را نشان می‌دهند (معمولًاً در JSON که رایج‌ترین قالب داده این روزها و پیش‌فرض برای django-rest-framework است).

برای مثال، می‌توانیم اندپوینت API زیر را برای یک وب سایت جدید به نام mysite در نظر بگیریم:

```
https://www.mysite.com/api/users      # GET returns all users  
https://www.mysite.com/api/users/<id> # GET returns a single user
```

در اولین اندپوینت /api/users، یک درخواست GET لیستی از تمام کاربران موجود را برمی گرداند. این نوع اندپوینت که چندین دیتا را برمی گرداند به عنوان یک مجموعه (کالکشن) شناخته می شود.

اندپوینت دوم /api/users/id اطلاعات مربوط به یک کاربر را برمی گردد.

اگر متود POST را به اندپوینت اول اضافه کنیم، می توانیم یک کاربر جدید ایجاد کنیم، در حالی که افزودن متود DELETE به اندپوینت دوم به ما امکان حذف یک کاربر را می دهد.

در طول این کتاب با اندپوینت های API بیشتر آشنا خواهیم شد، اما در نهایت ایجاد یک API مستلزم ساخت یک سری اندپوینت است: URL هایی با HTTP verbs مرتبط.

یک صفحه وب از HTML، CSS، تصاویر و موارد دیگر تشکیل شده است. اما اندپوینت تنها راهی برای دسترسی به داده ها از طریق HTTP verbs موجود است.

HTTP

قبلًا در این فصل در مورد HTTP صحبت کردہ‌ایم، اما در اینجا توضیح خواهیم داد که در واقع HTTP چیست و چگونه کار می‌کند.

پروتکل HTTP یک پروتکل درخواست-پاسخ بین دو کامپیوتر است که در آن یک اتصال TCP وجود دارد. رایانه ای که درخواست ها را می فرستد به عنوان کلاینت شناخته می شود در حالی که رایانه پاسخ دهنده به عنوان سرور شناخته می شود. به طور معمول یک کلاینت یک مرورگر وب است اما می تواند یک برنامه iOS یا هر دستگاه متصل به اینترنت باشد. سرور نامی مناسب برای هر رایانه ای است که برای کار از طریق اینترنت بهینه شده است. تنها چیزی که برای تبدیل یک لپتاپ به سرور نیاز داریم یک نرم افزار مخصوص و اتصال دائمی به اینترنت است.

هر پیام HTTP از یک خط وضعیت(status)، سرصفحه درخواست(header) و داده های بدن(bdody) که اختیاری می باشد تشکیل شده است. به عنوان مثال، در اینجا یک نمونه پیام HTTP است که ممکن است یک مرورگر برای درخواست به صفحه اصلی Google <https://www.google.com> ارسال کند.

```
GET / HTTP/1.1
Host: google.com
Accept-Language: en-US
```

خط اول به عنوان خط درخواست شناخته می شود و متود پروتکل HTTP که استفاده شده را مشخص می کند (GET) (/) نشان دهنده مسیر و (HTTP/1.1) ورژن HTTP که در حال استفاده است را مشخص می کند.

دو خطی که در ادامه آمده سربرگ(header) HTTP می باشد: هاست همان نام دامنه و accept_Language هم همان زبان مورد استفاده است که در این مورد انگلیسی می باشد. به طور کل سربرگ های (header) HTTP در دسترس زیادی وجود دارد.

پیام های HTTP همچنین دارای بخش سوم اختیاری هستند که به عنوان بدن شناخته می شود. با این حال ما فقط یک پیام بدن با پاسخ های HTTP حاوی داده می بینیم.

برای سادگی، اجازه دهید فرض کنیم که صفحه اصلی گوگل فقط حاوی یک صفحه html به صورت "Hello, World" است.

پاسخ پیام HTTP از یک سرور Google ممکن است به این صورت باشد.

```
HTTP/1.1 200 OK
Date: Mon, 03 Aug 2020 23:26:07 GMT
Server: gws
Accept-Ranges: bytes
Content-Length: 13
Content-Type: text/html; charset=UTF-8
```

Hello, world!

خط اول خط پاسخ است و مشخص می کند که ما از HTTP/1.1 استفاده می کنیم. کد وضعیت 200 نشان دهنده موفقیت آمیز بودن درخواست مشتری است (به زودی در مورد کدهای وضعیت بیشتر توضیح خواهیم داد).

بنابراین هر پیام HTTP درخواست یا پاسخ (request, response)، فرمت زیر را دارد:

Response/request line
Headers...

(optional) Body

اکثر صفحات وب حاوی منابع متعددی هستند که به چندین چرخه درخواست/پاسخ HTTP نیاز دارند. اگر یک صفحه وب دارای HTML، یک فایل CSS و یک تصویر باشد، قبل از اینکه صفحه وب کامل در مرورگر نمایش داده شود، سه بار درخواست و پاسخ جداگانه بین کلاینت و سرور لازم است.

Status Codes

هنگامی که مرورگر وب شما یک درخواست HTTP را در یک URL فرستاد، هیچ تضمینی وجود ندارد که همه چیز واقعاً کار کند! بنابراین یک لیست طولانی از [کدهای وضعیت HTTP](#) همراه هر پاسخ HTTP وجود دارد.

می توانید نوع کلی کد وضعیت را بر اساس سیستم زیر تشخیص دهید:

- کدهای xx 2xx - درخواست توسط کلاینت دریافت، درک و پذیرفته شد
- کدهای xx 3xx - درخواست url منتقل شده است
- کدهای xx 4xx - خطای وجود دارد، معمولاً نوع درخواست URL به صورت نادرست توسط کلاینت فرستاده شده است.
- کدهای xx 5xx - ارور سرور - سرور نمیتواند درخواست کلاینت را به درستی پاسخ دهد.

نیازی به حفظ تمام کدهای وضعیت موجود نیست. با تمرین با رایج ترین موارد مانند 200 (OK)، 201 (ایجاد شده)، 301 (به طور دائم منتقل شده)، 404 (یافت نشد) و 500 (خطای سرور) آشنا خواهید شد.

نکته مهمی که باید به خاطر بسیارید این است که، به طور کلی، تنها چهار نتیجه کلی برای هر درخواست HTTP وجود دارد: انجام شد (2xx)، به نحوی هدایت شد (3xx)، خطای کلاینت (4xx)، یا سرور یک خطا ایجاد کرده است (5xx). این کدهای وضعیت به طور خودکار در خط درخواست/پاسخ در بالای هر پیام HTTP قرار می گیرند.

Statelessness(عدم وابستگی)

آخرین نکته مهم در مورد HTTP این است که یک پروتکل بدون وضعیت است. این بدان معناست که هر درخواست/پاسخ کاملاً مستقل از قبلی است. هیچ حافظه ذخیره شده ای از فعل و انفعالات گذشته وجود ندارد که در علوم کامپیوتر به عنوان وضعیت(state) شناخته می شود.

این ویژگی عدم وابستگی مزایای زیادی برای HTTP به همراه دارد. از آنجایی که همه سیستم‌های ارتباطی الکترونیکی در طول زمان علامت‌های خود را از دست می‌دهند، اگر پروتکل بدون وضعیت نداشته باشیم، بهتر بگوییم اگر یک چرخه درخواست/پاسخ طی نشود، همه چیز دائمًا خراب می‌شود. در نتیجه HTTP به عنوان یک پروتکل توزیع شده بسیار انعطاف‌پذیر شناخته می‌شود.

اما نکته منفی این است که مدیریت وضعیت(state) واقعاً در برنامه‌های وب بسیار مهم است. این موضوع وضعیت(state) این است که چگونه یک وب سایت به یاد می‌آورد که شما وارد سیستم شده‌اید و چگونه یک سایت فروشگاه اینترنتی سبد خرید شما را مدیریت می‌کند. این برای نحوه استفاده ما از وب سایت‌های مدرن است، اما در خود HTTP پشتیبانی نمی‌شود.

وضعیت‌ها قبلاً روی سرور حفظ می‌شد، اما بیشتر و بیشتر به سمت کلاینت، مرورگر وب، در فریمورک‌های فرانت‌اند مدرن مانند React، Angular و Vue منتقل شده‌اند. هنگامی که احراز هویت کاربر را پوشش می‌دهیم، درباره این موضوع بیشتر می‌آموزیم، اما به یاد داشته باشید که HTTP بدون وضعیت است. این باعث می‌شود برای ارسال مطمئن اطلاعات بین دو رایانه یک نکته مثبت باشد، اما در به خاطر سپردن هر چیزی خارج از هر درخواست/پاسخ فردی، یک ویژگی منفی باشد.

REST

موضوع REST (Representational State Transfer) یک معماری است که اولین بار در سال 2000 توسط روی فیلدینگ در پایان نامه خود پیشنهاد شد. این رویکردی است که بالای معماری دنیای وب یعنی بالای موضوع پروتکل HTTP قرار می‌گیرد.

کل کتاب دارای مجموعه پروتکل اینترنت می‌باشد که در مورد آنچه که API را واقعاً REST می‌کند یا نه نوشته شده است. اما سه ویژگی اصلی وجود دارد که ما در اینجا برای اهداف خود روی آنها تمرکز خواهیم کرد. هر RESTful API دارای:

- مانند HTTP بدون وضعیت(stateless) است
- CRUD (GET, POST, PUT, DELETE, etc) از پشتیبانی می‌کند
- داده‌ها را در قالب JSON یا XML برمی‌گرداند

هر API RESTful حداقل باید این سه اصل را داشته باشد. این استاندارد مهم است زیرا یک اصول ثابت برای طراحی و استفاده از API‌های وب ارائه می‌دهد.

Conclusion(نتیجه‌گیری)

در حالی که فناوری‌های زیادی در زیربنای اینترنت مدرن وجود دارد، ما به عنوان توسعه دهنده‌گان مجبور نیستیم همه آن را از ابتدا پیاده سازی کنیم. ترکیب زیبای Django REST Framework به درستی اکثر پیچیدگی‌های مربوط به API‌های وب را کنترل می‌کند. با این حال، داشتن حداقل درک کلی از کار کردن این معماری‌ها با هم مهم است.

در نهایت یک وب API مجموعه ای از اندپوینت هاست که بخش های خاصی از یک دیتابیس را نمایش می دهد. ما به عنوان توسعه دهنده، URL ها را برای هر اندپوینت کنترل می کنیم، که چه داده هایی در دسترس است و چه اقداماتی از طریق HTTP verbs ممکن است. همانطور که در ادامه کتاب خواهیم دید، با استفاده از هدرهای HTTP، می توانیم سطوح مختلف احراز هویت و دسترسی را نیز تنظیم کنیم.

کتابخانه سایت و API

جنگو رست Django REST Framework در کنار Django web Framework برای ایجاد رابط های برنامه های کاربردی وب (web APIs) ها کار می کند. ما نمی توانیم تنها با استفاده از Django REST Framework برای ساخت یک رابط برنامه کاربردی و استفاده کنیم بلکه همواره باید آن را بعد از نصب و پیکربندی Django به پروژه اضافه کنیم. در این فصل، ما شباهت ها و تفاوت های بین جنگو مرسوم (Django web Framework Traditional Django) و Django REST Framework می پردازیم. مهمترین تفاوت این است که جنگو وبسایت هایی محتوی صفحات وب را ایجاد می کند در حالیکه Django REST Framework به ایجاد رابط برنامه های کاربردی می پردازد که مجموعه ای از نقاط پایانی (endpoint) را که شامل متدهای HTTP در دسترس که پاسخی در قالب JSON باز میگردانند، می باشند. برای نشان دادن این مفاهیم، ما یک وبسایت کتابخانه با جنگوی مرسوم می سازیم و سپس آن را با کمک Django REST Framework به یک API web می تبدیل کنیم.

طمئن شوید که Python 3 و Pipenv بر روی کامپیوتر شما نصب شده باشد. اگر به راهنمایی نیاز دارید، دستورالعمل کامل در [اینجا](#) می باشد.

جنگو مرسوم

ابتدا ما نیاز داریم که محلی را برای ذخیره کد بر روی کامپیوترا من اختصاص دهیم. این محل میتواند هر جایی در سیستم شما باشد اما اگر از مک (macOS) استفاده می کنید برای راحتی آن را در پوشه Desktop قرار دهید. محل قرارگیری واقعاً اهمیتی ندارد و فقط باید به راحتی قابل دسترسی باشد.

```
$ cd ~/Desktop  
$ mkdir code && cd code
```

این پوشه code محل قرارگیری تمام کدهای داخل این کتاب خواهد بود. قدم بعدی ایجاد یک محل اختصاصی برای وبسایت کتابخانه مان، نصب جنگو با Pipenv، و بعد ورود به محیط مجازی با استفاده از دستور shell است. شما همیشه باید یک محیط مجازی اختصاصی برای هر پروژه جدید پایتون استفاده کنید.

```
$ mkdir library && cd library  
$ pipenv install django~=3.1.0  
$ pipenv shell  
(library) $
```

Pipenv درون مسیر و محل فعلی یک Pipfile و Pipfile.lock میسازد. کلمه library (اسم محیط مجازی) داخل پرانتز نشان دهنده این است که محیط مجازی ما فعال است. یک وبسایت جنگویی مرسوم از یک پروژه تنها و یک یا بیشتر از یک برنامه (app) که اعمال جداگانه ای را برعهده دارند، تشکیل شده اند. باید با دستور startproject یک پروژه جدید ایجاد کنیم. گذاشتן علامت نقطه را در آخر دستور برای نصب کدها در محل فعلی فراموش نکنید. اگر نقطه را نگذارید، جنگو یک پوشه اضافی بصورت پیشفرض میسازد.

```
(library) $ django-admin startproject config .
```

جنگو بصورت خودکار یک پروژه جدید برای ما تولید میکند که میتوانیم با دستور tree آن را مشاهده کنیم.(توجه: اگر tree کار نمیکند، آن را با [Homebrew](#) نصب کنید: brew install tree).

```
(library) $ tree
.
├── Pipfile
├── Pipfile.lock
└── config
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py

1 directory, 8 files
```

فایل ها وظایف زیر را برعهده دارند:

- فایل `__init__.py` یک روش پایتونی است برای رفتار کردن با یک محل یا دایرکتوری مانند یک پکیج؛ این فایل خالی است.
- فایل `settings.py` تمامی تنظیمات و پیکربندی های پروژه ما را شامل میشود.
- فایل `urls.py` مسیرهای URL سطح بالا را کنترل میکند.
- فایل `wsgi.py` مخفف web server gateway interface است و به جنگو کمک میکند که صفحات وب نهایی را سرویس دهی کند.
- فایل `manage.py` دستورات مختلف جنگو را از قبیل اجرا کردن وب سرور محلی(`local web server`) یا ایجاد یک برنامه جدید را اجرا میکند.

دستور migrate را برای همگام سازی پایگاه داده با تنظیمات پیش فرض جنگو اجرا کنید و وب سرور محلی جنگو را بالا بیاورید.

```
(library) $ python manage.py migrate
(library) $ python manage.py runserver
```

یک مرورگر وب باز کنید و به آدرس <http://127.0.0.1:8000> بروید تا از نصب درست پروژه اطمینان حاصل کنید.

Django: the Web framework for [+ 127.0.0.1:8000](#)

django View release notes for Django 2.2



The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.

 [Django Documentation](#)
Topics, references, & how-to's

 [Tutorial: A Polling App](#)
Get started with Django

 [Django Community](#)
Connect, get help, or contribute

Django welcome page

اولین برنامه

قدم بعدی اضافه کردن برنامه هایی است که مسئول محدوده های مجازی از عملکرد پروژه هستند. یک پروژه جنگو میتواند دارای چندین برنامه باشد.

서ور محلی را با کلیدهای ترکیبی `Control+c` متوقف کنید و سپس یک برنامه بنام `books` ایجاد کنید.

```
(library) $ python manage.py startapp books
```

حالا ببینیم که جنگو چه فایلهایی تولید کرده است.

```
(library) $ tree
.
├── Pipfile
├── Pipfile.lock
└── books
    └── __init__.py
```

```

    ├── admin.py
    ├── apps.py
    └── migrations
        └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── library_project
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── manage.py

```

هر برنامه یک فایل `__init__.py` دارد که آن را به عنوان یک پکیج پایتون معرفی می کند. 6 فایل جدید ایجاد شده اند:

- فایل `admin.py` یک فایل پیکربندی برای برنامه داخلی مدیر جنگو.
- فایل `apps.py` یک فایل پیکربندی برای خود برنامه.
- دایرکتوری `migrations` فایلهای `migrations` را برای تغییرات پایگاه داده ذخیره می کند.
- فایل `models.py` جاییکه ما مدل های پایگاه داده را تعریف می کنیم.
- فایل `tests.py` برای تست های خاص برنامه می باشد.
- فایل `views.py` جایی است که ما منطق درخواست و پاسخ(request/response) برنامه وب (web app) را پیاده می کنیم.

معمولًا توسعه دهنگان هم در هر برنامه یک فایل `urls.py` هم برای مسیردهی ایجاد می کنند.

بیایید فایل ها را بسازیم تا پروژه کتابخانه ما همه کتاب ها را در صفحه اصلی فهرست کند.

فایل `settings.py` را با ویرایشگر متن باز کنید. اولین قدم اضافه کردن برنامه های جدید به تنظیمات `INSTALLED_APPS` می باشد. ما همیشه برنامه های جدید را به آخر این لیست اضافه می کنیم چون جنگو آن ها را به ترتیب می خواند و ما می خواهیم که برنامه های هسته داخلی جنگو مانند `admin` و `auth` قبل از برنامه های ما بارگیری شوند.

```

# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Local
    'books', # new
]

```

سپس دستور `migrate` را برای همگام سازی پایگاه داده با تغییرات، اجرا می کنیم.

```
(library) $ python manage.py migrate
```

هر صفحه وب در جنگو مرسوم به چندین فایل نیاز دارد: یک view و یک url. اما قبل از همه ما به یک مدل پایگاه داده نیاز داریم بنابراین بباید از آن شروع کنیم.

مدل ها

فایل را در ویرایشگر متن(text editor) خود باز کنید و آن را بصورت زیر بروزرسانی کنید:

```
# books/models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=250)
    subtitle = models.CharField(max_length=250)
    author = models.CharField(max_length=100)
    isbn = models.CharField(max_length=13)

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

این یک مدل ساده جنگو است که در خط بالا آن ما models را از Django وارد می کنیم و سپس یک کلاس book ایجاد کرده ایم که آن را توسعه می دهد. درون این کلاس چهار فیلد وجود دارد: title، subtitle، author، isbn. ما همچنین یک متد `__str__` را ایم تا عنوان کتاب در قسمت مدیریت نمایش داده شود.

توجه داشته باشید که ISBN یک شناسه 13 حرفی یکتا میباشد که به هر کتاب منتشر شده ای اختصاص داده می شود.

چون ما یک مدل پایگاه داده جدید ساخته ایم نیاز به ساخت یک فایل مهاجرت(migration) داریم تا تغییر ایجاد شده در پایگاه داده هم اعمال شود. مشخص کردن نام برنامه اختیاری است اما اینجا توصیه می شود. می توانیم تنها تایپ کنیم `python manage.py makemigrations` اما اگر چندین برنامه که تغییرات پایگاه داده ای داشته اند، وجود داشته باشد، همه آنها به فایل migrations اضافه میشوند که خطایابی و رفع اشکال را در آینده تبدیل به یک چالش میکند. فایل های migrations خود را تا جای ممکن مجزا نگه دارید.

سپس دستور `migrate` را برای بروزرسانی پایگاه داده اجرا کنید.

```
(library) $ python manage.py makemigrations books
Migrations for 'books':
  books/migrations/0001_initial.py
    - Create model Book
(library) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, books, contenttypes, sessions
Running migrations:
  Applying books.0001_initial... OK
```

اگر هرکدام از مطالبی که تا اینجا بررسی کردیم برای شما جدید بنظر می آید، به شما پیشنهاد می کنم مطالعه این کتاب را همینجا خاتمه دهید و برای تشریح جزئی تر جنگو مرسوم، کتاب [Django for Beginners](#) را مرور کنید.

مدیر

ما می توانیم وارد کردن داده به مدل جدیدمان را با استفاده از برنامه داخلی جنگو انجام دهیم. اما اول باید دو کار را انجام دهیم: ایجاد یک حساب کاربری superuser و بروزرسانی فایل admin.py تا برنامه books نمایش داده شود.

با اکانت superuser شروع می کنیم. در ترمینال دستور زیر را اجرا کنید.

```
(library) $ python manage.py createsuperuser
```

مطابق درخواست ها یک username، email و password وارد کنید. توجه داشته باشید که به دلایل امنیتی، متن در هنگام وارد کردن password بر روی صفحه نمایش داده نمی شود. حالا فایل admin.py را بروزرسانی کنید.

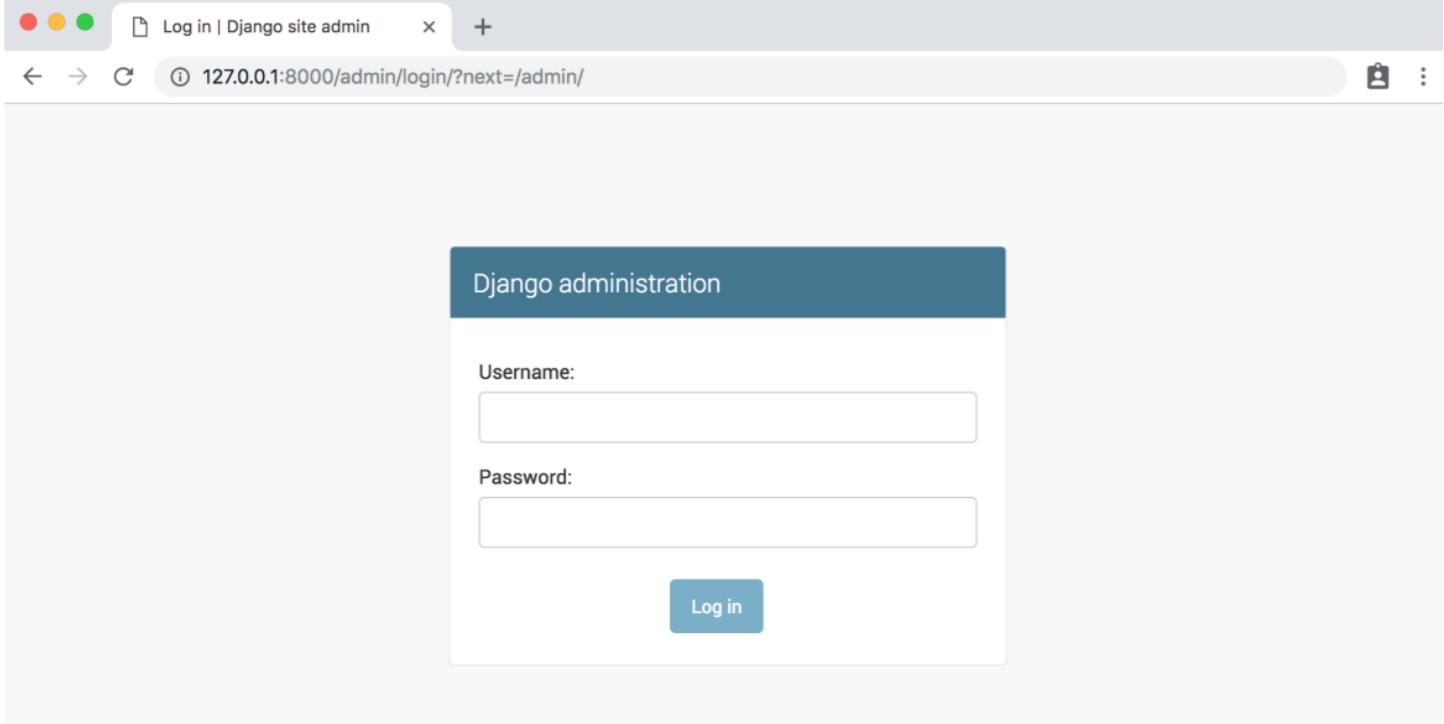
```
# books/admin.py
from django.contrib import admin
from .models import Book

admin.site.register(Book)
```

این تمام کاری بود که باید انجام می دادیم! سرور محلی را دوباره بالا بیاورید.

```
(library) $ python manage.py runserver
```

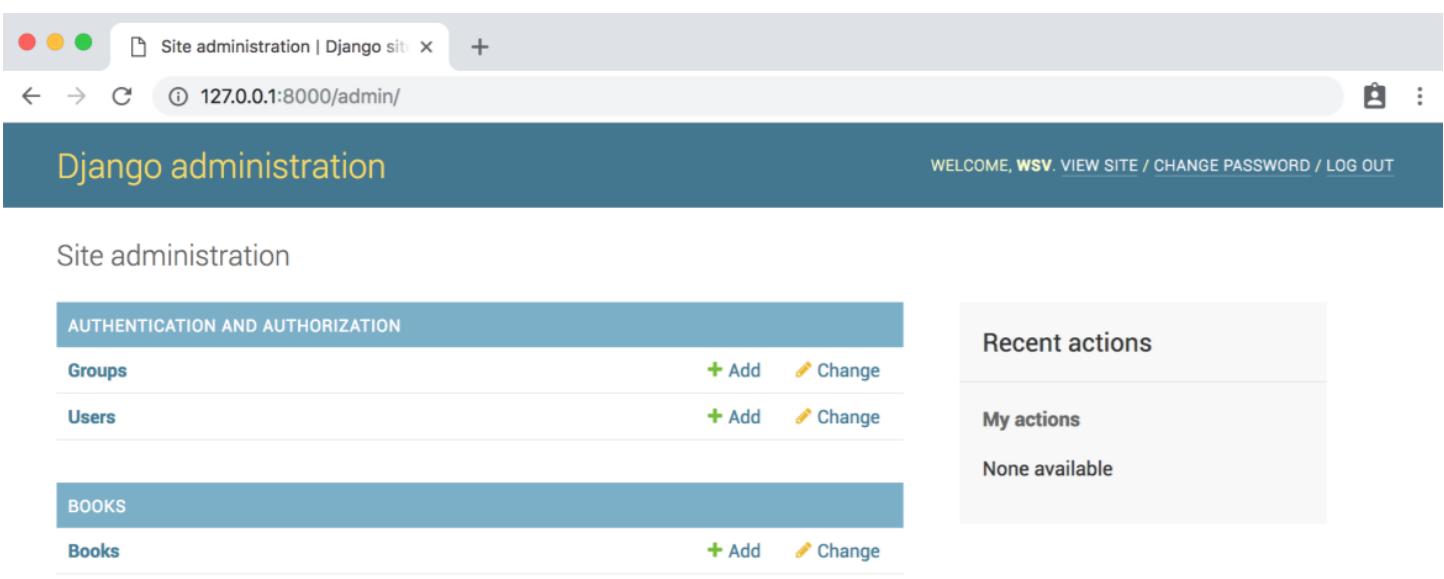
به آدرس <http://127.0.0.1:8000/admin> بروید و وارد شوید.



The screenshot shows the Django administration login interface. It features a dark blue header bar with the text "Django administration". Below it is a light gray form area containing two input fields: "Username:" and "Password:", each with a corresponding text input box. A blue "Log in" button is centered at the bottom of the form.

Admin login

شما به صفحه خانه admin منتقل خواهید شد.



The screenshot shows the Django administration homepage. At the top, there's a header bar with the text "Site administration | Django site" and the URL "127.0.0.1:8000/admin/". To the right of the header is a welcome message "WELCOME, wsv. [VIEW SITE](#) / [CHANGE PASSWORD](#) / [LOG OUT](#)". The main content area has a dark blue header bar with the text "Django administration". Below this, there are two sections: "AUTHENTICATION AND AUTHORIZATION" and "BOOKS". The "AUTHENTICATION AND AUTHORIZATION" section contains links for "Groups" and "Users", each with "Add" and "Change" buttons. The "BOOKS" section contains a link for "Books", also with "Add" and "Change" buttons. To the right of the main content is a sidebar titled "Recent actions" which lists "My actions" and "None available".

Admin homepage

روی لینک Books کلیک کنید.

The screenshot shows the Django administration interface. At the top, there's a header bar with the title "Select book to change | Django" and the URL "127.0.0.1:8000/admin/books/book/". Below the header, the title "Django administration" is displayed, along with a "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT" message. A navigation bar shows "Home > Books > Books". The main content area is titled "Select book to change" and contains a message "0 books". In the top right corner, there's a button labeled "ADD BOOK +".

Admin books page

سپس روی دکمه Add Book + در گوشه بالا سمت راست کلیک کنید.

The screenshot shows the "Add book" form within the Django administration interface. The title bar says "Add book | Django site admin" and the URL is "127.0.0.1:8000/admin/books/book/add/". The main content area is titled "Django administration" and "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". The navigation bar shows "Home > Books > Books > Add book". The form itself has four fields: "Title" (Django for Beginners), "Subtitle" (Build websites with Python and Django), "Author" (William S. Vincent), and "Isbn" (978-198317266). At the bottom of the form are three buttons: "Save and add another", "Save and continue editing", and a larger "SAVE" button.

Admin add book

من جزئیات را برای کتاب Django for Beginners خودم وارد کرده ام. شما می توانید هر متنی که بخواهید اینجا وارد کنید. این اطلاعات صرفا برای اهداف نمایشی است. بعد از کلیک کردن روی دکمه Save به صفحه Books که در آن تمام اطلاعات ثبت شده فعلی لیست شده اند، منتقل می شویم.

The screenshot shows the Django administration interface. At the top, there's a header bar with the title "Django administration" and a "WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT" link. Below the header, a navigation bar shows "Home > Books > Books". A green success message box at the top states "The book "Django for Beginners" was added successfully." The main content area is titled "Select book to change" and contains a table with one row. The table has columns for "Action", "BOOK", and "Django for Beginners". There is a single checkbox next to "BOOK". The footer of the page includes a "ADD BOOK +" button.

Admin books list

پروژه جنگو مرسوم ما آلان داده دارد اما ما یک راه برای نمایش آن به عنوان یک صفحه وب نیاز داریم. این یعنی ایجاد فایل های URLs و templates، views باید اینکار را آلان انجام دهیم.

(views) نما ها

فایل نحوه نمایش محتوای مدل های پایگاه داده را کنترل می کند. از آنجاییکه ما میخواهیم تمام کتابها را لیست کنیم می توانیم از کلاس عمومی داخلی [ListView](#) استفاده کنیم. فایل books/views.py را بروزرسانی کنید.

```
# books/views.py
from django.views.generic import ListView
from .models import Book

class BookListView(ListView):
    model = Book
    template_name = 'book_list.html'
```

در خطوط بالا ما ListView و مدل Book خودمان را وارد کرده ایم. سپس یک کلاس BookListView که مشخص میکند از چه مدل و قالبی(template) باید استفاده شود، ایجاد می کنیم.(توجه داشته باشید که قالب را هنوز نساخته ایم.)

دو گام دیگر برای اینکه یک صفحه پیج آماده داشته باشیم، باقیمانده است: درست کردن قالب و پیکربندی URL ها. باید با URL ها شروع کنیم.

مکان یاب های منبع یکسان(URLs)

نیاز است که هم فایل urls.py اصلی که مربوط به پروژه و هم فایل books داریم را تنظیم کنیم. وقتی یک کاربر از سایت ما بازدید می کند ابتدا با فایل library_project/urls.py تعامل خواهد داشت بنابراین ابتدا پیکربندیهای مربوط به این فایل را انجام می دهیم.

```
# library_project/urls.py
from django.contrib import admin
from django.urls import path, include # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('books.urls')), # new
]
```

دو خط بالا برنامه داخلی admin، متده path برای مسیرهای ما و متده include که با برنامه books ما استفاده می شود را وارد می کنند. اگر کاربر به /admin/ برود به برنامه admin منتقل می شود. ما برای مسیر برنامه books از رشته خالی " استفاده کردیم که این معنی را می دهد که یک کاربر در صفحه اصلی بطور مستقیم به برنامه books منتقل می شود.

حالا ما می توانیم فایل books/urls.py را تنظیم کنیم. اما جنگو به دلایل بطور پیش فرض شامل فایل urls.py در برنامه ها نمی باشد بنابراین خود ما باید آن را ایجاد کنیم.

```
(library) $ touch books/urls.py
```

حالا با استفاده از یک ویرایشگر متن فایل جدید را بروزرسانی کنید.

```
# books/urls.py
from django.urls import path
from .views import BookListView

urlpatterns = [
    path('', BookListView.as_view(), name='home'),
]
```

ما فایل views.py خودمان را وارد می کنیم، BookListView را در آدرس رشته خالی " تنظیم می کنیم و یک named با مقدار home به عنوان یک URL اضافه می کنیم.

روشی که جنگو کار می کند، حالا وقتی یک کاربر به صفحه خانه وب سایت ما می رود، آنها نخست به فایل library_project/urls.py می رسند سپس به books/urls.py منتقل می شوند که مشخص می کند باید از استفاده شود. در این فایل view از مدل BookListview برای فهرست کردن همه کتاب ها استفاده شده است.

قدم نهایی ایجاد فایل قالب است که چیدمان در صفحه حقیقی وب را کنترل می کند. ما قبلاً نام آن (book_list.html) را در فایل view مشخص کرده ایم. دو گزینه برای مکان آن وجود دارد: بطور پیش فرض بارکننده قالب جنگو داخل برنامه books ما در مسیر books/templates/books/book_list.html بدنبال قالب ها می گردد. همچنین ما می توانیم یک دایرکتوری templates جدا هم سطح پروژه ایجاد کنیم و فایل settings.py را برای رفتن به آن مکان برای یافتن قالب ها بروزرسانی کنیم.

اینکه شما در پروژه های خود از کدام روش استفاده کنید یک ترجیح شخصی است. ما در اینجا از ساختار پیشفرض استفاده می کنیم. اگر درباره روش دوم کنجکاو هستید، کتاب [Django for Beginners](#) را بررسی کنید.

با ساخت یک پوشه templates در برنامه books شروع می کنیم، سپس داخل آن یک پوشه books و در آخر یک فایل book_list.html در آن می سازیم.

```
(library) $ mkdir books/templates  
(library) $ mkdir books/templates/books  
(library) $ touch books/templates/books/book_list.html
```

حالا فایل template را بروزرسانی کنید.

```
<!-- books/templates/books/book_list.html -->  
<h1>All books</h1>  
{% for book in object_list %}  
  <ul>  
    <li>Title: {{ book.title }}</li>  
    <li>Subtitle: {{ book.subtitle }}</li>  
    <li>Author: {{ book.author }}</li>  
    <li>ISBN: {{ book.isbn }}</li>  
  </ul>  
{% endfor %}
```

جنگو با [زبان قالبی](#) عرضه می شود که اجازه اعمال عملیات های منطقی ساده را به ما می دهد. در اینجا ما از تگ for برای حلقه زدن بر روی تمام کتاب های در دسترس استفاده می کنیم. تگ های قالب باید همیشه باید شامل برآکت های باز و بسته باشند. بنابراین فرمت همیشه بصورت { % ... for % } می باشد و سپس ما باید حلقه را با { % } خاتمه دهیم.

چیزی که ما در حال حقه زدن بر روی آن هستیم، شئ ای است که شامل تمامی کتابهای در دسترس در مدل ما می باشد. نام این شئ object_list است. بنابراین برای حلقه زدن بر روی هر کتاب ما می نویسیم { % object_list for book in % }. و سپس هر فیلد از مدلمان را نمایش می دهیم.

صفحه وب

حالا میتوانیم سرور محلی جنگو را بالا بیاوریم و صفحه وب خود را مشاهده کنیم.

```
(library) $ python manage.py runserver
```

به آدرس صفحه خانه که <http://127.0.0.1:8000> میباشد بروید.



All books

- Title: Django for Beginners
- Subtitle: Build websites with Python and Django
- Author: William S. Vincent
- ISBN: 978-198317266

[Book web page](#)

اگر ما کتاب های دیگری را در admin اضافه کنیم، آنها نیز در اینجا به نمایش درخواهند آمد.
این یک مرور سریع از وبسایت جنگویی مرسوم بود. حالا بباید به آن رابط برنامه کاربردی هم اضافه کنیم!

رست فریمورک جنگو(Django REST Framework)

رست فریمورک جنگو هم مانند بقیه برنامه های شخص ثالث نصب می شود. با فشردن کلیدهای Control+C سرور محلی را اگر در حال اجرا است، متوقف کنید سپس در خط فرمان دستور زیر را تایپ کنید.

```
(library) $ pipenv install djangorestframework==3.10.3
```

عبارت rest_framework را به تنظیمات settings.py در فایل INSTALLED_APPS اضافه کنید. من عادت دارم که یک فاصله بین برنامه های شخص ثالث و محلی قرار دهم به این دلیل که تعداد برنامه ها در بیشتر پروژه ها به سرعت افزایش پیدا می کنند.

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 3rd party
    'rest_framework', # new

    # Local
    'books',
]
```

در نهایت رابط برنامه کاربردی ما یک نقطه پایانی که تمام کتاب‌ها را در فرمت JSON لیست می‌کند، نمایش می‌دهد. بنابراین ما یک مسیر URL جدید، یک view جدید و یک فایل serializer جدید (درباره این فایل بزودی بیشتر می‌خوانید) نیاز خواهیم داشت. راه‌های زیادی وجود دارد که می‌توانیم این فایل‌ها را سازمان دهیم اگرچه ترجیح من این است که یک برنامه مختص به رابط‌های برنامه کاربردی ایجاد کنیم. در این روش حتی اگر در آینده هم برنامه‌های بیشتری اضافه کنیم، هر برنامه می‌تواند شامل model‌ها، view‌ها، template‌ها و url‌های مورد نیاز برای صفحات وب باشد اما تمام فایل‌های مختص به رابط‌های برنامه کاربردی مربوط به کل پروژه درون یک برنامه اختصاص یافته به رابط‌های برنامه کاربردی قرار می‌گیرند.

بیایید ابتدا یک برنامه جدید بنام api بسازیم.

```
(library) $ python manage.py startapp api
```

سپس آن را به INSTALLED_APPS اضافه کنید.

```
# config/settings.py
INSTALLED_APPS = [
    # Local
    'books.apps.BooksConfig',
    'api.apps.ApiConfig', # new

    # 3rd party
    'rest_framework',

    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

برنامه api مدل پایگاه داده ای مختص به خود ندارد بنابراین نیازی به ایجاد یک فایل مهاجرت و بروزرسانی پایگاه داده طبق روال معمول نیست.

مسیرها (URLs)

بیایید با تنظیم URL‌ها شروع کنیم. اضافه کردن یک نقطه پایانی درست مشابه پیکربندی مسیرهای یک برنامه جنگوی مرسوم می‌باشد. ابتدا نیاز است که برنامه api و مسیرهای آن را در فایل urls.py اصلی پروژه و در مسیر /api اضافه و تنظیم کنیم.

```
# library_project/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
```

```
    path('admin/', admin.site.urls),
    path('', include('books.urls')),
    path('api/', include('api.urls'))), # new
]
```

سپس یک فایل urls.py درون برنامه api می سازیم.

```
(library) $ touch api/urls.py
```

و آن را به شکل زیر بروزرسانی می کنیم:

```
# api/urls.py
from django.urls import path
from .views import BookAPIView

urlpatterns = [
    path('', BookAPIView.as_view()),
]
```

همه چیز آماده است.

نما ها

مرحله بعد فایل views.py ما است که متکی به نماهای کلاسی عمومی داخلی(built-in generics class views) رست فریمورک جنگو می باشد. این نماها در ظاهر از نماهای کلاسی عمومی جنگوی مرسوم تقلید می کند اما این دو گروه از نماها یکی نیستند. برای جلوگیری از سردرگمی، بعضی از توسعه دهندگان یک فایل API views را api/views.py نامند. شخصا، وقتی که در یک برنامه مختص به رابط های برنامه کاربردی کار می کنم دچار سردرگمی نمی شوم اگر که نما های مربوط به Django REST Framework را بنامم اما نظر ها در اینباره متفاوت هستند.

محتوای فایل views.py را بصورت زیر بروزرسانی کنید:

```
# api/views.py
from rest_framework import generics
from books.models import Book
from .serializers import BookSerializer

class BookAPIView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

در خطوط بالا نماهای کلاسی عمومی(generics class of views) رست فریمورک جنگو، مدل ها را از برنامه books و serializers را از برنامه api (ما بعدا ایجاد می کنیم) وارد کرده ایم.

سپس یک BookAPIView که از ListAPIView برای ایجاد یک نقطه پایانی فقط خواندنی برای تمام نمونه های کتاب استفاده می کند، می سازیم. چندین نمای عمومی در دسترس وجود دارند که جلوتر در فصل های بعد آنها را بررسی خواهیم کرد. دو قدمی که در نمای ما نیاز است، مشخص کردن queryset که تمام کتاب های در دسترس می باشد و سپس serializer_class که BookSerializer می باشد، است.

سربالایزرها(Serializers)

یک سربالایزر داده را به شکلی که استفاده از آنها در اینترنت راحت باشد، معمولا JSON، و در یک نقطه پایان رابط برنامه تعاملی به نمایش درمی آید، درمی آورد. در فصل های آینده سربالایزرها و JSON را جزئی تر پوشش خواهیم داد. برای آن من میخواهم نشان دهم که ساخت یک سربالایزر با استفاده از رست فریمورک جنگو برای تبدیل مدل های جنگو به JSON چقدر آسان است.

یک فایل serializer.py درون برنامه api درست کنید.

```
(library) $ touch api/serializers.py
```

سپس آن را بصورت زیر در ویرایشگر متن بروزرسانی کنید.

```
# api/serializers.py
from rest_framework import serializers
from books.models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ('title', 'subtitle', 'author', 'isbn')
```

در خطوط بالا ما کلاس serializer رست فریمورک جنگو و مدل Book را از فایل مدل برنامه books وارد می کنیم. ما ModelSerializer رست فریمورک جنگو را در کلاس BookSerializer که مدل پایگاه داده Book و فیلد های پایگاه داده ای که می خواهیم نمایش بدهیم(title, subtitle, author, isbn) را مشخص می کند، توسعه می دهیم.

کرل(cURL)

ما میخواهیم ببینیم نقطه پایانی رابط برنامه کاربردی ما به چه شکلی است. می دانیم که باید در مسیر یک JSON برگرداند. بباید مطمئن شویم که سرور محلی جنگو در حال اجرا است:

```
(library) $ python manage.py runserver
```

حالا یک خط فرمان دوم جدید باز کنید. از این خط فرمان برای رفتن به آدرس رابط برنامه کاربردی که در خط فرمان موجود در حال اجرا است، استفاده می کنیم.

ما می توانیم از برنامه محبوب [curl](#) برای اجرا درخواست های HTTP در خط فرمان استفاده کنیم. تمام چیزی که برای یک درخواست GET ساده نیاز داریم این است که curl را بنویسیم و URL ای که می خواهیم صدا بزنیم را برای آن مشخص کنیم.

```
$ curl http://127.0.0.1:8000/api/
[
  {
    "title": "Django for Beginners",
    "subtitle": "Build websites with Python and Django",
    "author": "William S. Vincent",
    "isbn": "978-198317266"
  }
]
```

تمام داده ها به شکل JSON در آنجا هستند اما فالب بندی ضعیفی دارند و درک آن ها سخت است. خوشبختانه Django REST Framework یک شگفتی دیگر هم برای ما دارد: یک حالت بصری قدرتمند برای نقاط پایانی رابط های برنامه کاربردی ما.

رابط برنامه کاربردی تحت مرورگر(Browsable API)

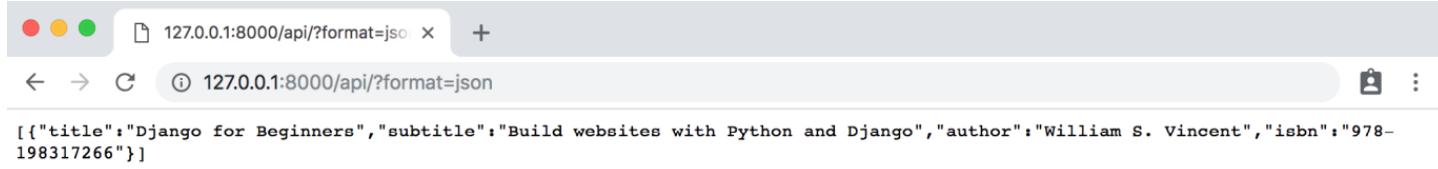
در حالیکه سرور محلی همچنان در اولین خط فرمان در حال اجرا است، در مرورگر به آدرس [بروید](http://127.0.0.1:8000/api).

The screenshot shows a web browser window with the title 'Book Api - Django REST frame'. The address bar contains '127.0.0.1:8000/api/'. The main content area is titled 'Django REST framework' and 'Book Api'. Below this, there is a 'GET /api/' button. To the right of the button are 'OPTIONS' and 'GET' buttons. The response body shows the following JSON data:

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
  {
    "title": "Django for Beginners",
    "subtitle": "Build websites with Python and Django",
    "author": "William S. Vincent",
    "isbn": "978-198317266"
  }
]
```

وای به آن نگاه کنید! Django REST Framework این تجسم را برای ما بطور پیشفرض فراهم کرده است. و امکانات زیادی در این صفحه تعییه وجود دارد که در طول کتاب آن ها را بررسی می کنیم. حالا میخواهیم این صفحه را با نقطه پایانی JSON خام مقایسه کنیم. بر روی دکمه GET کلیک کنید و از منو گزینه json را انتخاب کنید.



The screenshot shows a browser window with the URL `127.0.0.1:8000/api/?format=json`. The page title is "Book API JSON". The content of the page is a single line of JSON code:

```
[{"title": "Django for Beginners", "subtitle": "Build websites with Python and Django", "author": "William S. Vincent", "isbn": "978-198317266"}]
```

Book API JSON

این شکل JSON خام نقطه پایانی رابط برنامه کاربردی ما است. فکر می کنم میتوانیم توافق کنیم که نسخه Django REST Framework، جذاب تر است.

نتیجه گیری

ما در این فصل مباحث زیادی را پوشش دادیم پس اگر موضوعات آن کمی گیج کننده بنظر می رسند، نگران نباشید. ابتدا یک وبسایت کتابخانه جنگویی مرسوم ساختیم. سپس Django REST Framework را اضافه کردیم و یک با تعداد کمی کد یک نقطه پایانی رابط برنامه کاربردی ساختیم. در دو فصل بعدی ما بک اند رابط برنامه کاربری خود را می سازیم و آن را به یک فرانت اند ریکتی متصل می کنیم تا یک مثال کامل را نشان دهیم که به درک اینکه در عمل تمام این اصول چگونه به یکدیگر مرتبط می شوند کمک می کند.

Todo API

در طول دو فصل بعدی، یک React front-end Todo API back-end می‌سازیم و سپس آن را با یک REST API خود را ساخته‌ایم و نحوه عملکرد HTTP و REST را به صورت انتزاعی بررسی کرده‌ایم، اما هنوز به احتمال زیاد شما هنوز «کاملاً» نمی‌بینید که چگونه همه با هم هماهنگ می‌شوند.

از آنجایی که ما در حال ساخت یک front-end و back-end اختصاصی هستیم، کد خود را به یک ساختار مشابه تقسیم می‌کنیم. در دایرکتوری کد موجود خود، یک دایرکتوری todo حاوی کد جنگو پایتون پشتیبان و کد JavaScript جلویی ایجاد خواهیم کرد. طرح نهایی به این صورت خواهد بود.

```
todo
|   ├── frontend
|   |   ├── React...
|   ├── backend
|   |   ├── Django...
```

این فصل بر روی back-end و فصل ۴ روی front-end تمرکز دارد.

تنظیمات اولیه

اولین قدم برای هر API جنگو این است که جنگو را نصب کنید و بعداً جنگو REST Framework را در بالای آن اضافه کنید. ابتدا یک دایرکتوری اختصاصی todo در دایرکتوری کد ما در دسکتاپ ایجاد کنید.

یک کنسول خط فرمان جدید باز کنید و دستورات زیر را در آن تایپ کنید:

```
$ cd ~/Desktop
$ cd code
$ mkdir todo && cd todo
```

توجه: مطمئن شوید که محیط مجازی را از فصل قبل غیرفعال کرده‌اید. می‌توانید این کار را با تایپ exit انجام دهید. آیا دیگر هیچ پرانتزی جلوی خط فرمان شما نیست؟ خوب سپس شما در یک محیط مجازی موجود نیستید.

در این پوشه todo، دایرکتوری های front-end و back-end ما قرار خواهند گرفت. بیایید پوشه Backend را ایجاد کنیم، جنگو را نصب کنیم و یک محیط مجازی جدید را فعال کنیم.

```
(backend) $ django-admin startproject config .
(backend) $ python manage.py startapp todos
(backend) $ python manage.py migrate
```

در جنگو ما همیشه نیاز داریم که برنامه‌های جدیدی را به تنظیمات INSTALLED_APPS خود اضافه کنیم، پس این کار را اکنون انجام دهید. config/settings.py را در ویرایشگر متن خود باز کنید. در انتهای فایل موارد زیر را اضافه کنید.

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Local
    'todos', # new
]
```

اگر اکنون سرور python manager.py را در خط فرمان اجرا کنید و در مرورگر وب خود به آدرس [/بروید](http://127.0.0.1:8000)، می توانید ببینید که پروژه ما با موفقیت نصب شده است.

Django: the Web framework for [x](#) +

127.0.0.1:8000 Guest :

django

[View release notes for Django 3.1](#)



The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

[!\[\]\(13093576e34411b016149ffa5b040d56_img.jpg\) Django Documentation](#)
Topics, references, & how-to's

[!\[\]\(b32d464f05edc0682a0399e0907cb00f_img.jpg\) Tutorial: A Polling App](#)
Get started with Django

[!\[\]\(3e0d1252f9a124926fc646730d6dc271_img.jpg\) Django Community](#)
Connect, get help, or contribute

برای رفتن به بخش بعد آماده ایم!

مدلها

Next up is defining our Todo database model within the todos app. We will keep things basic and have only two fields: title and body. todo خود را در برنامه Todo سپس باید مدل پایگاه داده ای ایجاد کنیم. قرار است همه چیز را ساده نگه داریم و فقط دو فیلد داشته باشیم: موضوع و بدنه متن موضوع

```
# todos/models.py
from django.db import models

class Todo(models.Model):
```

```
title = models.CharField(max_length=200)
body = models.TextField()

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

ما مدل‌ها را در بالا وارد می‌کنیم و سپس آن‌ها را برای ایجاد مدل Todo خودمان طبقه‌بندی می‌کنیم. ما همچنان یک متده «str» اضافه می‌کنیم تا برای هر نمونه مدل آینده یک نام قابل خواندن توسط انسان ارائه کنیم.

از آنجایی که مدل خود را به روزرسانی کردہ‌ایم، زمان رقص دو مرحله‌ای جنگو برای ساخت یک فایل مهاجرت جدید و سپس همگام‌سازی پایگاه داده با تغییرات هر بار فرا رسیده است. در خط فرمان Ctrl+c را تایپ کنید تا سرور محلی ما متوقف شود. سپس این دو دستور را اجرا کنید:

```
(backend) $ python manage.py makemigrations todos
Migrations for 'todos':
  todos/migrations/0001_initial.py
    - Create model Todo
(backend) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, todos
Running migrations:
  Applying todos.0001_initial... OK
```

اضافه کردن برنامه خاصی که می‌خواهیم برای آن یک فایل مهاجرت ایجاد کنیم اختیاری است – به جای آن می‌توانیم فقط makemigrations را تایپ کنیم – با این حال بهترین روش برای اتخاذ آن است. فایل‌های مهاجرت راهی فوق‌العاده برای اشکال‌زدایی برنامه‌ها هستند و شما باید برای هر تغییر کوچک یک فایل مهاجرت ایجاد کنید. اگر مدل‌ها را در دو برنامه مختلف به روز کرده بودیم و سپس پایتون را اجرا می‌کردیم makemigrations فایل مهاجرتی منفرد حاصل حاوی داده‌های مربوط به هر دو برنامه است. این فقط اشکال‌زدایی را سخت‌تر می‌کند. سعی کنید مهاجرت‌های خود را تا حد امکان کوچک نگه دارید.

اکنون می‌توانیم از برنامه داخلی Django admin برای تعامل با پایگاه داده خود استفاده کنیم. اگر بلافضله وارد адمنی می‌شدیم، برنامه Todo ما ظاهر نمی‌شد. ما باید به صراحت آن را از طریق فایل todos/admin.py به صورت زیر اضافه کنیم.

```
# todos/admin.py
from django.contrib import admin
from .models import Todo

admin.site.register(Todo)
```

درست! اکنون می‌توانیم یک حساب کاربری فوق‌العاده برای ورود به адمن ایجاد کنیم.

```
(backend) $ python manage.py createsuperuser
```

و سپس سرور محلی را دوباره راه اندازی کنید:

```
(backend) $ python manage.py runserver
```

اگر به آدرس <http://127.0.0.1:8000/admin> بروید، اکنون می توانید وارد شوید. روی "+ Add" در کنار Todos کلیک کنید و 3 مورد جدید برای انجام کار ایجاد کنید، مطمئن شوید که یک عنوان و متن برای هر دو اضافه کنید. مال من به این صورت است:

The screenshot shows the Django administration interface. In the top navigation bar, it says "Select todo to change | Django" and has a "+ Add" button. The URL is "127.0.0.1:8000/admin/todos/todo/". On the right, it says "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT" and "Guest". The main title is "Django administration". Below it, the breadcrumb navigation shows "Home > Todos > Todos". On the left, there's a sidebar with "AUTHENTICATION AND AUTHORIZATION" section containing "Groups" and "Users" with "+ Add" buttons. The main content area has a heading "Todos" with a "+ Add" button. It lists four todos: "TODO", "Learn HTTP", "Second item", and "1st todo", each with a checkbox. At the bottom, it says "3 todos". Above the list, a message says "The todo "Learn HTTP" was added successfully."

در واقع در این مرحله کار با بخش سنتی جنگو از Todo API ما تمام شده است. از آنجایی که ما زحمت ایجاد صفحات وب برای این پروژه را نداریم، نیازی به نشانی‌های وب، نماها، یا قالب‌ها نیست. تنها چیزی که ما نیاز داریم یک مدل است و Django REST Framework بقیه را بر عهده خواهد گرفت.

Django REST Framework

با زدن `Control+c` سرور لوکال خود را قطع کنید و `pipenv install djangorestframework==3.11.0` را از طریق Django REST Framework نصب کنید.

```
(backend) $ pipenv install djangorestframework==3.11.0
```

سپس `rest_framework` را مانند هر برنامه شخص ثالث دیگری به تنظیمات `INSTALLED_APPS` خود اضافه کنید. همچنین می‌خواهیم تنظیمات خاص چارچوب REST جنگو را که همه در `REST_FRAMEWORK` وجود دارند، پیکربندی کنیم. برای شروع، اجازه دهید به صراحت مجوزها را روی `AllowAny` تنظیم کنیم. این خط در پایین فایل قرار می‌گیرد.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```
# 3rd party
'rest_framework', # new

# Local
'todos',
]

# new
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

جنگو رست فهرست طولانی‌ای از تنظیمات پیش‌فرض به طور ضمنی دارد. لیست کامل را می‌توانید در [اینجا](#) ببینید. AllowAny یکی از آنهاست، به این معنی که وقتی ما آن را به صراحت تنظیم می‌کنیم، همانطور که در بالا انجام دادیم، اثر دقیقاً مشابه است که اگر هیچ مجموعه پیکربندی DEFAULT_PERMISSION_CLASSES نداشته باشیم.

یادگیری تنظیمات پیش فرض چیزی است که زمان می برد. در طول کتاب با تعدادی از آنها آشنا می شویم. نکته اصلی که باید به خاطر بسپارید این است که تنظیمات پیش فرض ضمنی به گونه‌ای طراحی شده‌اند که توسعه دهنده‌گان بتوانند به سرعت در یک محیط توسعه محلی شروع به کار کنند. هر چند تنظیمات پیش فرض برای تولید مناسب نیست. بنابراین معمولاً ما در طول یک پروژه تغییراتی در آنها ایجاد می کنیم.

خوب، یس Django REST Framework نصب شده است. بعدش چی؟

برخلاف پروژه کتابخانه در فصل قبل که هم یک صفحه وب و هم یک API ساختیم، در اینجا فقط یک API ایجاد می‌کنیم. بنابراین ما نیازی به ایجاد هیچ فایل قالب یا نمای سنتی چنگو نداریم.

در عوض، ما سه فایل را که برای تبدیل مدل پایگاه داده ما به یک وب API اختصاص دارند، به روزرسانی می‌کنیم:
serializers.py، urls.py، و views.py

URLs

من دوست دارم ابتدا با URL ها شروع کنم زیرا آنها نقطه ورودی برای نقاط پایانی API ما هستند. درست مانند پروژه حنگو سنتری، فایل urls.py به ما امکان می دهد مسیر پایی را بیکرندی کنیم.

از فایل سطح پروژه جنگو که config/urls.py است شروع کنید. ما شامل را در خط دوم وارد می کنیم و یک مسیر برای برنامه todos خود در api/ اضافه می کنیم.

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('todos.urls')), # new
]
```

```
(backend) $ touch todos/urls.py
```

و با کد زیر آن را آپدیت کنید.

```
# todos/urls.py
from django.urls import path
from .views import ListTodo, DetailTodo

urlpatterns = [
    path('<int:pk>/', DetailTodo.as_view()),
    path('', ListTodo.as_view()),
]
```

توجه داشته باشید که ما به دو نما - DetailTodo و ListTodo می کنیم که هنوز باید ایجاد کنیم. اما مسیریابی اکنون کامل شده است. لیستی از همه کارها در رشته خالی "، به عبارت دیگر در /api وجود خواهد داشت. و هر کار جداگانه در کلید اصلی خود در دسترس خواهد بود، که مقداری است که جنگو به طور خودکار در هر جدول پایگاه داده تنظیم می کند. ورودی اول 1، دومی 2 و غیره است. بنابراین اولین کار ما در نهایت در نقطه پایانی /api/1/ قرار خواهد گرفت.

سریال ساز ها

بیایید مرور کنیم که تا به حال کجا هستیم. ما با یک پروژه و برنامه سنتی جنگو شروع کردیم که در آن یک مدل پایگاه داده ساختیم و داده ها را اضافه کردیم. سپس Django REST Framework را نصب کردیم و URL های خود را پیکربندی کردیم. اکنون باید داده های خود را از مدل ها به JSON تبدیل کنیم که در URL ها خروجی می شود. بنابراین ما به یک سریال ساز نیاز داریم.

با یک کلاس سریال ساز داخلی قدرتمند عرضه می شود که می توانیم به سرعت با مقدار کمی کد آن را گسترش دهیم. این کاری است که ما در اینجا انجام خواهیم داد.

ابتدا یک فایل serializers.py جدید در برنامه todos ایجاد کنید.

```
(backend) $ touch todos/serializers.py
```

سپس با کد زیر آن را آپدیت کنید.

```
# todos/serializers.py
from rest_framework import serializers
from .models import Todo

class TodoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Todo
        fields = ('id', 'title', 'body',)
```

در بالا، ما سریال‌سازها را از Django REST Framework خود وارد کردیم. سپس یک کلاس TodoSerializer ایجاد می‌کنیم. قالب در اینجا بسیار شبیه به نحوه ایجاد ما است کلاس‌ها یا فرم‌ها را در خود جنگو مدل کنید. ما مشخص می‌کنیم که از کدام مدل استفاده کنیم و فیلدهای خاص روی آن را که می‌خواهیم در معرض نمایش بگذاریم. به یاد داشته باشید که id به طور خودکار توسط جنگو ایجاد می‌شود بنابراین ما این کار را نکردیم باید آن را در مدل Todo خود تعریف کنیم، اما ما از آن در نمای جزئیات خود استفاده خواهیم کرد.

و همین است. Django REST Framework اکنون داده‌های ما را به شکل جادویی به JSON تبدیل می‌کند و فیلدهای شناسه، عنوان و بدن مدل Todo را نشان می‌دهد.

آخرین کاری که باید انجام دهیم این است که فایل views.py خود را پیکربندی کنیم

Views

در جنگو سنتی از نماها برای سفارشی کردن داده‌هایی که به قالب‌ها ارسال شود استفاده می‌شود. در Django REST Framework view ها همین کار را انجام می‌دهند اما برای داده‌های سریالی ما.

سینتکس نماهای چارچوب جنگو REST عمداً کاملاً شبیه نماهای جنگو معمولی است و درست مانند جنگو معمولی، جنگو REST با نماهای عمومی برای موارد استفاده رایج ارائه می‌شود. این چیزی است که ما در اینجا استفاده خواهیم کرد.

فایل todos/views.py را به صورت زیر به روزرسانی کنید:

```
# todos/views.py
from rest_framework import generics
from .models import Todo
from .serializers import TodoSerializer

class ListTodo(generics.ListAPIView):
    queryset = Todo.objects.all()
    serializer_class = TodoSerializer

class DetailTodo(generics.RetrieveAPIView):
    queryset = Todo.objects.all()
    serializer_class = TodoSerializer
```

در بالا، نماهای ژنریک چارچوب Django REST و هر دو فایل serializers.py و models.py خود را وارد می‌کنیم.

از فایل todos/urls.py به یاد بیاورید که ما دو مسیر و در نتیجه دو نمای مجزا داریم. ما از [ListAPIView](#) برای نمایش همه کارها و از [RetrieveAPIView](#) برای نمایش یک نمونه مدل استفاده خواهیم کرد.

خوانندگان زیرک متوجه خواهند شد که در اینجا مقداری افزونگی در کد وجود دارد. ما اساساً و queryset_class را برای هر نما تکرار می‌کنیم، حتی اگر نمای عمومی توسعه یافته متفاوت باشد. بعداً در این کتاب با مجموعه‌های نمایش و روت‌های آشنایی شویم که به این مشکل رسیدگی می‌کنند و به ما امکان می‌دهند همان نماهای API و URL‌ها را با کد بسیار کمتر ایجاد کنیم.

اما در حال حاضر کار ما تمام شده است! API ما آماده مصرف است. همانطور که می بینید، تنها تفاوت واقعی serializers.py و Django REST Framework باشد. Django REST Framework اضافه کنیم و نیازی به فایل قالب نداریم. در غیر این صورت فایل‌های views.py و urls.py به روشی مشابه عمل می‌کنند.

صرف API

صرف سنتی API یک چالش بود. به سادگی تجسم خوبی برای تمام اطلاعات موجود در بدنه و هدر یک پاسخ یا درخواست HTTP مشخص وجود نداشت.

در عوض، بیشتر توسعه دهنده‌گان از یک سرویس گیرنده HTTP خط فرمان مانند [cURL](#) استفاده کردند که در فصل قبل یا [HTTPie](#) دیدیم.

در سال 2012، محصول نرم‌افزار شخص ثالث [Postman](#) راه‌اندازی شد و اکنون توسط میلیون‌ها توسعه‌دهنده در سراسر جهان که می‌خواهند روشی بصری و غنی برای تعامل با API‌ها داشته باشند، از آن استفاده می‌کنند.

اما یکی از شگفت انگیزترین چیزها در مورد Django REST Framework این است که با یک API قابل مرور قدرتمند عرضه می‌شود که می‌توانیم بلافاصله از آن استفاده کنیم. اگر متوجه شدید که در مورد مصرف API به شخصی سازی بیشتری نیاز دارید، ابزارهایی مانند Postman در دسترس هستند. اما اغلب API داخلی بیش از اندازه کافی است.

قابل مرور API

بیایید اکنون از API قابل مرور برای تعامل با داده‌های خود استفاده کنیم. مطمئن شوید که سرور محلی در حال اجرا است.

```
(backend) $ python manage.py runserver
```

سپس به [/http://127.0.0.1:8000/api](http://127.0.0.1:8000/api) بروید تا نقطه پایانی بازدیدهای لیست API ما را ببینید.

List Todo – Django REST frame +

Guest :

Django REST framework WSV

List Todo

OPTIONS GET

GET /api/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
  {  
    "id": 1,  
    "title": "1st todo",  
    "body": "Learn Django properly."  
  },  
  {  
    "id": 2,  
    "title": "Second item",  
    "body": "Learn Python."  
  },  
  {  
    "id": 3,  
    "title": "Learn HTTP",  
    "body": "It's important."  
  }]  
]
```

این صفحه سه کاری را که قبلاً در مدل پایگاه داده ایجاد کردیم را نشان می‌دهد. نقطه پایانی API به عنوان یک مجموعه شناخته می‌شود زیرا چندین مورد را نشان می‌دهد.

کارهای زیادی می‌توانیم با API قابل مرور خود انجام دهیم. برای شروع، بباید نمای خام JSON را ببینیم - آنچه در واقع از طریق اینترنت منتقل می‌شود. بر روی دکمه "GET" در گوش سمت راست بالا کلیک کنید و JSON را انتخاب کنید.

127.0.0.1:8000/api/?format=json +

Guest :

127.0.0.1:8000/api/?format=json

```
[{"id":1,"title":"1st todo","body":"Learn Django properly."}, {"id":2,"title":"Second item","body":"Learn Python."}, {"id":3,"title":"Learn HTTP","body":"It's important."}]
```

اگر به صفحه نمایش لیست ما در <http://127.0.0.1:8000/api/> برگردید، می‌بینیم که اطلاعات بیشتری وجود دارد. به یاد داشته باشید که فعل HTTP GET برای خواندن داده‌ها استفاده می‌شود در حالی که POST برای به روز رسانی یا ایجاد داده‌ها استفاده می‌شود.

در زیر "List Todo" می‌گویید که ما یک GET در این نقطه پایانی انجام داده‌ایم. زیر آن می‌گویید HTTP 200 OK که کد وضعیت ماست، همه چیز کار می‌کند. بسیار مهم در زیر آن نشان می‌دهد: ALLOW: GET, HEAD, OPTIONS. توجه داشته باشید که شامل POST نمی‌شود زیرا این یک نقطه پایانی فقط خواندنی است، ما فقط می‌توانیم GET را انجام دهیم.

ما همچنین یک نمای DetailTodo برای هر مدل جداگانه ایجاد کردیم. این به عنوان یک نمونه شناخته می‌شود و در [قابل مشاهده است.](http://127.0.0.1:8000/api/1)

Django REST framework

List Todo / Detail Todo

Detail Todo

OPTIONS GET

GET /api/1/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{ "id": 1, "title": "1st todo", "body": "Learn Django properly." }
```

همچنین می‌توانید به نقاط پایانی برای موارد زیر بروید:

- <http://127.0.0.1:8000/api/2>
- <http://127.0.0.1:8000/api/3>

CORS

آخرین مرحله‌ای که باید انجام دهیم وجود دارد و آن به [اشتراک گذاری منابع متقطع CORS](#) است. هر زمان که یک کلاینت با یک API میزبانی شده در دامنه دیگری (yoursite.com) در مقابل localhost:3000 (یا پورت localhost:8000) در مقابل localhost:8000 تعامل داشته باشد، مشکلات امنیتی احتمالی وجود دارد.

به طور خاص، CORS از سرور می‌خواهد که سربرگ‌های HTTP خاصی را شامل شود که به مشتری اجازه می‌دهد تعیین کند که آیا درخواست‌های بین دامنه‌ای مجاز هستند یا نه.

اختصاصی ارتباط برقرار می‌کند که در پورت دیگری برای توسعه محلی و front-end ما با یک Django API back-end پس از استقرار در دامنه دیگری قرار دارد.

ساده‌ترین راه برای رسیدگی به این مورد - و آنچه توسط [Django REST Framework](#) توصیه می‌شود - استفاده از میان‌افزار است که به طور خودکار هدرهای HTTP مناسب را بر اساس تنظیمات ما شامل می‌شود.

بسته‌ای که ما استفاده خواهیم کرد [django-cors-headers](#) است که می‌تواند به راحتی به پروژه موجود ما اضافه شود.

ابتدا با Control+c از سرور خود خارج شوید و سپس هدرهای django-cors را با Pipenv نصب کنید.

```
(backend) $ pipenv install django-cors-headers==3.4.0
```

- corsheader کنید INSTALLED_APPS اضافه هارا به
- CorsMiddleware در CommonMiddleWare را بالاتر از MIDDLEWARE اضافه کنید
- یک CORS_ORIGIN_WHITELIST ایجاد کنید

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 3rd party
    'rest_framework',
    'corsheaders', # new

    # Local
    'todos',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware', # new
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

# new
CORS_ORIGIN_WHITELIST = (
    'http://localhost:3000',
    'http://localhost:8000',
)
```

بسیار مهم است که corsheaders.middleware.CorsMiddleware در مکان مناسب ظاهر شود. این بالاتر از MIDDLEWARE در تنظیمات django.middleware.common.CommonMiddleware است زیرا میان افزارها از بالا به پایین بارگذاری می شوند. همچنین توجه داشته باشید که ما دو دامنه را در لیست سفید قرار داده ایم: localhost:3000 و localhost:8000. اولی پورت پیشفرض React است که در فصل بعدی از آن برای front-end خود استفاده خواهیم کرد. دومی پورت پیشفرض جنگو است.

شما همیشه باید برای پروژه های جنگو خود تست بنویسید. مقدار کمی از زمان صرف شده از قبل باعث صرفه جویی در زمان و تلاش زیادی برای خطاهای اشکال زدایی می شود. بیایید دو تست اساسی اضافه کنیم تا تأیید کنیم که عنوان و محتوای متن همانطور که انتظار می رود رفتار می کنند. فایل todos/tests.py را با موارد زیر پر کنید:

```
# todos/tests.py
from django.test import TestCase
from .models import Todo

class TodoModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        Todo.objects.create(title='first todo', body='a body here')

    def test_title_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.title}'
        self.assertEqual(expected_object_name, 'first todo')

    def test_body_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.body}'
        self.assertEqual(expected_object_name, 'a body here')
```

این از کلاس [TestCase](#) داخلی جنگو استفاده می کند. ابتدا داده های خود را در setUpTestData تنظیم می کنیم و سپس دو تست جدید می نویسیم. سپس با دستور python manager.py تست ها را اجرا کنید.

```
(backend) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

..
-----
Ran 2 tests in 0.002s
OK
Destroying test database for alias 'default'...
```

و تمام! بکاند ما اکنون کامل شده است. مطمئن شوید که سرور در حال اجرا است زیرا در فصل بعدی از آن استفاده خواهیم کرد.

```
(backend) $ python manage.py runserver
```

نتیجه گیری

با حداقل مقدار کد Django REST Framework به ما این امکان را می دهد که یک API جنگو را از ابتداء ایجاد کنیم. تنها قطعاتی که از جنگو سنتی نیاز داشتیم یک فایل models.py و مسیرهای urls.py ما بود. فایلهای views.py و serializers.py کاملاً مختص چارچوب REST جنگو بودند.

برخلاف مثال ما در فصل قبل، ما هیچ صفحه وب برای این پروژه ایجاد نکردیم زیرا هدف ما فقط ایجاد یک API بود. با این حال در هر نقطه ای در آینده، ما به راحتی می توانیم! این فقط نیاز به افزودن یک نمای جدید، URL و یک الگو دارد تا مدل پایگاه داده موجود ما را نشان دهد.

یک نکته مهم در این مثال این است که ما هدرهای CORS را اضافه کردیم و به صراحة فقط دامنه های localhost:8000 و localhost:3000 را برای دسترسی به API خود تنظیم کردیم. تنظیم صحیح هدرهای CORS موضوعی آسان است که هنگام شروع ساختن APIها در مورد آن سردرگم می شود. پیکربندی های بسیار بیشتری وجود دارد که می توانیم انجام دهیم و بعداً انجام خواهیم داد، اما در پایان روز ایجاد API های جنگو در مورد ساخت یک مدل، نوشتن برخی از مسیرهای URL، و سپس افزودن کمی جادوی ارائه شده توسط سریال سازها و نماهای جنگو REST Framework است. فصل بعدی یک React front-end می سازیم و آن را به باطن Todo API خود متصل می کنیم.

فصل 4 : Todo React Front-end

به منظور برقراری ارتباط با برنامه دیگر یک API وجود دارد. در این فصل ما از Todo API قبلی از طریق فرانت اند استفاده خواهیم کرد. درنتیجه می توانید ببینید که چطور در عمل همه چیز در کنار هم کار می کنند.

در اینجا از React استفاده کرده ام به این دلیل که اخیراً معروف ترین کتابخانه‌ی فرانت اند جاوااسکریپت می‌باشد اما تکنولوژی‌هایی که در اینجا توصیف می‌شوند با هر فریم ورک معروف فرانت اند دیگری مثل Vue, Angular یا Ember هم کار می‌کنند. حتی با اپ‌های موبایلی iOS یا اندروید، اپ‌های دسکتاپ، یا هر چیز دیگری هم کار می‌کنند. فرآیند اتصال به یک بک‌نند API به طرز قابل توجهی مشابه است.

اگر درگیر این مسئله شده‌اید و یا می‌خواهید بیشتر بدانید که واقعاً چه اتفاقی با React می‌افتد، [official tutorial](#) را بررسی نمایید.

نصب Node

با پیکره بندی React به عنوان فرانت اند خودمان شروع می‌کنیم. در ابتدا یک کنسول خط فرمان (کامند لاین) جدید را باز می‌کنیم پس حالا دو کنسول باز هستند. این مهم هست. چون باید todo بک اندی که در فصل قبل راه اندازی کردیم هم چنان در سرور محلی درحال اجرا باشد. از کنسول دوم نیز برای ساخت و اجرای فرانت اند React روی یک پورت محلی مجزا استفاده خواهیم کرد. اینگونه ما به تقلید اینکه تنظیمات پروداکشن سفارشی شده و استقرار یافته‌ی front/back چگونه باشد، می‌پردازیم.

در خط فرمان جدید دوم [NodeJS](#) که یک موتور زمان اجرای (ران تایم) جاوااسکریپتی است را نصب می‌کنیم. این موجب می‌شود بتوانیم جاوااسکریپت را در محیطی خارج از یک مرورگر وب اجرا نماییم.

روی یک کامپیوتر مک می‌توانیم [Django for Beginners](#) استفاده کنیم که اگر دستورات [Homebrew](#) را برای پیکره بندی کامپیوتر محلی خودتان اجرا کرده باشد، باید نصب باشد.

خط فرمان

```
$ brew install node
```

روی ویندوز روش‌های زیادی برای نصب وجود دارد که معروف ترین آن‌ها استفاده از [nvm-windows](#) است که در مخزن گیت آن دستورات نصب کامل و به روزی در این باره وجود دارد.

اگر در محیط لینوکس هستید از [nvm](#) استفاده کنید. تا این لحظه که کتاب نوشته می‌شود دستور با استفاده از URL اجرا می‌گردد.

خط فرمان

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

یا از wget استفاده کنید.

```
$ wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

سپس اجرا کنید:

خط فرمان

```
$ command -v nvm
```

حال کنسول خط فرمان خود را بسته و آن را دوباره باز کنید تا نصب کامل شود.

نصب React

برای اجرای سریع یک پروژه React جدید از پکیج بسیار خوب [create-react-app](#) استفاده می کنیم. این درواقع دیگر بخار پروژه ما را می سازد و تمامی وابستگی های مورد نیاز پروژه را با یک دستور نصب می کند!

برای نصب react از [npm](#) استفاده می کنیم. npm ابزار مدیریت پکیج جاوااسکریپت است. npm همانند pipenv در پایتون مدیریت و نصب چندین پکیج نرم افزاری را ساده تر می کند. نسخه های اخیر npm، شامل [npx](#) هم هستند. npx یک روش پیشرفته برای نصب پکیج ها به صورت محلی بدون آلودگی فضای نام سراسری است (polluting the global namespace). این روش پیشنهادی برای نصب React می باشد که ما در اینجا از آن استفاده خواهیم کرد.

با ورود به Desktop (اگر روی مک هستید) و سپس فolder todo از اینکه در مسیر درستی قرار گرفته اید، اطمینان حاصل نمایید.

خط فرمان

```
$ cd ~/Desktop
$ cd todo
```

یک React App جدید به نام frontend ایجاد کنید.

خط فرمان

```
$ npx create-react-app frontend
```

حال باید ساختار مسیر شما به صورت زیر باشد.

دیاگرام

```
todo
|   └─frontend
|       └─React...
```

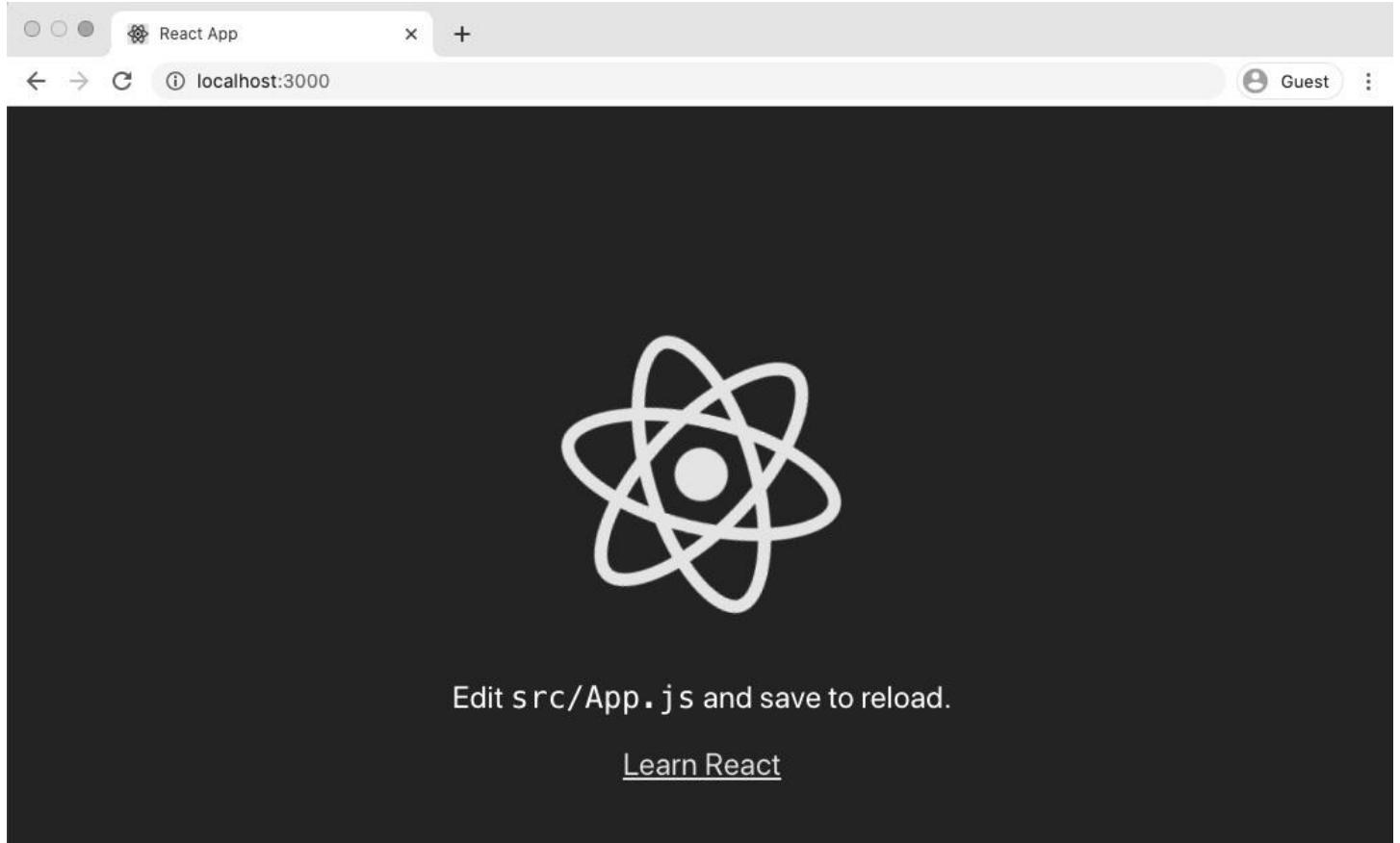
```
|   └── backend  
|       └── Django...
```

به پروژه fronted خود برگشته و npm را با دستور React App اجرا می کنیم.

خط فرمان

```
$ cd frontend  
$ npm start
```

اگر وارد <http://localhost:3000> شوید صفحه خانه پیشفرض create-react-app را مشاهده خواهید کرد.



ماک دیتا

اگر به اندپوینت API بازگردید می توانید فایل خام JSON را در مرورگر به آدرس <http://127.0.0.1:8000/api/?format=json> ملاحظه کنید.

Code

```
[  
{  
    "id":1,  
    "title":"1st todo",  
    "body":"Learn Django properly."  
},
```

```
{
  "id":2,
  "title":"Second item",
  "body":"Learn Python."
},
{
  "id":3,
  "title":"Learn HTTP",
  "body":"It's important."
}
]
```

این مورد زمانی که یک درخواست GET به اندپوینت API صادر شود، برگردانده می شود. در نهایت از API به صورت مستقیم استفاده خواهیم کرد اما یک گام اولیه خوب این است که ابتدا دیتا را ماک کنیم (ایجاد داده های ساختگی) و سپس فرآیند API call را پیکره بندی نماییم.

تنها فایلی که نیاز به بروزرسانی آن در React app src/App.js هست، فایل data را در متغیری به نام list که درواقع یک آرایه با 3 مقدار است، ماک می نماییم.

کد

```
// src/App.js
import React, { Component } from 'react';

const list = [
{
  "id":1,
  "title":"1st todo",
  "body":"Learn Django properly."
},
{
  "id":2,
  "title":"Second item",
  "body":"Learn Python."
},
{
  "id":3,
  "title":"Learn HTTP",
  "body":"It's important."
}
]
```

در مرحله بعدی آرایه list را در state کامپوننت خودمان بارگذاری می کنیم و سپس از متده آرایه ای جاوااسکریپت map() برای نمایش تمامی آیتم ها استفاده می نماییم.

عمدا سریع از این بخش عبور می کنم، پس اگر قبل از react استفاده نکرده اید، فقط کد را کپی کنید تا با این کار ببینید که اتصال یک فرانت اند react به بک اند جنگو چگونه انجام می شود. این کد کاملی است که باید آن را در فایل src/App.js قرار دهید.

```
// src/App.js
import React, { Component } from 'react';

const list = [
{
  "id":1,
  "title":"1st todo",
  "body":"Learn Django properly."
},
{
  "id":2,
  "title":"Second item",
  "body":"Learn Python."
},
{
  "id":3,
  "title":"Learn HTTP",
  "body":"It's important."
}
]

class App extends Component {
  constructor(props) {
    super(props);
    this.state = { list };
  }
  render() {
    return (
      <div>
        {this.state.list.map(item => (
          <div key={item.id}>
            <h1>{item.title}</h1>
            <p>{item.body}</p>
          </div>
        ))}
      </div>
    );
  }
}

export default App;
```

آرایه list را در state اپ کامپوننت مورد نظر بارگذاری نموده ایم، پس از آن از map برای جستجو در هر آیتم از لیستی که title و body را نشان می دهد استفاده می کنیم. همچنین id را به عنوان کلیدی که یک نیازمندی خاص react می باشد، به این مقادیر اضافه کرده ایم، id به طور خودکار توسط جنگو به هر فیلد دیتابیسی اضافه شده است. حال باید todo هایمان را که در صفحه اصلی <http://localhost:3000> لیست شده اند بدون نیاز به رفرش نمودن صفحه ببینید.



1st todo

Learn Django properly.

Second item

Learn Python.

Learn HTTP

It's important.

توجه کنید: اگر زمانی را صرف کار با react کنید احتمالاً جایی این پیغام خطرا مشاهده نمایید:

```
sh: react-scripts: command not found while running npm start .
```

نگران نباشید. این یک مشکل بسیار بسیار رایج در توسعه جاوااسکریپت است. راه حل معمول این است که دستور `npm install` و پس از آن دوباره `npm start` را اجرا کنید. اگر این راه حل جواب نداد، فolder `node_modules` را پاک کرده و `npm install` را اجرا کنید. این کار در 99% موقع مشکل را حل می کند. به توسعه جاوااسکریپت مدرن خوش آمدید (:

فریم ورک رست جنگو +

حال بباید به جای استفاده از `mock data` در متغیر `list`، به طور واقعی به `Todo API` خودمان متصل شویم. در کنسول خط فرمان دیگر، سرور جنگو در حال اجراست و می دانیم که اندپوینت API که تمامی `todo` ها را لیست نموده در <http://127.0.0.1:8000/api> قرار دارد. پس ما باید یک درخواست GET را به آن صادر کنیم.

برای ایجاد درخواست های HTTP دو روش معروف وجود دارد: با استفاده از `built-in Fetch API` و یا با استفاده از `axios` که با چندین فیچر اضافی همراه است. در این مثال از `axios` استفاده خواهیم کرد. که روی خط فرمان در حال اجرا است را با دستور `Control+c` متوقف کنید. سپس `axios` را نصب نمایید.

خط فرمان

```
$ npm install axios
```

دوباره npm start را با دستور React app راه اندازی کنید.

خط فرمان

```
$ npm start
```

سپس در ادیتور متنی خود قبل از فایل App.js باید `axios` را ایمپورت و فراخوانی کنید.

```
// src/App.js
import React, { Component } from 'react';
import axios from 'axios'; // new
...
```

دو گام باقی مانده است. اول اینکه از axios برای درخواست GET استفاده خواهیم کرد. برای این هدف می توانیم یک تابع getTodos اختصاصی شده بسازیم.

دوم اینکه می خواهیم مطمئن شویم که API call در زمان درستی در طول چرخه حیات React صادر می شود. درخواست های HTTP باید با استفاده از componentDidMount ساخته شوند تا ما getTodos را آن جا فراخوانی کنیم.

از آنجایی که دیگر به list mock نیاز نخواهد شد، می توانیم آن را پاک کنیم. حال فایل کامل App.js به صورت زیر خواهد بود:

کد

```
// src/App.js
import React, { Component } from 'react';
import axios from 'axios'; // new

class App extends Component {
  state = {
    todos: []
  };

  // new
  componentDidMount() {
    this.getTodos();
  }

  // new
  getTodos() {
    axios
      .get('http://127.0.0.1:8000/api/')
      .then(res => {
        this.setState({ todos: res.data });
      })
      .catch(err => {
        console.log(err);
      });
  }

  render() {
    return (
      <div>
        {this.state.todos.map(item => (
          <div key={item.id}>
```

```

        <h1>{item.title}</h1>
        <span>{item.body}</span>
    </div>
)
})
);
}

export default App;

```

اگر دوباره به <http://localhost:3000> نگاهی بیاندازید می بینید با وجود اینکه دیگر hardcoded data نداریم اما صفحه تغییری نکرده و همان است. از حالا تمامی این ها از طریق اندپوینت API و درخواست ما می آیند.



حال این روش با react انجام شده است!

نتیجه گیری

ما Django backend API خود را به فرانت اند react متصل کرده ایم. حتی بهتر از آن، می توانیم در آینده فرانت خودمان را بروز نماییم یا با تغییر نیازمندی های پروژه آن را کاملاً عوض کنیم.

بدین منظور اتخاذ یک روش مبتنی بر API-first بهترین روش برای آینده سایت شماست. ممکن است در ابتدا کار بیشتری برای انجام لازم باشد اما انعطاف پذیری بیشتری فراهم می‌باشد. در فصل های بعدی API های خود را بهبود خواهیم داد تا درخواست های HTTP بیشتری مثل POST (افزودن todo های جدید)، PUT (به روز رسانی todo های موجود) و DELETE (حذف todo ها) را پشتیبانی نماید.

در فصل بعدی شروع به ساخت یک API CRUD(Create, Read, Update, Delete) مقاوم که تمامی عملیات Blog را پشتیبانی نماید، خواهیم کرد و پس از آن احراز هویت کاربر را به آن می افزاییم تا کاربران بتوانند از طریق API ما وارد سایت شده، از آن خارج شوند و با حساب کاربری خود ثبت نام نمایند.

فصل 5: ای پی آی و بلاگ

پروژه بعدی ما یک Blog API با استفاده از مجموعه Django REST Framework ایجاد، خواندن، به روزرسانی، حذف) است. ما همچنین ویوست ها، روت‌ها و اسناد را بررسی خواهیم کرد.

در این بخش ما بخش اصلی API را می‌سازیم. دقیقاً مانند پروژه کتابخانه و TODO API ساخت پروژه را ابتدا با جنگو شروع کرده و سپس آن را به چهارچوب Django Rest Framework اضافه می‌کنیم. تفاوت اصلی این است که نقاط پایانی API ما از ابتدا از CRUD پشتیبانی می‌کنند؛ و همانطور که خواهید دید فریم ورک REST جنگو نیز کاملاً مشابه همین کار را بدون مشکل انجام میدهد.

تنظیمات اولیه

تنظیمات اولیه مانند قبل است. در ابتدا به دایرکتوری اصلی پروژه رفته و برنامه‌ای با نام blogapi می‌سازیم. سپس جنگو را در محیط مجازی خود نصب می‌کنیم و یک پیکربندی برای پروژه جدید جنگو خود و برنامه بلاگ برای پست‌ها انجام می‌دهیم.

Command Line

```
$ cd ~/Desktop && cd code
$ mkdir blogapi && cd blogapi
$ pipenv install django~=3.1.0
$ pipenv shell
(blogapi) $ django-admin startproject config .
(blogapi) $ python manage.py startapp posts
```

هنگامی که برنامه بلاگ را در پروژه ایجاد کردیم باید به جنگو نیز حضور این برنامه را اعلام کنیم. به این منظور برنامه را در قسمت config/settings.py در فایل INSTALLED_APPS اضافه می‌کنیم.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Local
    'posts', # new
]
```

اکنون migrate را برای اولین بار اجرا کنید تا پایگاه داده ما با تنظیمات پیش‌فرض جنگو و برنامه جدید همگام شود.

Command Line

```
(blogapi) $ python manage.py migrate
```

مدل

مدل پایگاه داده ما دارای پنج فیلد است: نویسنده، عنوان، بدن، create_at و updated_at. ما می‌توانیم از مدل کاربر داخلی جنگو به عنوان کاربر اصلی استفاده کنیم، مشروط بر اینکه آن را در خط دوم کد خود ایمپورت کنیم.

Code

```
# posts/models.py
from django.db import models
from django.contrib.auth.models import User

class Post(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=50)
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

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

توجه داشته باشید که ما همچنین در حال تعریف این هستیم که نمایش str مدل باید چگونه باشد که بهترین روش جنگو برای نمایش تیتر پست‌ها در صفحه ادمین جنگو است.

اکنون پایگاه داده خود را با ایجاد یک فایل migrate جدید و سپس اجرای migrate برای همگام سازی پایگاه داده با تغییرات مدل خود، به روزرسانی کنید.

Command Line

```
(blogapi) $ python manage.py makemigrations posts
Migrations for 'posts':
  posts/migrations/0001_initial.py
    - Create model Post
(blogapi) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, posts, sessions
Running migrations:
  Applying posts.0001_initial... OK
```

اکنون میخواهیم داده‌های خود را در برنامه داخلی ادمین جنگو مشاهده کنیم. برای این منظور ویو را به شکل زیر به فایل posts/admin.py اضافه می‌کنیم.

```
# posts/admin.py
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

در ادامه برای ایجاد یک ابر کاربر برای دسترسی به پنل ادمین دستور زیر را تایپ کرده و ورودی های مورد نظر را به آن می دهیم.

Command Line

```
(blogapi) $ python manage.py createsuperuser
```

حال وب سرور لوکال را با دستور زیر اجرا می کنیم.

Command Line

```
(blogapi) $ python manage.py runserver
```

سپس به آدرس <http://127.0.0.1:8000/admin> رفته و با اطلاعات ابرکاربر خود به داشبورد ادمین وارد می شویم. روی دکمه "+ افزودن" در کنار پست ها کلیک کنید و یک پست و بلاگ جدید ایجاد کنید. در کنار "نویسنده" یک منوی کشویی وجود خواهد داشت که دارای حساب کاربری ابرکاربر شما است (نام کاربر من wsv است). مطمئن شوید که یک نویسنده انتخاب شده باشد. سپس عنوان و محتوای متن را اضافه کنید سپس روی دکمه "ذخیره" کلیک کنید.

Add post | Django site admin

127.0.0.1:8000/admin/posts/post/add/

Django administration

Welcome, **WSV**. View site / Change password / Log out

Home > Posts > Posts > Add post

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add
Users	+ Add
POSTS	
Posts	+ Add

Add post

Author: [+ Add](#)

Title:

Body:

This is my first blog post.

[Save and add another](#) [Save and continue editing](#) **SAVE**

در ادامه شما به صفحه ای هدایت می شوید که همه پست های موجود در وبلاگ را نمایش می دهد.

Select post to change | Django

127.0.0.1:8000/admin/posts/post/

Django administration

Welcome, **WSV**. View site / Change password / Log out

Home > Posts > Posts

AUTHENTICATION AND AUTHORIZATION	
Groups	+ Add
Users	+ Add
POSTS	
Posts	+ Add

The post "Hello world!" was added successfully.

Select post to change

Action: [Go](#) 0 of 1 selected

POST

Hello world!

1 post

[ADD POST +](#)

تست ها

در ادامه یک تست ابتدایی برای مدل پست وبلاگ می نویسیم. در این تست اطمینان حاصل می شود که کاربر وارد شده به سیستم بتواند یک پست با عنوان و بدنه ایجاد کند.

```

# posts/tests.py
from django.test import TestCase
from django.contrib.auth.models import User
from .models import Post

class BlogTests(TestCase):

    @classmethod
    def setUpTestData(cls):
        # Create a user
        testuser1 = User.objects.create_user(
            username='testuser1', password='abc123')
        testuser1.save()

        # Create a blog post
        test_post = Post.objects.create(
            author=testuser1, title='Blog title', body='Body content...')
        test_post.save()

    def test_blog_content(self):
        post = Post.objects.get(id=1)
        author = f'{post.author}'
        title = f'{post.title}'
        body = f'{post.body}'
        self.assertEqual(author, 'testuser1')
        self.assertEqual(title, 'Blog title')
        self.assertEqual(body, 'Body content...')


```

برای اطمینان از کارکرد صحیح تست با فشردن Control+c سرور لوكال را بسته و سپس تست ها را اجرا می کنیم.

Command Line

```
(blogapi) $ python manage.py test
```

پس از اجرا باید خروجی های مانند زیر مشاهده شود که نشان دهنده کارکرد صحیح همه قسمت های برنامه است.

Command Line

```

(blogapi) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----
Ran 1 test in 0.105s

OK
Destroying test database for alias 'default'...


```

در این زمان قسمت معمول ای پی ای جنگو به پایان رسیده است و زمان آن است که مدل و دیتاهاي را به دیتابیس اضافه کنیم. بنابراین زمان آن است که از جنگو رست فریمورک برای انتقال دیتاهاي مدل به ای پی آی خود بهره ببریم.

جنگو رست فریمورک

همانطور که قبلًا دیده ایم، **Django REST Framework** وظیفه تبدیل مدل های پایگاه داده ما به یک API RESTful را بر عهده دارد. سه مرحله اصلی برای این فرآیند وجود دارد:

- `urls.py` فایل مشخص کننده مسیر ها
- `serializers.py` فایل برای تبدیل دیتا به جیسون
- `views.py` فایل برای پیاده سازی منطق هر کدام از اندپوینت ها

در کامندهاین با استفاده از `pipenv` جنگو رست فریمورک را به شکل زیر نصب می کنیم.

Command Line

```
(blogapi) $ pipenv install djangorestframework~=3.11.0
```

سپس آن را در قسمت `INSTALLED_APPS` در فایل `config/settings.py` اضافه می کنیم. همچنین بهتر است برای جنگو رست فریمورک در قسمت پرمیشن ها در ستینگ `AllowAny` تعريف شده است را مشخص کنیم. این قسمت در فصل بعدی آپدیت خواهد شد.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 3rd-party apps
    'rest_framework', # new

    # Local
    'posts.apps.PostsConfig',
]

# new
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

=اکنون ما باید `url` ها ، `View` ها و سریالایزرهاي برنامه را ایجاد کنیم.

URL ها

باید با مسیرهای URL برای مکان های واقعی endpoint ها شروع کنیم. ابتدا فایل `url.py` را با وارد کردن در خط دوم در آدرس `/api/v1/` برای پست های برنامه آپدیت می کنیم.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')), # new
]
```

این مسئله تمرین مناسبی است برای اینکه همیشه ورژن API ها را به صورت `v2` ، `v1/` ، ... نگه داریم. به دلیل اینکه ممکن است زمانی که تغییرات حجمی بر روی API ها اعمال میکنیم باعث باغ یا لگ شود و نیاز به ورژن های قبلی داشته باشیم. به این ترتیب میتوانیم در زمان لانچ ورژن جدید API از ورژن های قبلی نیز ساپورت کنیم. زیرا ممکن است سرویس برخی از مشتریان بر روی ورژن های قبلی لانچ شده باشد.

توجه داشته باشید که از آنجایی که تنها برنامه ما در این مرحله پست است، می توانیم آن را مستقیماً در اینجا قرار دهیم. اگر ما چندین برنامه در یک پروژه داشتیم، ممکن است منطقی تر باشد که یک برنامه اختصاصی api ایجاد کنیم و سپس همه مسیرهای آدرس API دیگر را در آن قرار دهیم. اما برای پروژه های اساسی مانند این، ترجیح می دهم از یک برنامه api که فقط برای مسیریابی استفاده می شود اجتناب کنم. در صورت نیاز همیشه می توانیم یک را بعداً اضافه کنیم.

در مرحله بعد فایل `url.py` برنامه را ایجاد میکنیم.

Command Line

```
(blogapi) $ touch posts/urls.py
```

سپس کد زیر را در آن فایل وارد میکنیم.

Code

```
# posts/urls.py
from django.urls import path
from .views import PostList, PostDetail

urlpatterns = [
    path('<int:pk>/', PostDetail.as_view()),
    path('', PostList.as_view()),
]
```

همه مسیرهای وبلاگ در /api/v1/ خواهد بود، بنابراین نمای PostList ما (که به زودی خواهیم نوشت) دارای رشته " در /api/v1/ خواهد بود و نمای PostDetail (همچنین باید نوشته شود) در /api/v1/ # کلید اصلی ورودی را نشان می‌دهد. به عنوان مثال، اولین پست وبلاگ دارای شناسه اولیه 1 است، بنابراین در مسیر /api/v1/1/ پست دوم در /api/v1/2/ و غیره خواهد بود.

سربالایزرهای

اکنون زمان سربالایز کردن است. یک فایل serializers.py جدید در برنامه پست‌های خود ایجاد کنید.

Command Line

```
(blogapi) $ touch posts/serializers.py
```

سربالایز نه تنها داده‌ها را به JSON تبدیل می‌کند، بلکه می‌تواند تعیین کند که کدام فیلدها را شامل یا حذف کند. در مورد ما، فیلد شناسه‌ای را که Django به طور خودکار به مدل‌های پایگاه داده اضافه می‌کنیم، اما فیلد «updated_at» را با درج نکردن آن در فیلدهای خود حذف می‌کنیم.

توانایی گنجاندن/حذف فیلدها در API ما یک ویژگی قابل توجه است. اغلب اوقات، یک مدل پایگاه داده زیربنایی، فیلدهای بسیار بیشتری نسبت به آنچه باید در معرض نمایش قرار گیرد، خواهد داشت. کلاس سربالایز قدرتمند کنترل این مورد را بسیار ساده می‌کند.

Code

```
# posts/serializers.py
from rest_framework import serializers
from .models import Post

class PostSerializer(serializers.ModelSerializer):

    class Meta:
        fields = ('id', 'author', 'title', 'body', 'created_at',)
        model = Post
```

در بالای فایل، کلاس سربالایزرهای Django REST Framework و مدل‌های خودمان را وارد کرده‌ایم. سپس یک PostSerializer ایجاد کردیم و یک کلاس Meta اضافه کردیم که در آن مشخص کردیم کدام فیلدها را شامل شود و به صراحت مدل را برای استفاده تنظیم کردیم. راه‌های زیادی برای سفارشی‌سازی سربالایزرهای ساخته شده وجود دارد، اما برای موارد استفاده رایج، مانند وبلاگ اولیه، این روش پیاده سازی شده کفایت می‌کند.

ویوهای

مرحله نهایی ساخت ویوهای برنامه است. Django Rest Framework چندین ویو عمومی مناسب و کمک کننده برای این کار در اختیار دارد. در حال حاضر ما از [ListAPIView](#) در کتابخانه و برنامه toolist برای ساخت اندپوینت‌های فقط قابل خواندن جهت گرفتن لیست مدل‌های ساخته شده در برنامه استفاده می‌کنیم. در برنامه Todos همچنین از [RetrieveAPIView](#) جهت ساخت اندپوینت تکی فقط خواندنی که مشابه نمای جزئیات در جنگو سنتی است نیز استفاده می‌کنیم.

برای API و بلاگ ما میخواهیم لیستی از همه پست های بلاگ را به صورت اندپوینت خواندنی-نوشتی در اختیار داشته باشیم که به این منظور از [ListCreateAPIView](#) استفاده می نماییم که بسیار شبیه به [ListView](#) است که قبلا از آن استفاده میکردیم با این تفاوت که به کاربر امکان نوشتن نیز میدهد. ما همچنین نیاز داریم که برای هر پست به صورت جدا امکانات شامل خواندن، ادیت کردن و حذف پست را نیز داشته باشیم. که به این منظوری کتابخونه عمومی ای برای این هدف با نام [RetrieveUpdateDestroyAPIView](#) در Django REST Framework وجود دارد که ما در این برنامه از آن استفاده میکنیم.

اکنون فایل views.py را به شکل زیر آپدیت میکنیم.

Code

```
# posts/views.py
from rest_framework import generics
from .models import Post
from .serializers import PostSerializer

class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

در بالا ب فایل کتابخانه generic همانند فایل مدل هاو سریالایزر خود وارد میکنیم. لیست پست ها از کتابخانه عمومی [ListCreateAPIView](#) و هر پست از [RetrieveUpdateDestroyAPIView](#) استفاده میکند.

بسیار شگفت انگیز است که تنها کاری که باید انجام دهیم این است که نمای کلی خود را به روز کنیم تا رفتار یک نقطه پایانی API را به طور اساسی تغییر دهیم. این مزیت استفاده از یک فریم ورک با ویژگی های کامل مانند Django REST Framework است. همه این قابلیت ها در دسترس هستند، آزمایش شده اند و فقط کار می کنند. به عنوان توسعه دهنده گان، مجبور نیستیم چرخ را در اینجا دوباره اختراع کنیم.

ای پی آی قابل مرور

سرور محلی را برای برقراری ارتباط با ای پی آی خود اجرا میکنیم.

Command Line

```
(blogapi) $ python manage.py runserver
```

برای دیدن لیست پست های ایجاد شده به آدرس <http://127.0.0.1:8000/api/v1> میرویم.

Post List – Django REST framework

127.0.0.1:8000/api/v1/ Guest

Django REST framework

Post List

OPTIONS GET

GET /api/v1/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
    {  
        "id": 1,  
        "author": 1,  
        "title": "Hello world!",  
        "body": "This is my first blog post.",  
        "created_at": "2020-07-29T17:37:03.350702Z"  
    }  
]
```

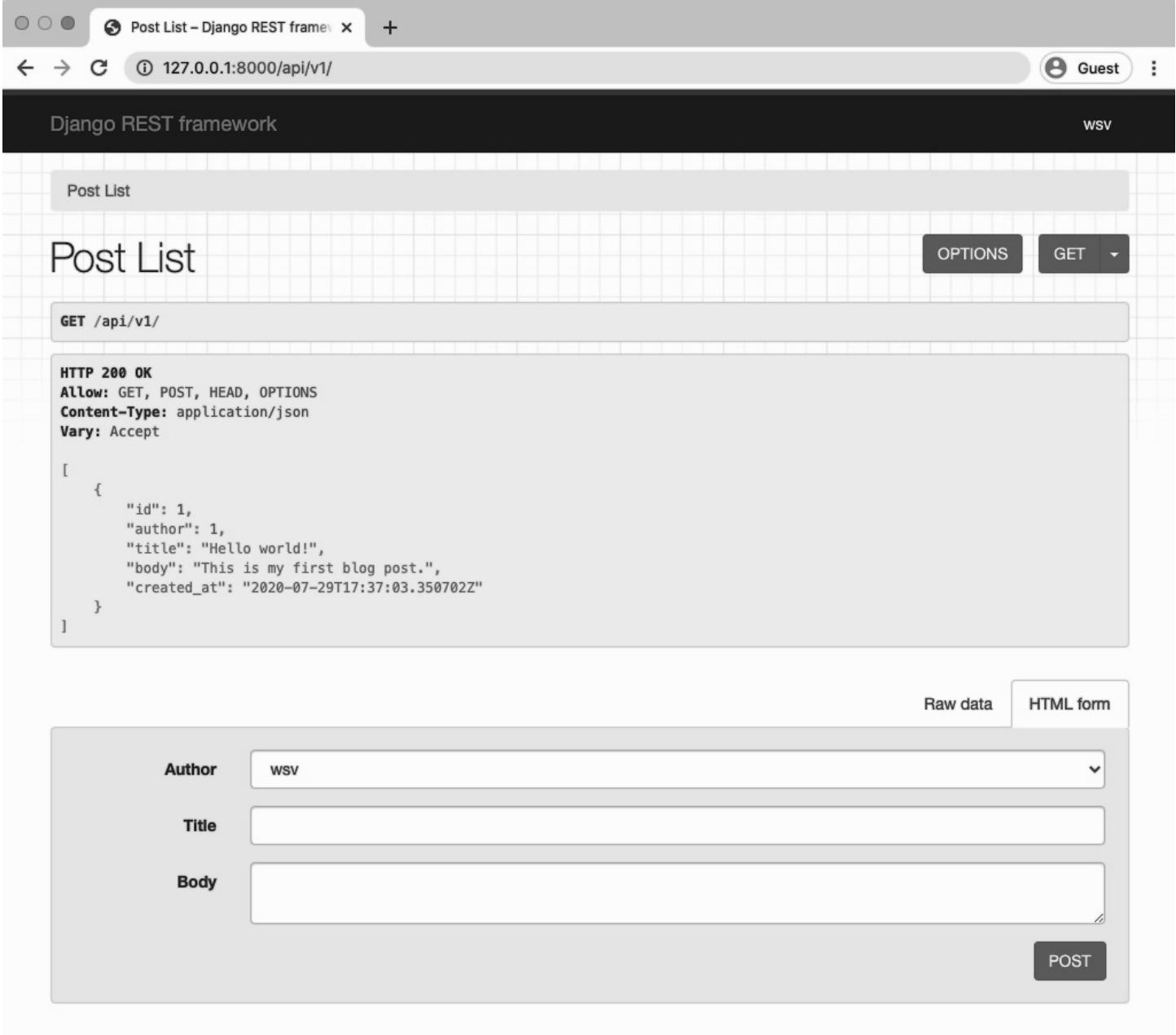
Raw data HTML form

Author: WSV

Title:

Body:

POST



این صفحه یک لیست از پست های موجود در ای پی آی را با فرمت جیسون نشان می دهد. توجه داشته باشید که هر دو متدهای GET و POST قابل اجرا هستند.

اکنون اجازه دهید تأیید کنیم که نقطه پایانی نمونه مدل ما - که به جای یک لیست از همه پست ها به یک پست مربوط می شود - وجود دارد.

به آدرس [/http://127.0.0.1:8000/api/v1/1](http://127.0.0.1:8000/api/v1/1) میرویم.

Post Detail – Django REST fram X +

127.0.0.1:8000/api/v1/1/ Guest :

Django REST framework

Post List / Post Detail

Post Detail

DELETE OPTIONS GET ▾

GET /api/v1/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "id": 1,  
    "author": 1,  
    "title": "Hello world!",  
    "body": "This is my first blog post.",  
    "created_at": "2020-07-29T17:37:03.350702Z"  
}
```

Raw data HTML form

Author	wsv
Title	Hello world!
Body	This is my first blog post.

PUT

در هدر مشاهده میشود که متدهای DELETE و ساپورت میشوند ولی نمیتوان از متدهای POST, PUT, PATCH استفاده کرد. در حقیقت شما میتوانید از فرم HTML موجود برای ایجاد تغییرات در فرم و از کلید DELETE برای حذف آن استفاده کنید.

اکنون زمان تست آن ها است. بباید موضوع را با یه متن اضافه در انتهای آن ویرایش کنیم. به این منظور مانند عکس زیر از دکمه PUT استفاده میکنیم.

Post Detail – Django REST fram X +

← → C ⓘ 127.0.0.1:8000/api/v1/1/ Guest ::

Django REST framework

Post List / Post Detail

Post Detail

DELETE OPTIONS GET ▾

PUT /api/v1/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "id": 1,  
    "author": 1,  
    "title": "Hello world! (edited)",  
    "body": "This is my first blog post.",  
    "created_at": "2020-07-29T17:37:03.350702Z"  
}
```

Raw data HTML form

Author	wsy
Title	Hello world! (edited)
Body	This is my first blog post.

PUT

The screenshot shows the Django REST framework's built-in browsable API interface. At the top, there are navigation buttons for back, forward, search, and user authentication (Guest). The URL is 127.0.0.1:8000/api/v1/1/. The main header says 'Post Detail'. Below it, there are buttons for DELETE, OPTIONS, and GET with a dropdown arrow. A large button labeled 'PUT' is followed by the URL '/api/v1/1/'. The response is a 200 OK status with headers: Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS; Content-Type: application/json; Vary: Accept. The JSON response body is shown as a code block. Below the response, there are two tabs: 'Raw data' and 'HTML form'. The 'HTML form' tab is selected, showing three input fields: 'Author' (set to 'wsy'), 'Title' (set to 'Hello world! (edited)'), and 'Body' (set to 'This is my first blog post.'). A large 'PUT' button is at the bottom right of the form area.

اکنون با کلیک روی دکمه بالای صفحه یا به صورت مستقیم از آدرس <http://127.0.0.1:8000/api/v1> به صفحه لیست تمامی پست ها برمی گردیم. در اینجا نیز مشاهده میشود که موضوع به درستی ویرایش شده است.

Post List – Django REST framework

127.0.0.1:8000/api/v1/ Guest

Django REST framework

Post List

OPTIONS GET

GET /api/v1/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
    {  
        "id": 1,  
        "author": 1,  
        "title": "Hello world! (edited)",  
        "body": "This is my first blog post.",  
        "created_at": "2020-07-29T17:37:03.350702Z"  
    }  
]
```

Raw data HTML form

Author: wsv

Title:

Body:

POST

The screenshot shows the Django REST framework's browsable API interface. At the top, there are navigation buttons for back, forward, and search, along with the URL '127.0.0.1:8000/api/v1/'. The title bar says 'Post List' and 'Django REST framework'. On the right, there are 'OPTIONS' and 'GET' buttons. Below this, a 'GET /api/v1/' button is shown. The main area displays the response for a GET request, which includes HTTP headers (Allow: GET, POST, HEAD, OPTIONS; Content-Type: application/json; Vary: Accept) and a JSON array containing one post object. The post has fields: id, author, title, body, and created_at. Below the response, there is a form with fields for Author (set to 'wsv'), Title, and Body. A 'POST' button is at the bottom of the form.

نتیجه گیری

ای پی آی وبلاگ ما در این مرحله به درستی کار میکند. اگرچه یک مشکل بزرگ در این برنامه وجود دارد: هر کسی می تواند پست دلخواهی را ادیت و یا حذف کند. به بیان دیگر، مجوزی برای این کار برای کاربران وجود ندارد. در فصل بعدی با هم میگیریم که چگونه مجوزهای لازم را برای تامین امنیت برنامه خود به کار ببریم.

امنیت بخش مهمی از هر وبسایت است اما با API ها دو چندان می‌شود. در حال حاضر API ما اجازه دسترسی کامل به هر شخصی را می‌دهد. هیچ محدودیتی وجود ندارد و هر کاربری هر کاری می‌تواند انجام دهد که بسیار خطرناک است. به عنوان مثال، یک کاربر ناشناس می‌تواند پستی را ایجاد کند، بخواند، تغییر دهد یا حذف کند. حتی آن پستی که خودش ایجاد نکرده است. مشخصاً ما چنین چیزی را نمی‌خواهیم.

البته Django REST Framework با تنظیمات ساده‌ی مجوزها همراه می‌باشد که ما می‌توانیم از آنها برای ایمن کردن API استفاده کنیم. این تنظیمات را می‌توان در سطح پروژه، در سطح نما یا در سطح هر مدل اعمال کرد.

در این فصل ابتدا یک کاربر جدید اضافه می‌کنیم و تنظیمات چندین مجوز را آزمایش می‌کنیم. سپس مجوز شخصی‌سازی شده‌ی خودمان را طوری ایجاد می‌کنیم که تنها نویسنده آن پست بتواند آنرا بروزرسانی یا حذف کند.

ایجاد کاربر جدید

باید با ایجاد کاربر دوم شروع کنیم. با این کار می‌توانیم بین اکانت‌های این دو کاربر جابجا شویم تا تنظیمات مجوزها را آزمایش کنیم.

به پنل ادمین در <http://127.0.0.1:8000/admin> بروید. سپس روی "Add" کنار Users کلیک کنید. نام کاربری و پسورد را برای کاربر جدید وارد کنید و سپس بر روی دکمه‌ی 'Save' کلیک کنید. من در اینجا نام کاربری را testuser انتخاب کرده‌ام.

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

Save and add another Save and continue editing **SAVE**

تصویر بعدی صفحه‌ی ادمین برای تغییر کاربرها می‌باشد. من کاربر خود را testuser انتخاب کرده‌ام و در اینجا می‌توانم اطلاعات بیشتری را نظیر نام، نام‌خانوادگی، آدرس ایمیل، آدرس و ... به مدل کاربر اضافه کنم. اما هیچکدام از آنها برای مقصود ما مهم نیستند: ما فقط به نام کاربری و پسورد برای آزمایش نیاز داریم.

Change user | Django site admin + Guest

127.0.0.1:8000/admin/auth/user/2/change/

Django administration

WELCOME, **WSV**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Authentication and Authorization > Users > testuser

AUTHENTICATION AND AUTHORIZATION

- Groups [+ Add](#)
- Users [+ Add](#)

POSTS

- Posts [+ Add](#)

Change user

[HISTORY](#)

Username: Required. 150 characters or fewer. Letters, digits and @./-/_. only.

Password:
algorithm: pbkdf2_sha256 iterations: 216000 salt: VTuWii***** hash: sZW1Ey*****
Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:

Last name:

Email address:

Permissions

Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

Staff status
Designates whether the user can log into this admin site.

Superuser status
Designates that this user has all permissions without explicitly assigning them.

Groups:

+ Available groups [?](#)

The groups this user belongs to. A user will get all permissions granted to each of their groups. Hold down "Control", or "Command" on a Mac, to select more than one.

Chosen groups [?](#)

User permissions:

Available user permissions [?](#)

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
admin | log entry | Can view log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | group | Can view group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | permission | Can view permission
auth | user | Can add user

Chosen user permissions [?](#)

همانطور که می‌بینید دو کاربر ما حضور دارند.

اضافه کردن قابلیت ورود به API قابل مرور

هر بار که بخواهیم بین دو کاربر جابجا شویم باید به پنل ادمین جنگو رفته و از آن حساب خارج شده و به دیگری لاگین کنیم. سپس به API موردنظر برویم.

این اتفاق معمولی هست که Django REST Framework تنظیمی یک خطی برای اضافه کردن قابلیت وارد و خارج شدن کاربر به صورت مستقیم به API قابل جستجو دارد. که ما آن را پیاده‌سازی می‌کنیم.

در مسیر پروژه داخل فایل `rest_framework.urls.py`، یک مسیر که شامل `rest_framework.urls` می‌باشد را به URL‌ها اضافه کنید. تا حدودی گیج کننده است، مسیر واقعی مشخص شده می‌تواند هر چیزی که ما می‌خواهیم باشد. چیزی که مهم است یک جایی قرار دارد. ما از مسیر `api-auth` بخارط این که با مستندات مطابقت دارد استفاده می‌کنیم، اما ما هر چیزی را می‌توانستیم استفاده کنیم و همه آنها نیز شبیه به هم عمل می‌کردند.

کد

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')),
]
```

```
path('api-auth/', include('rest_framework.urls')), # new  
]
```

حال به مسیر API قابل مرور در <http://127.0.0.1:8000/api/v1>، بروید. یک تغییر کوچک بوجود آمده است: که یک فلش رو به پایین در کنار نام کاربری در گوشه بالا سمت راست ظاهر شده است. روی آن کلیک کنید.

The screenshot shows the Django REST framework's "Post List" interface at <http://127.0.0.1:8000/api/v1/>. At the top right, there is a "Guest" button and a dropdown menu set to "WSV". Below the header, the page title is "Post List". On the right, there are "OPTIONS" and "GET" buttons. A "Raw data" tab is selected, showing the response body:

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "author": 1,
        "title": "Hello world! (edited)",
        "body": "This is my first blog post.",
        "created_at": "2020-07-29T17:37:03.350702Z"
    }
]
```

Below the response, there is a form with three fields: "Author" (set to "WSV"), "Title" (empty), and "Body" (empty). A "POST" button is located at the bottom right of the form area.

از آنجایی که ما به عنوان کاربر اصلی وارد شده ایم در اینجا برای من WSV نمایان شده است. بر روی لینک کلیک کنید و یک منو کشویی برای خارج شدن (Logout) ظاهر می شود. روی آن کلیک کنید.

لینک بالا در سمت راست حالا به لاگین (login) تغییر پیدا کرده است. روی آن کلیک کنید. به صفحه لاگین Django هدایت می شویم. در اینجا از اکانت آزمایشی استفاده می کنیم. در نهایت به صفحه اصلی API جایی که کاربر آزمایشی در سمت راست و بالا نمایان می باشد، هدایت می شویم.

Post List – Django REST framework X +

← → C ⓘ 127.0.0.1:8000/api/v1/ Guest :

Django REST framework testuser ▾

Post List

Post List OPTIONS GET ▾

GET /api/v1/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
    {  
        "id": 1,  
        "author": 1,  
        "title": "Hello world! (edited)",  
        "body": "This is my first blog post.",  
        "created_at": "2020-07-29T17:37:03.350702Z"  
    }  
]
```

Raw data HTML form

Author	wsv
Title	
Body	

POST

به عنوان قدم آخر از حساب آزمایشی خود خارج شوید.

The screenshot shows the Django REST framework's built-in API browser at <http://127.0.0.1:8000/api/v1/>. The title bar says "Post List - Django REST framework". The main area displays the "Post List" endpoint. At the top right are "OPTIONS" and "GET" buttons. Below them is a "Raw data" tab and an "HTML form" tab. The "Raw data" tab shows the response to a GET request, which includes HTTP headers (Allow: GET, POST, HEAD, OPTIONS; Content-Type: application/json; Vary: Accept) and a JSON array of one post object:

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[{"id": 1, "author": 1, "title": "Hello world! (edited)", "body": "This is my first blog post.", "created_at": "2020-07-29T17:37:03.350702Z"}]
```

The "HTML form" tab contains fields for "Author" (set to "WSV"), "Title" (empty), and "Body" (empty). A "POST" button is at the bottom right.

باید لینک لاگین را دوباره در سمت راست بالا ببینید.

مجوز برای همه

در حال حاضر، هر کاربر ناشناس احراز هویت نشده می‌تواند به لیست پست‌ها دسترسی داشته باشد. ما این را می‌دانیم زیرا اگرچه لاگین نکرده‌ایم، اما می‌توانیم تنها پستمان را مشاهده کنیم. حتی بدتر، هر کسی می‌تواند دسترسی کامل برای ایجاد پست، بروزرسانی و یا حذف آنها داشته باشد.

صفحه جزئیات در آدرس <http://127.0.0.1:8000/api/v1/1> اطلاعات قابل مشاهده می‌باشد و هر کاربر تصادفی می‌تواند پستی را در صورت وجود بروزرسانی یا حذف کند. که خوب نیست.

Post Detail – Django REST fram +

127.0.0.1:8000/api/v1/1/ Guest Log in

Django REST framework Post List / Post Detail

Post Detail

DELETE **OPTIONS** **GET**

GET /api/v1/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
    "id": 1,
    "author": 1,
    "title": "Hello world! (edited)",
    "body": "This is my first blog post.",
    "created_at": "2020-07-29T17:37:03.350702Z"
}
```

Raw data **HTML form**

Author	wsv
Title	Hello world! (edited)
Body	This is my first blog post.

PUT

دلیلی که ما هنوز می‌توانیم پست‌ها و جزئیات آنها را مشاهده کنیم این است که ما تنظیمات مجوزها را در فایل config/settings.py روی AllowAny قرار داده‌ایم. به عنوان یک یادآوری کوچک، به شکل زیر می‌باشد:

کد

```
# config/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

مجوز در سطح نما

آنچه اکنون می‌خواهیم این است که دسترسی کاربران را به API محدود کنیم. چندین روش برای این منظور وجود دارد: در سطح پروژه، در سطح نما(view) یا در سطح شیء اما از آنجایی که ما فقط دو نما داریم پس بباید از آنجا شروع کنیم و برای هر کدام مجوز تعیین کنیم.

در فایل posts/views.py، مازول permissions را وارد کنید و سپس به هر فیلد permission_classes را اضافه کنید.

کد

```

# posts/views.py
from rest_framework import generics, permissions # new
from .models import Post
from .serializers import PostSerializer


class PostList(generics.ListCreateAPIView):
    permission_classes = (permissions.IsAuthenticated,) # new
    queryset = Post.objects.all()
    serializer_class = PostSerializer


class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = (permissions.IsAuthenticated,) # new
    queryset = Post.objects.all()
    serializer_class = PostSerializer

```

این تمام چیزی بود که نیاز داشتیم. صفحه را در <http://127.0.0.1:8000/api/v1> رفرش کنید. ببینید چه اتفاقی افتاد!

ما دیگر نمی‌توانیم لیست پست‌ها را ببینیم. در عوض با پیام HTTP 403 که کد وضعیت ممنوع می‌باشد زیرا ما لاگین نشده‌ایم. و از آنجایی که مجوز نداریم هیچ فرمی در API برای ویرایش داده‌ها وجود ندارد.

اگر به مسیر جزئیات پست در http://127.0.0.1:8000/api/v1 بروید پیامی مشابه را خواهید دید و همچنین فرمی برای ویرایش وجود ندارد.

The screenshot shows a browser window with the title "Post Detail - Django REST framework". The URL is "127.0.0.1:8000/api/v1/1/". The page displays a "Post Detail" section with a "GET" button. Below it, a code block shows the response to a GET request: "HTTP 403 Forbidden" with headers "Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The JSON response is {"detail": "Authentication credentials were not provided."}.

بنابراین از این لحظه تنها کاربران لاگین شده می‌توانند صفحه API را ببینند. اگر با حساب testuser یا superuser خود وارد شوید این صفحات در دسترس هستند.

اما به این فکر کنید که اگر API از نظر پیچیدگی گستردگتر شود. در واقع ما نماها و صفحات بیشتری را در آینده خواهیم داشت. که اضافه کردن `permission_classes` اختصاصی برای هر نما اگر بخواهیم از تنظیمات مجوز مشابه در تمام API استفاده کنیم کاری تکراری به نظر می‌آید.

آیا بهتر نیست که مجوزها را یکبار برای همیشه، برای سطح پروژه انجام دهیم، تا اینکه به ازای هر نما اینکار را انجام دهیم؟

مجوز در سطح پروژه

در این مرحله باید سر خود را به نشانه موافقت تکان دهیم. این یک روش ساده‌تر و امن‌تر برای تنظیم سیاست محدودیت‌های دسترسی در سطح پروژه و در حد نیاز اعمال آن در سطح نما می‌باشد. این کاری است که انجام می‌دهیم.

خوبشخانه Django REST Framework با تعدادی از تنظیمات مجوزها در سطح پروژه همراه می‌باشد که می‌توانیم استفاده کنیم، شامل:

- `AllowAny`، احرار هویت شده باشد یا نه، اجازه دسترسی کامل را دارد
- `IsAuthenticated` کاربران ثبت‌نام شده و احرار هویت شده فقط اجازه دسترسی دارند
- `IsAdminUser` تنها ادمین یا کاربران سوپر اجازه دسترسی خواهند داشت
- `IsAuthenticatedOrReadOnly` تمام کاربران می‌توانند هر صفحه‌ای را ببینند، اما تنها کاربران احرار هویت شده اجازه نوشتن، ویرایش، یا حذف را خواهند داشت

پیاده‌سازی هر کدام از این چهار مورد مستلزم به بروزرسانی `DEFAULT_PERMISSION_CLASSES` و رفرش صفحه‌ی مرورگر می‌باشد. همین!

باید تنظیمات را روی `IsAuthenticated` قرار دهیم تا تنها کاربران احرار هویت و لاگین شده بتوانند صفحه API را ببینند. فایل config/settings.py را به صورت زیر آپدیت کنید.

```
# config/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.isAuthenticated', # new
    ]
}
```

حال به فایل posts/views.py بروید و تغییرات مجوزهایی که همین الان انجام دادیم را حذف کنید.

Code

```
# posts/views.py
from rest_framework import generics
from .models import Post
from .serializers import PostSerializer

class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

اگر صفحات لیست پست‌ها و جزئیات API را رفرش کنید، هنوز کد وضعیت ۴۰۳ را خواهید دید. حال همه‌ی کاربران برای دسترسی به API نیاز به احراز هویت دارند، اما همیشه می‌توانیم تغییراتی اضافی در سطح نما در صورت نیاز ایجاد کنیم.

مجوزهای سفارشی

نوبت به اولین مجوز سفارشی ما می‌باشد. به عنوان خلاصه‌ای از جایی که الان هستیم: دو کاربر داریم، testuser و superuser. یک پست و بلاگ در پایگاهداده ما وجود دارد، که توسط کاربر superuser ساخته شده است.

می‌خواهیم تنها نویسنده آن پست مشخص اجازه دسترسی به ویرایش و یا حذف آن را داشته باشد، در غیر اینصورت پست و بلاگ باید از نوع فقط خواندنی باشد. بنابراین حساب superuser باید دسترسی کامل به عملیات CRUD برای هر پست داشته باشد، اما کاربر معمولی یا همان testuser نباید این مجوزها را داشته باشد.

سرور محلی را با دکمه‌های ترکیبی Control+c متوقف کرده و یک فایل جدید در مسیر اپلیکیشن posts به نام permissions.py ایجاد کنید.

خط فرمان

```
(blogapi) $ touch posts/permissions.py
```

به صورت پیشفرض Django REST Framework متنکی بر کلاس BasePermission میباشد که تمام دیگر کلاس‌های مربوط به مجوزها از آن ارثبری میکنند. این به این معنی است که تنظیمات مجوزهای از پیش ساخته شده‌ای مثل و سایر تنظیمات آنرا گسترش می‌دهند. سورس کد اصلی آن در گیت‌هاب در [دسترس است](#).

کد

```
class BasePermission(object):
    """
    A base class from which all permission classes should inherit.
    """

    def has_permission(self, request, view):
        """
        Return `True` if permission is granted, `False` otherwise.
        """
        return True

    def has_object_permission(self, request, view, obj):
        """
        Return `True` if permission is granted, `False` otherwise.
        """
        return True
```

برای ساخت مجوز سفارشی شده خود، ما تابع `has_object_permission` را اورهاید(یا باطل) میکنیم به طور مشخص میخواهیم مجوز فقط خواندنی را به همه درخواست‌ها بدهیم اما برای هر درخواست نوشتن، مانند ویرایش یا حذف، نویسنده باید همانی باشد که به سایت وارد شده و لاگین کرده است. در اینجا فایل `posts/permissions.py` ما به صورت زیر میباشد.

کد

```
# posts/permissions.py
from rest_framework import permissions

class IsAuthorOrReadOnly(permissions.BasePermission):

    def has_object_permission(self, request, view, obj):
        # Read-only permissions are allowed for any request
        if request.method in permissions.SAFE_METHODS:
            return True

        # Write permissions are only allowed to the author of a post
        return obj.author == request.user
```

ابتدا کلاس `BasePermissions` را وارد کرده‌ایم و کلاس خود را `IsAuthorOrReadOnly` گسترش می‌دهد و از آن ارث‌بری می‌کند را ساخته‌ایم. سپس تابع `has_object_permission` را اوراید کرده‌ایم. اگر در خواست شامل افعال HTTP شامل متدهای امن که یک تاپل شامل GET و OPTIONS و HEAD، باشد پس از نوع درخواست فقط خواندنی است و مجوز مربوط داده می‌شود.

در غیراینصورت درخواست برای نوشتن از هر نوعی می‌باشد، که به معنی بروزرسانی منابع API بنابراین ایجاد، حذف و یا ویرایش می‌باشد. در این مورد، بررسی می‌کنیم که نویسنده‌ای که در شیء(آجکت) درخواست وجود دارد، که همان `obj.author` پست و بلاگ می‌باشد، با همان کاربری که درخواست را ارسال کرده مطابقت دارد.

در فایل `views.py` باید `IsAuthorOrReadOnly` را وارد کنیم و سپس می‌توانیم برای نمای جزئیات پست(`PostDetail`) در `permission_classes` اضافه کنیم.

کد

```
# posts/views.py
from rest_framework import generics
from .models import Post
from .permissions import IsAuthorOrReadOnly # new
from .serializers import PostSerializer

class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = (IsAuthorOrReadOnly,) # new
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

و کار ما به پایان رسید، حالا باید مجوزها را آزمایش کنیم. به صفحه جزئیات پست که در آدرس <http://127.0.0.1:8000/api/v1/1> می‌باشد، بروید. مطمئن شوید که به عنوان کاربر superuser که نویسنده پست می‌باشد، وارد شده‌باشید. نام کاربری باید در قسمت سمت راست بالای صفحه مشخص باشد

Post Detail – Django REST fram +

127.0.0.1:8000/api/v1/1/ Guest :

Django REST framework wsv ▾

Post List / Post Detail

Post Detail

DELETE OPTIONS GET ▾

GET /api/v1/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
    "id": 1,
    "author": 1,
    "title": "Hello world! (edited)",
    "body": "This is my first blog post.",
    "created_at": "2020-07-29T17:37:03.350702Z"
}
```

Raw data HTML form

Author	wsv
Title	Hello world! (edited)
Body	This is my first blog post.

PUT

اگرچه، اگر خارج شوید و با کاربر testuser وارد شوید، صفحه تغییر می‌کند.

Post Detail – Django REST fram +

127.0.0.1:8000/api/v1/1/ Guest :

Django REST framework testuser ▾

Post List / Post Detail

Post Detail

OPTIONS GET ▾

GET /api/v1/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
    "id": 1,
    "author": 1,
    "title": "Hello world! (edited)",
    "body": "This is my first blog post.",
    "created_at": "2020-07-29T17:37:03.350702Z"
}
```

ما می‌توانیم این صفحه را ببینیم زیرا دسترسی‌های فقط خواندنی مجاز هستند. اگرچه به دلیل کلاس مجوز که ساختیم نمی‌توانیم درخواست‌هایی نظیر PUT, DELETE بفرستیم. `IsAuthorOrReadOnly`

دقت کنید که نماهای عمومی تنها مجوزهای در سطح آبجکت را برای نماهایی که تنها یک مدل نمونه را بازمی‌گردانند، بررسی می‌کنند. اگر به فیلتر سطح آبجکت برای نماهایی که لیستی از نمونه‌ها را بازمی‌گردانند نیاز دارید، نیاز به فیلتر با [اووراید کردن کوئری‌ست اولیه](#) دارید.

نتیجه‌گیری

اعمال تنظیمات مناسب بخش مهمی از هر API می‌باشد. به عنوان یک استراتژی عمومی، ایده‌ی خوبی است که از مجوزهای سختگیرانه سطح پروژه استفاده کنید تا فقط کاربران احراز هویت شده دسترسی به API داشته باشند. سپس در صورت نیاز برای نماهای مختلف API مجوزهای در سطح نما یا مجوزهای سفارشی شده خود را ایجاد کنید.

فصل هفتم: احراز هویت کاربر

در فصل قبلی ما دسترسی‌های API‌هایمان را بروز رسانی کردیم، که به آن **مجوز**(authorization) می‌گویند. در این فصل، ما احراز هویت(authentication) را پیاده سازی می‌کنیم که فرآیندی است که، کاربر با آن می‌تواند برای حساب کاربری جدید ثبت نام کرده، به آن وارد یا خارج شود.

به طور سنتی، احراز هویت در وب سایت یکپارچه جنگویی، ساده است و متأثر از یک الگوی کوکی مبتنی بر جلسه(session based cookie pattern) است، که پایین تر آن را مورد بررسی قرار خواهیم داد. اما با وجود یک API، کارها کمی گول زننده‌تر می‌شود. به یاد داشته باشید که HTTP یک پروتکل بدون حفظ حالت(stateless protocol) است پس هیچ راه پیش ساخته‌ای برای به خاطر سپردن اینکه یک کاربر از یک درخواست به درخواست بعدی احراز هویت شده است یا خیر، وجود ندارد. هر بار که یک کاربر درخواست یک منبع محدود شده را می‌کند، باید تایید کند که خودش است.

راه حل آن، ارسال یک نشان یکتا به همراه هر درخواست می‌باشد. به صورت گیج کننده‌ای، یک دیدگاه توافق شده جهانی برای این نشان یکتا تعریف نشده و می‌تواند چندین فرم داشته باشد. فریمورک رست جنگو با [چهار نوع آپشن مختلف احراز هویت پیش ساخته](#) عرضه می‌شود: پایه(basic)، جلسه(session)، توکن(token) و پیش فرض(default). پکیج‌های واسط زیادی وجود دارند که ویژگی‌های بیشتری مانند جیسون وب توکن‌ها را(Json Web Token) یا به اختصار JWT) ارائه می‌دهند.

در این فصل به طور کامل بررسی می‌کنیم که احراز هویت API چگونه کار می‌کند، همچنین مزایا و معایب هر رویکرد را نیز مرور می‌کنیم و سپس یک انتخاب آگاهانه برای API و بلاگ خود انجام می‌دهیم. و در نهایت نقاط نهایی(API) برای ثبت نام، ورود به حساب کاربری و خروج از آن پیاده سازی می‌کنیم.

احراز هویت پایه

raig ترین فرم احراز هویت HTTP به عنوان **احراز هویت «پایه»** شناخته می‌شود. زمانی که کلاینت درخواست HTTP ارسال می‌کند، مجبور به ارسال یک اعتبارنامه(credential) احراز هویت تایید شده قبل از اعطای دسترسی است.

پروسه کامل درخواست/پاسخ به صورت زیر است :

1. کاربر یک درخواست http ارسال می‌کند.
2. سرور یک پاسخ که حاوی کد وضعیت 401 غیرمجاز(unauthorized) و یک هدر WWW-Authenticate با جزئیات چگونگی دسترسی است را برمی‌گرداند.
3. کاربر اعتبارنامه خود را از طریق هدر **Authorization** (مجوز) HTTP ارسال می‌کند.
4. سرور اعتبارنامه را چک کرده و پاسخ را به همراه یکی از کد وضعیت‌های 200 درست(OK) یا 403 منوع(forbidden) را به سمت کاربر ارسال می‌کند.

یک بار که کاربر تایید شد، می‌تواند تمام درخواست‌های بعدی خود را با اعتبارنامه هدر **Authorization** HTTP ارسال کند. ما می‌توانیم این فرایند را به صورت زیر نمایش دهیم:

نمودار

----->
GET / HTTP/1.1

<---

HTTP/1.1 401 Unauthorized

WWW-Authenticate: Basic

----->
GET / HTTP/1.1
Authorization: Basic d3N20nBhc3N3b3JkMTIz

<--

HTTP/1.1 200 OK

توجه داشته باشید که مجوزهای اعتبارنامه ارسال شده بر اساس [base64 encode](#) رمزگذاری نشده، نسخه‌ای از هستند. به عنوان مثال `wsv:password123 <username>:<password>` با `d3N20nBhc3N3b3JkMTIz` است.

مزیت اصلی این روش سادگی آن است اما چندین نقطه ضعف عمدۀ نیز دارد. مورد اول، برای هر درخواست، سرور باید نام کاربری و رمز عبور را جستجو کرده و تایید کند که این عمل ناکارآمد است. اما روش بهتر این است که احراز هویت یکبار صورت گیرد و برای درخواست‌های بعدی یک توکن ارسال شود که بگوید این کاربر تایید شده است. مورد دوم، ارسال اعتبارنامه‌های کاربر به صورت رمزگذاری نشده در سراسر اینترنت، فوق العاده نامن است. هر ترافیک شبکه‌ای که رمزگذاری نشده باشد خیلی راحت می‌تواند دریافت شده و مجدد استفاده شود. بدین ترتیب احراز هویت پایه فقط باید به وسیله پروتکل [HTTPS](#) مورد استفاده قرار بگیرد که نسخه امن شده [HTTP](#) است.

احراز هویت مبتنی بر جلسه

وبسایت‌های یکپارچه مانند وبسایت‌های سنتی جنگو، مدت‌های است که از احراز هویت مبتنی بر جلسه و کوکی به صورت ترکیبی استفاده می‌کنند. در سطح‌های بالاتر، کلاینت با اعتبارنامه خود(نام کاربری و پسورد) احراز هویت می‌کند و بعد از طرف سرور یک شناسه جلسه(*session ID*) دریافت می‌کند که به عنوان یک کوکی ذخیره می‌گردد. سپس این شناسه جلسه در هر یک از هدرهای درخواست‌های [HTTP](#) آینده ارسال می‌شود.

زمانی که شناسه جلسه فرستاده شد، سرور از این شئ جلسه برای جستجوی تمام اطلاعات در دسترس کاربر داده شده از جمله اعتبارنامه استفاده می‌کند.

این، یک روش با حفظ حالت(*stateful*) می‌باشد، به این دلیل که این سابقه باید در هر دو طرف یعنی شئ جلسه در سمت سرور و شناسه جلسه در سمت کلاینت حفظ و نگهداری گردد.

1. کاربر با اعتبارنامه خود(نام کاربری و رمز عبور) وارد می‌شود
2. سرور اعتبارنامه را بررسی می‌کند که درست باشد و در این صورت یک شئ جلسه ایجاد کرده و آن را در دیتابیس ذخیره می‌کند
3. سرور یک شناسه جلسه به سمت کلاینت ارسال می‌کند(خود شئ جلسه ارسال نمی‌شود بلکه ID آن ارسال می‌شود) که به عنوان کوکی در مرورگر ذخیره می‌شود
4. در تمام درخواست‌های بعدی شناسه جلسه به عنوان هدر HTTP گنجانده شده و در صورتی که این شناسه توسط دیتابیس تایید شود، درخواست پردازش می‌شود
5. یک بار که کاربر از حساب کاربری خود خارج شود آنگاه شناسه جلسه هم از سمت کلاینت و هم از سمت سرور حذف می‌شود
6. اگر کاربر مجدداً در حساب کاربری خود وارد شود آنگاه شناسه جلسه جدید توسط سرور ایجاد شده و به عنوان کوکی در سمت کلاینت ذخیره می‌شود

تنظیمات پیش فرض فریمورک رست جنگو در واقع ترکیبی از احراز هویت پایه و مبتنی بر جلسه است. سیستم احراز هویت سنتی مبتنی بر جلسه جنگو استفاده می‌شود و شناسه جلسه در هدر HTTP هر درخواست از طریق احراز هویت پایه ارسال می‌گردد.

مزیت این روش این است که ایمن‌تر است، زیرا اعتبارنامه کاربر فقط یک بار ارسال می‌شود، نه مانند احراز هویت پایه در هر چرخه درخواست/پاسخ، اعتبارنامه کاربر ارسال می‌شود. از طرفی این روش کارآمدتر است زیرا سرور مجبور نیست هر بار اعتبارنامه کاربر را تأیید کند، فقط شناسه جلسه را با شئ جلسه مطابقت می‌دهد که یک جستجوی سریع است.

با این وجود چند جنبه منفی نیز وجود دارد. اولاً آیدی جلسه فقط در مرورگری که کاربر در آن لاتین شده است معتبر است و در چندین دامنه کار نخواهد کرد. اما این یک مشکل واضح است زمانی که یک API نیاز به ساپورت چندین فرانت اند مانند یک وب سایت و یک اپلیکیشن موبایل دارد. دوماً، شئ جلسه باید به روز نگه داشته شود که می‌تواند چالشی در سایتهاي بزرگ که چندین سرور دارند، باشد. چگونه درستی یک شئ جلسه را در هر سرور حفظ می‌کنید؟ و سوماً ارسال کوکی در هر درخواست، حتی درخواست‌هایی که نیاز به احراز هویت ندارند، ناکارآمد است.

در نتیجه، عموماً استفاده از احراز هویت مبتنی بر جلسه برای API‌هایی که چندین فرانت اند دارند توصیه نمی‌شود.

احراز هویت مبتنی بر توکن

سومین دیدگاه عمده و روشی که احراز هویتی که ما برای API و بلاگمان پیاده‌سازی خواهیم کرد، استفاده از احراز هویت مبتنی بر توکن است. این روش یکی از محبوب‌ترین دیدگاه‌ها در سال‌های اخیر است که ناشی از رشد اپلیکیشن‌های تک صفحه‌ای است.

احراز هویت مبتنی بر توکن **قاد حفظ حالت(stateless)** هستند. یک بار که اعتبارنامه اولیه کاربر را به سرور ارسال می‌کند، یک توکن یکتا تولید شده و سپس در سمت کلاینت به عنوان کوکی یا در **حافظه محلی** ذخیره می‌شود. این توکن در هدر هر درخواست HTTP ارسال شده و سرور با آن احراز هویت کاربر را تایید می‌کند. خود سرور چه توکن معتبر باشد چه نباشد، سابقه‌ای از کاربر نگهداری نمی‌کند.

کوکی‌ها برای خواندن اطلاعات از سمت سرور استفاده می‌شوند و از نظر اندازه کوچک هستند (4KB) و به صورت خودکار به همراه هر درخواست HTTP ارسال می‌شوند. حافظه محلی برای اطلاعات سمت کلاینت طراحی شده‌اند و بسیار بزرگتر هستند (5120KB) و محتوای آن‌ها به صورت پیش فرض در هر درخواست HTTP ارسال نمی‌شود. توکن‌های ذخیره شده در کوکی‌ها و یا حافظه‌های محلی در برابر حمله‌های XSS آسیب پذیر هستند. بهترین روش فعلی، ذخیره آن‌ها در یک کوکی به همراه فلگ‌های httpOnly و Secure می‌باشد.

بیایید به یک نسخه ساده از پیام‌های HTTP واقعی در این جریان چالش/پاسخ نگاه کنیم. توجه داشته باشید که هدر Authorization استفاده از Token را مشخص کرده است، که این توکن نیز در هدر WWW-Authenticate درخواست پاسخ در نظر گرفته شده است.

نمودار



در این دیدگاه چندین مزیت وجود دارد از آنجایی که توکن‌ها در کلاینت ذخیره می‌شوند، دیگر مقیاس سرورها به منظور به روز نگه داشتن شئ جلسه، یک مشکل نخواهد بود. و توکن‌ها می‌توانند بین چندین فرانت اند به اشتراک گذاشته شوند. همان توکن می‌تواند نمایانگر یک کاربر روی وب سایت و همان کاربر روی اپلیکیشن موبایل باشد. اما همان شناسه جلسه نمی‌تواند بین فرانت اند‌های مختلف به اشتراک گذاشته شود که این محدودیت عمدۀ این روش است.

یک نقطه ضعف بالقوه در این روش این است که یک توکن ممکن است خیلی بزرگ شود. یک توکن شامل تمام اطلاعات یک کاربر است نه فقط شناسه به عنوان شناسه جلسه یا شی جلسه تنظیم شده. از آنجایی که توکن‌ها در هر درخواست ارسال می‌شوند مدیریت اندازه آنها می‌تواند به یک مسئله کارایی تبدیل شود.

نحوه پیاده سازی دقیق توکن به طور قابل توجهی می تواند متفاوت باشد. احراز هویت مبتنی بر توکن پیش ساخته فریمورک رست جنگو عمده اساسی طراحی شده است. به این معنا که تنظیماتی برای منقضی کردن توکن پشتیبانی نمی شود که یک بهبود امنیتی است، که می تواند اضافه شود. همچنین فقط یک توکن برای هر کاربر تولید می کند بنابراین یک کاربر در وبسایت و در اپلیکیشن موبایل از همان یک توکن استفاده می کند. از آن جایی که اطلاعات مربوط به کاربر به صورت محلی ذخیره می شود، این خود می تواند سبب مشکلی برای نگهداری و به روز رسانی دو مجموعه از اطلاعات کلاینت شود.

جیسون وب توکنها (JWT) جدید هستند در واقع نسخه بهبود یافته توکنها هستند که می توانند به فریمورک رست جنگو از طریق پکیج های واسط اضافه شوند. JWT ها چندین مزیت دارد از جمله اینکه می توانند توکن های یکتا برای کلاینت تولید کنند که دارای تایم انقضا باشد. آنها همچنین می توانند روی سرور و یا با سرویس های واسط مانند Auth0 تولید شوند و JWT ها می توانند رمز گذاری شوند تا به صورت ایمن بر بستر ارتباطات نامن HTTP ارسال شوند.

در نهایت مطمئن ترین شرط برای اکثر API های وب استفاده از طرح احراز هویت مبتنی بر توکن است. JWTها یک قابلیت اضافه هستند اگرچه نیاز به پیکربندی اضافه تری دارند. در نتیجه، در این کتاب از TokenAuthentication پیش ساخته استفاده خواهیم کرد.

احراز هویت پیش فرض

اولین قدم این است که تنظیمات احراز هویت جدیدمان را پیکربندی کنیم. فریمورک رست جنگو با تعدادی تنظیمات ارائه می شود که به طور ضمنی تنظیم شده اند. برای مثال قبل از تغییر DEFAULT_PERMISSION_CLASSES به که به صورت پیش فرض بر روی AllowAny تنظیم شده است.

به طور پیش فرض تنظیم شده است، بنابراین باید صراحتاً DEFAULT_AUTHENTICATION_CLASSES SessionAuthentication و BasicAuthentication را به فایل config/settings.py و خود اضافه کنیم.

کد

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
    'DEFAULT_AUTHENTICATION_CLASSES': [ # new
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication'
    ],
}
```

چرا از هر دو روش استفاده کنیم؟ پاسخ این است که آنها اهداف مختلفی را دنبال می کنند. Sessions برای تقویت API قابل مرور استفاده می شوند و توانایی وارد شدن و خارج شدن از آن. BasicAuthentication برای ارسال شناسه جلسه در هدرهای HTTP برای خود API استفاده می شود.

اگر دوباره به API قابل مرور در <http://127.0.0.1:8000/api/v1> مراجعه کنید، مانند قبل کار خواهد کرد. از نظر فنی، هیچ چیز تغییر نکرده است، ما فقط تنظیمات پیش فرض را به صراحت بیان کرده ایم.

پیاده سازی احراز هویت مبتنی بر توکن

اکنون نیاز داریم تا سیستم احراز هویت خود را بروز رسانی کنیم تا بتوانیم از توکن‌ها استفاده کنیم. قدم اول به روز رسانی TokenAuthentication به مقدار DEFAULT_AUTHENTICATION_CLASS مانند زیر می‌باشد:

کد

```
# config/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.TokenAuthentication', # new
    ],
}
```

ما SessionAuthentication را حذف نکرده زیرا برای API‌های قابل مرور به آن‌ها نیاز داریم، اما اکنون از توکن‌ها برای ارسال و دریافت اعتبارنامه احراز هویت در هدرهای HTTPمان استفاده می‌کنیم. همچنان ما نیاز داریم که اپ را برای تولید توکن‌ها روی سرور اضافه کنیم. آن اپ به همراه فریمورک رست جنگو است اما باید در authtoken بخش INSTALLED_APPS افزوده شود:

کد

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken', # new

    # Local
    'posts',
]
```

از آنجایی که در INSTALLED_APPS تغییراتی ایجاد کرده‌ایم نیاز داریم پایگاه داده خود را همگام سازی کنیم. سرور را با کلید ترکیبی Control + c متوقف کرده و سپس کامند زیر را اجرا کنید.

کامند لاین

```
(blogapi) $ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, authtoken, contenttypes, posts, sessions
```

Running migrations:

Applying authtoken.0001_initial... OK

Applying authtoken.0002_auto_20160226_1747... OK

سپس سرور را مجدداً راه اندازی کنید.

کامند لاین

```
(blogapi) $ python manage.py runserver
```

اگر به آدرس پنل مدیریت جنگو در <http://127.0.0.1:8000/admin> رجوع کنید شما بخش Tokens را در بالا مشاهده خواهید کرد. پیش از آن اطمینان حاصل کنید که با کاربر superuser وارد شده‌اید که دسترسی داشته باشید.

The screenshot shows the Django administration interface at <http://127.0.0.1:8000/admin/>. The top navigation bar includes links for Site administration | Django site, +, Guest, and LOG OUT. The main content area is titled "Django administration" and "Site administration". It features three main sections: "AUTH TOKEN" (with a "Tokens" list, "+ Add", and "Change" buttons), "AUTHENTICATION AND AUTHORIZATION" (with "Groups" and "Users" lists, "+ Add" and "Change" buttons), and "POSTS" (with a "Posts" list, "+ Add", and "Change" buttons). To the right, there is a sidebar titled "Recent actions" listing recent user actions: "testuser User", "Hello world! Post".

صفحه خانه پنل مدیریت جنگو به همراه توکن‌ها

بر روی لینک Tokens کلیک کنید. در حال حاضر هیچ توکنی وجود ندارد که ممکن است برای شما سوپرایز کننده باشد.

The screenshot shows the "Select Token to change" page at <http://127.0.0.1:8000/admin/authtoken/token/>. The top navigation bar includes links for Select Token to change | Django site, +, Guest, and LOG OUT. The main content area is titled "Django administration" and "Home > Auth Token > Tokens". It displays a message "Select Token to change" and "0 Tokens". A "ADD TOKEN +" button is located in the top right corner.

صفحه مدیریت توکن‌ها

در صورتی که ما همه کاربران موجود را داریم. توکن‌ها بعد از اینکه کاربری برای وارد شدن به حسابش، API را صدا بزند ساخته می‌شوند، که این بخش را ما اکنون انجام ندادیم در نتیجه توکنی برای مشاهده وجود ندارد. اما به زودی این کار را انجام خواهیم داد.

ما همچنین نیاز داریم نقاط پایانی‌ای ایجاد کنیم که کاربران بتوانند از طریق آن وارد یا خارج شوند. ما می‌توانیم یک اپ اختصاصی `users` برای این منظور ایجاد کنیم و سپس url‌ها، ویوها و سریالایزرهاخود را به آن اضافه کنیم. با این وجود احراز هویت کاربر بخشی است که واقعاً نمی‌خواهیم اشتباہی در آن رخ دهد. و از آن جایی که همه API‌ها به این قابلیت نیاز دارند، منطقی است که چندین پکیج واسط عالی و تست شده وجود دارند که می‌توانیم از آنها استفاده کنیم.

به ویژه ما از `dj-rest-auth` در ترکیب با `django-allauth` برای ساده‌تر شدن کارها استفاده خواهیم کرد. هیچ گونه احساس بدی در مورد استفاده از پکیج‌های واسط نداشته باشید. آن‌ها به دلیل وجود دارند و حتی حرفه‌ای‌های جنگو هم همیشه به آن‌ها تکیه می‌کنند. اگر مجبور نباشید، هیچ فایده‌ای برای اختراع مجدد چرخ وجود ندارد.

پکیج dj-rest-auth

اول ما نقاط‌های API‌های وارد شدن، خارج شدن و بازنشانی رمز عبور را اضافه خواهیم کرد. این بلافاصله با پکیج محبوب `dj-rest-auth` می‌آید. سرور را با `Control + c` متوقف کنید و سپس آن را نصب کنید.

کامند لاین

```
(blogapi) $ pipenv install dj-rest-auth==1.1.0
```

اپ جدید را به پیکربندی `config/settings.py` در فایل `INSTALLED_APPS` اضافه کنید.

کد

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken',
    'dj_rest_auth', # new

    # Local
    'posts',
]
```

فایل `config/urls.py` را با پکیج `dj-rest-auth` بروز رسانی کنید. ما url را با `api/v1/dj-rest-auth` مسیر دهی می‌کنیم. اطمینان حاصل کنید که URL با خط تیره - از هم جدا شوند و نه خط فاصله _ . این یک خطای آسان است.

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')),
    path('api-auth/', include('rest_framework.urls')),
    path('api/v1/dj-rest-auth/', include('dj_rest_auth.urls')), # new
]
]
```

و تمام. اگر تا به حال تلاش کرده باشید که نقطه پایانی احرار هویت کاربر را خودتان پیاده سازی کنید متوجه خواهید شد که `dj-rest-auth` واقعا فوق العاده است که چگونه چقدر از اتصال وقت و سردرد ما کم می‌کند. اکنون می‌توانیم سرور را مجدداً راه اندازی کنیم تا ببینیم `dj-rest-auth` چه چیزی را برای ما فراهم کرده است.

کامند لاین

```
(blogapi) $ python manage.py runserver
```

ما یک نقطه پایانی برای وارد شدن در <http://127.0.0.1:8000/api/v1/dj-rest-auth/login> داریم

Login – Django REST framework +

127.0.0.1:8000/api/v1/dj-rest-auth/login/ Guest

Django REST framework WSV ▾

Post List / Login

Login

Check the credentials and return the REST Token if the credentials are valid and authenticated.
Calls Django Auth login method to register User ID in Django session framework

Accept the following POST parameters: username, password
Return the REST Framework Token Object's key.

`GET /api/v1/dj-rest-auth/login/`

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{ "detail": "Method \"GET\" not allowed." }
```

Raw data HTML form

Username

Email

Password

POST

ورود کاربر API نقطه پایانی

و نقطه پایانی برای خارج شدن در داریم <http://127.0.0.1:8000/api/v1/dj-rest-auth/logout/>

Logout - Django REST framework +
← → ⌂ ⓘ 127.0.0.1:8000/api/v1/dj-rest-auth/logout/ Guest ⋮

Django REST framework
Post List / Logout
Logout
Calls Django logout method and delete the Token object assigned to the current User object.
Accepts/Returns nothing.
GET /api/v1/dj-rest-auth/logout/
HTTP 405 Method Not Allowed
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{ "detail": "Method \"GET\" not allowed." }
```

Media type: application/json
Content:
POST

ورود کاربر API نقطه پایانی

و همچنین نقاط پایانی بازنشانی رمز عبور نیز به صورت زیر در نظر گرفته شده است:

<http://127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/>

>Password Reset - Django REST x +

← → ⌂ ① 127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/ Guest :

Django REST framework wsv ▾

Post List / Password Reset

Password Reset

OPTIONS

Calls Django Auth PasswordResetForm save method.

Accepts the following POST parameters: email
Returns the success/fail message.

GET /api/v1/dj-rest-auth/password/reset/

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{ "detail": "Method \"GET\" not allowed." }
```

Raw data HTML form

Email POST

بازنشانی رمز عبور API

و برای تایید بازنشانی رمز عبور:

<http://127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/confirm>

Password Reset Confirm – Djar +
 127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/confirm/ Guest :
 Django REST framework wsv ▾

Post List / Password Reset / Password Reset Confirm
 OPTIONS

Password Reset Confirm
 Password reset e-mail link is confirmed, therefore this resets the user's password.

Accepts the following POST parameters: token, uid, new_password1, new_password2
 Returns the success/fail message.

GET /api/v1/dj-rest-auth/password/reset/confirm/

HTTP 405 Method Not Allowed
 Allow: POST, OPTIONS
 Content-Type: application/json
 Vary: Accept

```
{
    "detail": "Method \"GET\" not allowed."
}
```

Raw data HTML form

New password1
 New password2
 Uid
 Token

POST

تایید بازنشانی رمز عبور API

ثبت نام کاربر

مرحله بعدی طراحی و پیاده سازی نقطه پایانی ثبت نام کاربر است. جنگو سنتی یا حتی فریمورک رست جنگو، ویوها یا URL هایی را به صورت پیش ساخته برای ثبت نام کاربر طراحی نکرده است؛ به این معنا که نیاز داریم برای ثبت نام خودمان از صفر کد بنویسیم. این روش قدری با توجه به جدی بودن اشتباه و پیامدهای امنیتی آن قدری خطرناک است.

یک روش محبوب استفاده از پکیج واسط django-allauth است که ویژگی ثبت نام کاربر را به همراه ویژگی های اضافه تری برای سیستم احراز هویت جنگو مانند احراز هویت از طریق فیس بوک، گوگل، توییتر و ... را دارد. اگر ما اضافه کنیم در نتیجه نقاط پایانی ثبت نام کاربر را نیز dj-rest-auth را از پکیج dj-rest-auth.registration خواهیم داشت.

سرور محلی را با Control + c متوقف کرده و پکیج django-allauth را نصب کنید.

کامند لاین

```
(blogapi) $ pipenv install django-allauth~=0.42.0
```

سپس بخش INSTALLED_APPS را بروز رسانی می کنیم. ما باید بعد از آن پیکربندی های جدیدی را اضافه کنیم:

- django.contrib.sites
- allauth
- allauth.account
- allauth.socialaccount
- dj_rest_auth.registration

اطمینان حاصل کنید که SITE_ID و EMAIL_BACKEND را اضافه کرده باشید. از نظر فنی مهم نیست این تنظیمات را کجای فایل config/settings.py قرار داده باشید اما معمولاً پیکربندی های اضافه را مانند قسمت زیر به انتهای فایل اضافه می کنند.

کد

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites', # new

    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken',
    'allauth', # new
    'allauth.account', # new
    'allauth.socialaccount', # new
    'dj_rest_auth',
    'dj_rest_auth.registration', # new

    # Local
    'posts',
]

EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend' # new

SITE_ID = 1 # new
```

پیکربندی email backend مورد نیاز است زیرا به صورت پیش فرض یک ایمیل زمانی که یک کاربر جدید ثبت نام می کند ارسال خواهد شد، و از آنها می خواهد که حساب کاربری خود را تایید کنند. همچنین بجای راه اندازی کردن یک سرور ایمیل، ایمیل ها را با تنظیم کردن console.EmailBackend درون کنسول نمایش خواهیم داد.

پیکربندی SITE_ID یک بخش از فریمورک «sites» جنگو است که یک راهی برای میزبانی چندین وب سایت از یک پروژه جنگو یکسان است. در اینجا ما فقط یک وب سایت داریم که روی آن کار می کنیم اما django-allauth از فریمورک سایت استفاده می کند، پس ما باید تنظیمات پیش فرض را مشخص کنیم.

حال که اپهای جدیدی را افزودیم، لازم است که پایگاه داده را بروزرسانی کنیم.

کامند لاین

```
(blogapi) $ python manage.py migrate
```

سپس مسیر URL جدیدی را برای ثبت نام اضافه می‌کنیم.

کد

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')),
    path('api-auth/', include('rest_framework.urls')),
    path('api/v1/dj-rest-auth/', include('dj_rest_auth.urls')),
    path('api/v1/dj-rest-auth/registration/', # new
         include('dj_rest_auth.registration.urls')),
]
```

و تمام است. می‌توانیم سرور محلی را اجرا کنیم.

کامند لاین

```
(blogapi) $ python manage.py runserver
```

اکنون نقطه پایانی جدیدی برای ثبت نام کاربر در <http://127.0.0.1:8000/api/v1/dj-rest-auth/registration> وجود دارد.

Register – Django REST framework

127.0.0.1:8000/api/v1/dj-rest-auth/registration/ Guest

Django REST framework

Post List / Register

Register

OPTIONS

GET /api/v1/dj-rest-auth/registration/

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
"detail": "Method \"GET\" not allowed."
}

Raw data HTML form

Username

Email

Password1

Password2

POST

ثبت نام کاربر API

The screenshot shows the Django REST framework's 'Register' endpoint. At the top, it says 'Register – Django REST framework' and '127.0.0.1:8000/api/v1/dj-rest-auth/registration/'. There is a 'Guest' button. Below that is a header bar with 'Django REST framework' and 'Post List / Register'. The main content area has a title 'Register' with an 'OPTIONS' button. Underneath is a 'GET /api/v1/dj-rest-auth/registration/' section showing an error response: 'HTTP 405 Method Not Allowed' with headers 'Allow: POST, OPTIONS', 'Content-Type: application/json', and 'Vary: Accept'. The response body is '{ "detail": "Method \"GET\" not allowed." }'. Below this is a form for a 'POST' request with fields 'Username', 'Email', 'Password1', and 'Password2', each with an input field. A 'POST' button is at the bottom right. At the very bottom, it says 'ثبت نام کاربر API'.

توکن‌ها

برای اینکه اطمینان حاصل کنید همه چیز کار میکند، حساب کاربری سومی را از طریق نقطه پایانی API قابل مرور ایجاد کنید. من کاربرم را `testusers2` نامیدم. سپس روی دکمه «POST» کلیک کنید.

Register – Django REST framework +
← → C 127.0.0.1:8000/api/v1/dj-rest-auth/registration/ Guest :

Django REST framework

Post List / Register OPTIONS

GET /api/v1/dj-rest-auth/registration/

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{ "detail": "Method \"GET\" not allowed." }
```

Raw data HTML form

Username	testuser2
Email	testuser2@email.com
Password1	testpass123
Password2	testpass123

POST

ثبت نام کاربر جدید API

تصویر بعدی پاسخ HTTP از طرف سرور را نمایش می‌دهد. درخواست POST ثبت نام کاربر ما موفقیت آمیز بود و از این رو کد وضعیت 201 ایجاد شده(Created at 201) در بالا نمایش داده شده است. و مقدار برگشته key (کلید) همان توکن احراز هویت برای کاربر جدید است.

Register – Django REST framework

127.0.0.1:8000/api/v1/dj-rest-auth/registration/ Guest

Django REST framework WSV ▾

Post List / Register

Register

OPTIONS

POST /api/v1/dj-rest-auth/registration/

HTTP 201 Created
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "key": "b8f2d999ffcb48a3f78e60f408ba08318e036357"  
}
```

Raw data HTML form

Key b8f2d999ffcb48a3f78e60f408ba08318e036357

POST

اگر به کنسول کامند لاین نگاه کنید یک ایمیل که به صورت خودکار توسط django-allauth تولید شده را می‌بینید. متن پیش فرض ایمیل بعداً می‌تواند بروزرسانی شود و یک سرور ایمیل SMTP با پیکربندی اضافه افزوده شود که در کتاب [جنگو برای تازه کارها](#) پوشش داده شده است.

کامند لاین

Hello from example.com!

You're receiving this e-mail because user testuser2 has given yours as an e-mail address \ to connect their account.

To confirm this is correct, go to http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/account-confirm-email/MQ:1k0t5m:6l0l09er1p_cbxgkJWDuSw2j00M/

Thank you from example.com!
example.com

به آدرس پنل مدیریت جنگو در آدرس <http://127.0.0.1:8000/admin> بروید. به حساب سوپریوزر برای اینکار نیاز خواهید داشت. سپس بر روی لینک **Tokens** در بالای صفحه کلیک کنید. شما به صفحه توکن‌ها منتقل خواهید شد.

The screenshot shows the Django administration interface for the 'Tokens' model. The top navigation bar includes links for 'Home', 'Auth Token', and 'Tokens'. On the right, there are links for 'WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT'. Below the navigation, a search bar contains the placeholder 'Select Token to change' and an 'ADD TOKEN +' button. A table lists one token entry:

Action	KEY	USER	CREATED
<input type="checkbox"/>	b8f2d999ffcb48a3f78e60f408ba08318e036357	testuser2	July 29, 2020, 8:54 p.m.

A message at the bottom of the table area says '1 Token'.

توکن‌های مدیر

تنها یک توکن توسط فریمورک رست جنگو برای کاربر **testuser2** ایجاد شده است. هر تعداد کاربر جدید که از طریق API ثبت نام کنند توکن آنها در این بخش نشان داده می‌شود.

یک سوال منطقی در اینجا این است که چرا هیچ توکنی برای حساب سوپریوزر یا **testuser** وجود ندارد؟ پاسخ این است که ما این حساب‌ها را قبل از اینکه احراز هویت توکن را بیافزاییم ساخته‌ایم. اما جای نگرانی نیست، اگر یک بار با این حساب از طریق API وارد شویم، توکن به صورت خودکار اضافه شده و در دسترس خواهد بود.

بگذریم. باید با حساب جدید **testuser2** وارد شویم. در مرورگر و بتان به آدرس <http://127.0.0.1:8000/api/v1/dj-rest-auth/login> بروید. اطلاعات کاربری را برای حساب **testuser2** وارد کنید. بر روی دکمه «POST» کلیک کنید.

Login – Django REST framework

127.0.0.1:8000/api/v1/dj-rest-auth/login/ Guest

Django REST framework wsv ▾

Post List / Login

Login

Check the credentials and return the REST Token if the credentials are valid and authenticated.
Calls Django Auth login method to register User ID in Django session framework

Accept the following POST parameters: username, password
Return the REST Framework Token Object's key.

GET /api/v1/dj-rest-auth/login/

HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{ "detail": "Method \"GET\" not allowed." }
```

Raw data HTML form

Username: testuser2

Email: testuser2@email.com

Password:

POST

وارد شدن testuser2 با API

دو چیز اتفاق افتاده است. در گوشه بالا سمت راست حساب کاربری ما **testuser2** نمایان است که تایید می‌کند ما وارد شده‌ایم. و همچنین سرور نیز پاسخی HTTP به همراه توکن برگردانده است.

Check the credentials and return the REST Token if the credentials are valid and authenticated.
Calls Django Auth login method to register User ID in Django session framework

Accept the following POST parameters: username, password
Return the REST Framework Token Object's key.

POST /api/v1/dj-rest-auth/login/

HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "key": "b8f2d999ffcb48a3f78e60f408ba08318e036357"  
}
```

Raw data HTML form

Key: b8f2d999ffcb48a3f78e60f408ba08318e036357

POST

توکن ورود با API

ما نیاز داریم این توکن را در فریمورک فرانت اندمان دریافت و ذخیره کنیم. به طور سنتی این اتفاق سمت کلاینت، یا در [localStorage](#) یا به عنوان یک کوکی، رخ می‌دهد و سپس همه درخواست‌های شامل این توکن در هدر درخواست به عنوان راهی برای احراز هویت کاربر قرار داده می‌شود. توجه داشته باشید که نگرانی‌های امنیتی بیشتری در این مورد وجود دارد، بنابراین باید مراقب باشید که بهترین روش‌ها را در چارچوب انتخابی فرانت‌اند خود اجرا کنید.

نتیجه گیری

احراز هویت کاربران زمانی که برای اولین بار با API‌های وب کار می‌کنید، یکی از سخت‌ترین بخش‌ها برای درک کردن است. بدون بهره‌مندی از ساختار یکپارچه، ما به عنوان توسعه دهنده باید چرخه درخواست/پاسخ HTTP را عمیقاً درک و پیکربندی کنیم.

فریمورک رست جنگو به همراه پشتیبانی‌های زیادی برای این هدف آمده است که شامل احراز هویت مبتنی بر توکن به صورت پیش ساخته نیز است. هر چند توسعه دهنده‌گان باید بخش‌های اضافی مثل ثبت نام کاربر و ویوها URLs ویوها/urls مختص آن‌ها را پیکربندی کنند. در نتیجه روش محبوب، قدرتمند و امن این است که به پکیج‌های واسطه dj-rest-auth و django-allauth برای کم کردن حجم کدهایی که می‌خواهیم از صفر بنویسیم، تکیه کنیم.

ویوست و روتر ها

ویوست ها (Viewsets) و روتر ها (routers) ابزارهایی در Django REST Framework هستند که می‌توانند سرعت توسعه API را افزایش دهند. آنها یک لایه اضافی از انتزاع در بالای ویوها و URL ها هستند. مزیت اصلی، این است که یک viewset می‌تواند جایگزین چندین view مرتبط شود. و یک router می‌تواند به طور خودکار url برای توسعه دهنده ایجاد کند. در پروژه های بزرگتر با endpoint زیاد، این بدان معناست که یک توسعه دهنده باید کد کمتری بنویسد. همچنین، مسلمًا برای یک توسعه دهنده با تجربه، درک و استدلال در مورد تعداد کمی از viewset و ترکیبات router آسان تر از یک لیست طولانی از view ها و URL ها است.

در این فصل ما دو API endpoint جدید را به پروژه فعلی خود اضافه خواهیم کرد و خواهیم دید که چگونه تغییر از view ها و URL ها می‌تواند به همان عملکرد با کد بسیار کمتر دست یافته.

اندپوینت های کاربر User endpoints

در حال حاضر ما API های زیر را در پروژه خود داریم. همه آنها با پیشوند /api/v1/ هستند که برای اختصار نشان داده نشده است:

Diagram

Endpoint	HTTP Verb
/	GET
/:pk/	GET
/rest-auth/registration	POST
/rest-auth/login	POST
/rest-auth/logout	GET
/rest-auth/password/reset	POST
/rest-auth/password/reset/confirm	POST

دو endpoint توسط ما ایجاد شد در حالی که dj-rest-auth پنج تای دیگر را ارائه می‌دهد. باید اکنون دو دیگر برای فهرست کردن همه و تک کاربران اضافه کنیم. این یک ویژگی مشترک در بسیاری از API ها است که واضح تر می‌کند view ها و URL ها می‌توانند منطقی باشد.

جنگو یک مدل کلاس کاربر داخلی (User) مرسوم دارد که در فصل قبل برای احراز هویت از آن استفاده کرده ایم. بنابراین نیازی به ایجاد مدل جدید در دیتابیس نداریم. در عوض ما فقط endpoint های جدید را سیم کشی کنیم. این فرآیند همیشه شامل سه مرحله زیر است:

- کلاس serializer جدید برای مدل
- view های جدید برای هر endpoint
- مسیرهای URL جدید برای هر endpoint

با serializer شروع کنید. ما نیاز داریم مدل User را import کنیم و یک کلاس UserSerializer ایجاد کنیم که از آن استفاده کند. سپس آن را به فایل posts/serializers.py موجود خود اضافه می‌کنیم.

code

```
# posts/serializers.py
from django.contrib.auth import get_user_model # new
from rest_framework import serializers
from .models import Post

class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = ('id', 'author', 'title', 'body', 'created_at',)

class UserSerializer(serializers.ModelSerializer): # new
    class Meta:
        model = get_user_model()
        fields = ('id', 'username',)
```

شایان ذکر است که در حالی که ما از `get_user_model` برای ارجاع به مدل `User` در اینجا استفاده کرده ایم ، در واقع سه راه مختلف برای ارجاع به مدل `User` در جنگو وجود دارد.

با استفاده از `get_user_model` اطمینان حاصل می کنیم که به مدل `User` صحیح اشاره می کنیم، چه `User` پیش فرض باشد یا یک مدل [User سفارشی](#) همانطور که اغلب در پروژه های جنگو جدید تعریف می شود.

در ادامه باید برای هر endpoint (views) را تعريف کنیم. ابتدا `UserSerializer` را به لیست ها اضافه کنید. سپس هم یک کلاس `UserList` ایجاد کنید که همه کاربران را فهرست می کند و هم یک کلاس `UserDetails` که نمای جزئیات یک کاربر را ارائه می دهد. درست مانند `view` های پست خود، می توانیم از `ListCreateAPIView` و `RetrieveUpdateDestroyAPIView` در اینجا استفاده کنیم. ما همچنین نیاز به ارجاع به مدل کاربران از طریق `get_user_model` داریم بنابراین در خط بالا `import get_user_model`

code

```
# posts/views.py
from django.contrib.auth import get_user_model # new
from rest_framework import generics
from .models import Post
from .permissions import IsAuthorOrReadOnly
from .serializers import PostSerializer, UserSerializer # new

class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer

class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = (IsAuthorOrReadOnly,)
```

```

queryset = Post.objects.all()
serializer_class = PostSerializer

class UserList(generics.ListCreateAPIView): # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer

class UserDetail(generics.RetrieveUpdateDestroyAPIView): # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer

```

اگر متوجه شده باشید، در اینجا کمی تکرار وجود دارد. هم view های پست و هم view های کاربر و queryset یکسانی دارند. شاید بتوان این را به نوعی برای ذخیره کد ترکیب کرد؟

در نهایت ما مسیرهای URL خود را داریم. اطمینان حاصل کنید که view های UserList و UserDetail را import کرده اید. سپس می توانیم از پیشوند users/ برای هر کدام استفاده کنیم.

code

```

# posts/urls.py
from django.urls import path
from .views import UserList, UserDetail, PostList, PostDetail # new

urlpatterns = [
    path('users/', UserList.as_view()), # new
    path('users/<int:pk>/', UserDetail.as_view()), # new
    path('', PostList.as_view()),
    path('<int:pk>/', PostDetail.as_view()),
]

```

و ما تمام کردیم. اطمینان حاصل کنید که سرور محلی همچنان در حال اجرا است و به API قابل مرور (API) بروید تا تأیید کنید همه چیز همانطور که انتظار می رود کار می کند.

لیست endpoint کاربران ما در <http://127.0.0.1:8000/api/v1/users> قرار دارد.

User List – Django REST framework

127.0.0.1:8000/api/v1/users/ Guest testuser2

Django REST framework

Post List / User List

User List

OPTIONS GET

GET /api/v1/users/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
  {  
    "id": 1,  
    "username": "wsv"  
  },  
  {  
    "id": 2,  
    "username": "testuser"  
  },  
  {  
    "id": 3,  
    "username": "testuser2"  
  }  
]
```

Raw data HTML form

Username

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

POST

کد وضعیت (status code) برابر 200 OK است که به این معنی است که همه چیز کار می کند. ما می توانیم سه کاربر موجود خود را ببینیم.

نقشه پایانی (endpoint) جزئیات کاربر در کلید اصلی برای هر کاربر در دسترس است. بنابراین حساب کاربری [./http://127.0.0.1:8000/api/v1/users/1](http://127.0.0.1:8000/api/v1/users/1) ما در آدرس زیر قرار دارد:

The screenshot shows the Django REST framework's User Detail page. At the top, there are navigation links: Post List / User List / User Detail. On the right, there are user authentication options: Guest and testuser2. Below the header, the title is "User Detail". To the right of the title are three buttons: DELETE, OPTIONS, and a dropdown menu containing "GET". A grey box contains the URL "GET /api/v1/users/1/". Below this is a "HTTP 200 OK" response with headers: Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS; Content-Type: application/json; Vary: Accept. The response body is a JSON object: { "id": 1, "username": "wsv" }. At the bottom, there are two tabs: "Raw data" and "HTML form". Under "HTML form", there is a "Username" input field containing "wsv". Below the input field is the placeholder text: "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.". To the right of the input field is a "PUT" button.

ویوست ها

ویوست ها راهی برای ترکیب منطق برای چندین view مرتبط در یک کلاس واحد است. به عبارت دیگر، یک ویوست می‌تواند جایگزین چندین ویو شود. در حال حاضر ما چهار ویو (view) داریم: دو مورد برای پست های وبلاگ و دو مورد برای کاربران. در عوض می‌توانیم عملکرد یکسانی را با دو ویوست تقلید کنیم: یکی برای پست های وبلاگ و دیگری برای کاربران.

معامله این است که کمبود خوانایی برای اشخاص توسعه دهنده که با ویوست ها آشنایی ندارند وجود دارد. بنابراین این یک معامله است.

اینجاست که کد در فایل آپدیت شده‌ی ما `posts/views.py` ظاهر می‌شود وقتی که ویوست هارا با هم عوض می‌کنیم.

code

```
# posts/views.py
from django.contrib.auth import get_user_model
from rest_framework import viewsets # new
from .models import Post
from .permissions import IsAuthorOrReadOnly
from .serializers import PostSerializer, UserSerializer

class PostViewSet(viewsets.ModelViewSet): # new
    permission_classes = (IsAuthorOrReadOnly,)
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

```
class UserViewSet(viewsets.ModelViewSet): # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
```

در بالا به جای import generics از rest_framework، اکنون در حال کردن import viewsets در خط دوم هستیم. سپس ما از ModelViewSet استفاده می کنیم که هم ویو لیست و هم ویو جزئیات را برای ما فراهم می کند. و دیگر مجبور نیستیم همان view را برای هر serializer_class و queryset تکرار کنیم.

در این مرحله، وب سرور محلی متوقف می شود زیرا جنگو از عدم وجود مسیرهای URL مربوطه شکایت می کند. باید آن ها در کار بعدی تنظیم کنیم.

روترها

روترها مستقیما با ویوست ها کار می کنند که به صورت اتوماتیک url ها را برای ما تولید کنند. فایل posts/urls.py حال حاضر ما چهار الگوی url دارد: دو تا از url ها برای بلاگ پست ها و دو تای دیگر به یوزرها اختصاص دارد. در عوض ما می توانیم برای هر viewset یک مسیر(route) داشته باشیم. بنابراین به جای چهار مسیر(route) ما دو مسیر(route) داریم. به نظر بهتر میاد. درسته؟

جنگو رست فریمورک دو روتر دیفالت و پیشفرض دارد: DefaultRouter و SimpleRouter. ما در این کتاب از روتر ساده(SimpleRouter) استفاده خواهیم کرد همچنین امکان ساخت روترهای شخصی سازی شده برای عملکرد پیشرفته تر وجود دارد. چیزی که قراره کد قدیمی آپدیت بشود چیزی شبیه به این هست:

```
# posts/urls.py
from django.urls import path
from rest_framework.routers import SimpleRouter
from .views import UserViewSet, PostViewSet

router = SimpleRouter()
router.register('users', UserViewSet, basename='users')
router.register('', PostViewSet, basename='posts')

urlpatterns = router.urls
```

در خط بالا SimpleRouter همراه با ویوها ایمپورت شده است. روتر بر روی حالت viewset سمت شده است و ما هر کدام از viewset ها برای یوزر و پست ها را ثبت(register) می کنیم. در نهایت ما URL های خود را برای استفاده از روترهای جدید تنظیم می کنیم. پیش میرویم و چهار اندپوینت خود را با دستور python manage.py runserver بررسی میکنیم.

User List – Django REST framework

127.0.0.1:8000/api/v1/users/ Guest testuser2

Django REST framework

Post List / User List

User List

OPTIONS GET

GET /api/v1/users/

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
[{"id": 1, "username": "wsdv"}, {"id": 2, "username": "testuser"}, {"id": 3, "username": "testuser2"}]
```

Raw data HTML form

Username

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

POST

توجه داشته باشید که لیست کاربران یکسان است، اما view جزئیات کمی متفاوت است. این مورد به عنوان "User ModelViewSet" به جای "User Detail" شناخته می شود و یک آپشن اضافه "delete" وجود دارد که در Instance تعییه شده است.

User Instance – Django REST fr +

127.0.0.1:8000/api/v1/users/1/ Guest

Django REST framework testuser2 ▾

Post List / User List / User Instance

User Instance

DELETE OPTIONS GET ▾

GET /api/v1/users/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "id": 1,  
    "username": "wsv"  
}
```

Raw data HTML form

Username wsv
Required. 150 characters or fewer. Letters, digits and @./+/-/_ only.

PUT

This screenshot shows the Django REST framework's browsable API interface for a user instance. At the top, there are navigation links for 'Post List', 'User List', and 'User Instance'. On the right, there are buttons for 'DELETE', 'OPTIONS', and 'GET' (with a dropdown arrow). Below these, a 'Raw data' and 'HTML form' button is shown. The main area contains the response to a 'GET /api/v1/users/1/' request, which includes HTTP headers like 'HTTP 200 OK', 'Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS', 'Content-Type: application/json', and 'Vary: Accept'. The JSON response is: { "id": 1, "username": "wsv" }. Below this, there is a form field labeled 'Username' with the value 'wsv', with a note: 'Required. 150 characters or fewer. Letters, digits and @./+/-/_ only.' A 'PUT' button is located at the bottom right of the form area.

امکان شخص سازی viewset‌ها وجود دارد اما یک بده و بستان (tradeoff) مهم در ازای نوشتن کد کمتر با می باشد که تنظیمات پیش فرض ممکن است نیازمند یک سری پیکربندی اضافی برای رفع آنچیزی که میخواهید باشد.

با رفتن به اندپویت لیست پست ها به آدرس <http://127.0.0.1:8000/api/v1/> می توانیم ببینیم که با حالت قبل (بدون viewset) یکسان است.

Post List – Django REST framework

127.0.0.1:8000/api/v1/ Guest

Django REST framework

testuser2 ▾

Post List

OPTIONS GET

GET /api/v1/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[  
    {  
        "id": 1,  
        "author": 1,  
        "title": "Hello world! (edited)",  
        "body": "This is my first blog post.",  
        "created_at": "2020-07-29T17:37:03.350702Z"  
    }  
]
```

Raw data HTML form

Author: WSV

Title:

Body:

POST

This screenshot shows the 'Post List' endpoint of a Django REST framework application. At the top, there are 'OPTIONS' and 'GET' buttons. Below that is a 'GET /api/v1/' button. The main area displays a single post entry with fields: id (1), author (1), title ('Hello world! (edited)'), body ('This is my first blog post.'), and created_at ('2020-07-29T17:37:03.350702Z'). Below the list is a form with fields for Author (set to 'WSV'), Title, and Body, followed by a 'POST' button.

و مورد مهم این که همچنان دسترسی ها و مجوزها کار میکند. وقتی که به عنوان کاربر2 لایکین کنید آبجکت پست در آدرس [/http://127.0.0.1:8000/api/v1/1](http://127.0.0.1:8000/api/v1/1) می باشد.

Post Instance – Django REST framework

127.0.0.1:8000/api/v1/1/ Guest

Django REST framework

testuser2 ▾

Post Instance

OPTIONS GET

GET /api/v1/1/

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
    "id": 1,  
    "author": 1,  
    "title": "Hello world! (edited)",  
    "body": "This is my first blog post.",  
    "created_at": "2020-07-29T17:37:03.350702Z"  
}
```

This screenshot shows the 'Post Instance' endpoint for the first post. It has 'OPTIONS' and 'GET' buttons at the top, followed by a 'GET /api/v1/1/' button. The main area displays the same post entry as the previous screenshot. Below the list is a form with fields for Author (set to 'WSV'), Title, and Body, followed by a 'POST' button.

به هر حال اگر به عنوان کاربر superuser ما لایکین کنیم که تنها نویسنده بلاگ پست نیز هست ما تمام دسترسی ها از جمله خواندن نوشتن ویرایش کردن و حذف پست را داریم.

Post Instance – Django REST fr +

127.0.0.1:8000/api/v1/1/ Guest

Django REST framework wsv ▾

Post List / Post Instance

Post Instance

DELETE OPTIONS GET ▾

GET /api/v1/1/

HTTP 200 OK

Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{ "id": 1, "author": 1, "title": "Hello world! (edited)", "body": "This is my first blog post.", "created_at": "2020-07-29T17:37:03.350702Z" }
```

Raw data HTML form

Author: wsv

Title: Hello world! (edited)

Body: This is my first blog post.

PUT

The screenshot shows the Django REST framework's browsable API interface. At the top, there's a navigation bar with tabs for 'Post List' and 'Post Instance'. Below that, the title 'Post Instance' is displayed with buttons for 'DELETE', 'OPTIONS', and 'GET'. A dropdown menu next to 'GET' is set to 'wsv'. The main area shows a 'GET /api/v1/1/' button. Underneath it, an 'HTTP 200 OK' response is shown with headers: 'Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS', 'Content-Type: application/json', and 'Vary: Accept'. The response body is a JSON object representing a blog post with fields: id, author, title, body, and created_at. Below this, there's a form for creating a new post. It has three input fields: 'Author' (set to 'wsv'), 'Title' (set to 'Hello world! (edited)'), and 'Body' (set to 'This is my first blog post.'). At the bottom right of the form is a 'PUT' button. There are also 'Raw data' and 'HTML form' tabs above the form.

نتیجه گیری

Viewset ها و روت‌ها یک انتزاع قدرتمند هستند که مقدار کدهایی را که ما به عنوان توسعه دهنده‌گان باید بنویسیم را کاهش می‌دهند به هر حال این مختصر بودن به قیمت یادگیری اولیه یک سری موارد تمام می‌شود اولین باری که از viewset ها و روت‌ها به جای نماها و الگوهای URL استفاده می‌کنید، احساس عجیبی خواهد داشت.

در نهایت تصمیم گیری در مورد اینکه چه زمانی مجموعه ها و روت‌ها را به پروژه خود اضافه کنید کاملاً درونی است. یک قانون خوب این است که با نماها و URL ها شروع کنید. با افزایش پیچیدگی API شما، اگر متوجه شدید که اندپوینت یکسانی را بارها و بارها تکرار می‌کنید، به Viewset ها و روت‌ها نگاه کنید. تا در آن زمان، همه چیز را ساده نگه دارید.

الگو ها (schemas) و مستندات

اکنون که API خود را کامل کرده ایم، به راهی نیاز داریم تا عملکرد آن را به سرعت و با دقت برای دیگران مستند کنیم. به هر حال، در اکثر شرکت ها و تیم ها، توسعه دهنده ای که از API استفاده می کند، همان توسعه دهنده ای نیست که در ابتدا آن را ساخته است. خوبشخтанه، ابزارهای خودکاری برای مدیریت این موضوع برای ما وجود دارد.

URLs و HTTP های موجود API endpoint یک سند قابل خواندن توسط ماشین هست که چارچوب کلی یک schema هایی که پشتیبانی میکنند را مشخص میکند. مستندات چیزی است که methods (GET, POST, PUT, DELETE, etc.) به schema اضافه شده که خواندن و استفاده آن را برای انسان آسان تر میکند. در این فصل ما یک schema به یک پروژه وبلاگ خود اضافه می کنیم و سپس به دو روش مختلف مستند سازی آن را انجام میدهیم. در پایان، ما یک خود پیاده سازی کرده ایم API روش خودکار را برای ثبت هرگونه تغییر (چه در حال چه در آینده) در

به عنوان یادآوری، در اینجا لیست کاملی از API های endpoint فعلی را آمده است :

Diagram

Endpoint	HTTP Verb
/	GET
/:pk/	GET
/users/	GET
/users/:pk/	GET
/rest-auth/registration	POST
/rest-auth/login	POST
/rest-auth/logout	GET
/rest-auth/password/reset	POST
/rest-auth/password/reset/confirm	POST

Schemas

قبل از نسخه 3.9 پایتون Core API schema برای Django REST Framework وابسته بود اما حالا کاملاً به (که قبلاً Swagger نامیده میشد) schema OpenAPI تغییر کرده است. در قدم اول باید PyYAML و uritemplate نصب کنیم.

PyYAML schema تبدیل می کند، در حالی که YAML مبتنی بر OpenAPI های ما را به فرمت uritemplate اضافه می کند URL پارامترهایی را به مسیرهای

Command Line

```
(blogapi) $ pipenv install pyyaml==5.3.1 uritemplate==3.0.1
```

در مرحله بعد، یک انتخاب به ما ارائه می شود: ایجاد یک schema ایستا یا یک API شما اغلب تغییر نمی کند، طرحواره ایستا می تواند به صورت دوره ای تولید شود و از فایل های استاتیک برای عملکرد قوی استفاده شود. با این حال، اگر API شما اغلب تغییر می کند، ممکن است گزینه پویا را در نظر بگیرید. ما هر دو را در اینجا اجرا خواهیم کرد.

در مرحله اول، سراغ schema ایستا میرویم که از دستور مدیریتی generateschema برای تولید schema استفاده میکند. ما می توانیم نتیجه را در فایلی به نام openapi-schema.yml قرار دهیم.

Command Line

```
(blogapi) $ python manage.py generateschema > openapi-schema.yml
```

اگر آن فایل را باز کنید، بسیار طولانی است و خیلی کاربر پسند نیست. اما برای کامپیوتر کاملا فرمت شده است.
برای رویکرد پویا، config/urls.py را با وارد کردن get_schema_view به روزرسانی کنید و سپس یک مسیر اختصاصی در openapi ایجاد کنید. عنوان، توضیحات و نسخه را می توان در صورت نیاز سفارشی کرد.
عنوان، توضیحات و نسخه را می توان در صورت نیاز سفارشی کرد.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path
from rest_framework.schemas import get_schema_view # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')),
    path('api-auth/', include('rest_framework.urls')),
    path('api/v1/dj-rest-auth/', include('dj_rest_auth.urls')),
    path('api/v1/dj-rest-auth/registration/',
         include('dj_rest_auth.registration.urls')),
    path('openapi', get_schema_view( # new
        title="Blog API",
        description="A sample API for learning DRF",
        version="1.0.0"
    ), name='openapi-schema'),
]
```

اگر پروژه شما در بستر سرور محلی قرار دارد با دستور

```
(blogapi) $ python manage.py runserver
```

مجددآن را اجرا کنید، مشاهده میکنید زمانی که به آدرس <http://127.0.0.1:8000/openapi> مراجعه کردید schema به صورت خودکار ساخته شده و درسترس شما قرار میگیرد.

The screenshot shows the Django REST framework's built-in schema browser at 127.0.0.1:8000/openapi. The page title is "Schema". At the top right, there are buttons for "OPTIONS" and "GET". Below the title, it says "HTTP 200 OK" and lists the API's metadata: title ("Blog API"), version ("1.0.0"), and description ("A sample API for learning DRF"). The "paths" section shows the "/api/v1/users/" endpoint with "get" and "post" methods. The "get" method has an "operationId" of "listUsers", a description of "", and a response of type "array" containing objects with properties "id" (integer, readOnly: true), "username" (string, required: true), and "content" (application/json). The "post" method has an "operationId" of "createUser".

من شخصاً رویکرد پویا را در پروژه ها ترجیح می دهم.

مستندات

را به فرمت خواناتر برای schema داخلی است که API همچنین دارای ویژگی [مستندسازی](#) توسعه دهندگان دیگر تبدیل می کند.

در حال حاضر، سه رویکرد محبوب در اینجا وجود دارد: استفاده از [SwaggerUI](#)، [ReDoc](#) و یا بسته شخص ثالث [drf-yasg](#) از آنجایی که drf-yasg بسیار محبوب است و دارای بسیاری از ویژگی های داخلی است، ما در اینجا از آن استفاده خواهیم کرد.

مرحله اول نصب آخرین نسخه drf-yasg است.

```
(blogapi) $ pipenv install drf-yasg==1.17.1
```

مرحله دوم، آن را به پیکربندی config/settings.py اضافه کنید.

```

# config/settings.py
INSTALLED_APPS = [
    ...
    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'rest_auth',
    'rest_auth.registration',
    'drf_yasg', # new

    # Local
    'posts.apps.PostsConfig',
]

```

مرحله سوم، فایل `urls.py` در سطح پروژه را به روزرسانی کنید. در بالای فایل می‌توانیم DRF's `get_schema_view` در `url`s می‌توانیم `get_schema_view` در سطح پروژه را به روزرسانی کنیم. در بالای فایل می‌توانیم `openapi` را با `drf_yasg` جایگزین کنیم و همچنین `openapi` را وارد کنیم. همچنین مجوز DRF را برای گزینه‌های دیگر اضافه می‌کنیم.

متغیر `schema_view` به روزرسانی می‌شود و شامل فیلداتی اضافی مانند `contact`، `terms_of_service` و `license` است. سپس تحت الگوهای `url` خود مسیرهایی را برای `ReDoc` و `Swagger` اضافه می‌کنیم.

Code

```

# config/urls.py
from django.contrib import admin
from django.urls import include, path
from rest_framework import permissions # new
from drf_yasg.views import get_schema_view # new
from drf_yasg import openapi # new

schema_view = get_schema_view( # new
    openapi.Info(
        title="Blog API",
        default_version="v1",
        description="A sample API for learning DRF",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="hello@example.com"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')),
    path('api-auth/', include('rest_framework.urls')),
]

```

```

path('api/v1/dj-rest-auth/', include('dj_rest_auth.urls')),
path('api/v1/dj-rest-auth/registration/',
     include('dj_rest_auth.registration.urls')),
path('swagger/', schema_view.with_ui( # new
    'swagger', cache_timeout=0), name='schema-swagger-ui'),
path('redoc/', schema_view.with_ui( # new
    'redoc', cache_timeout=0), name='schema-redoc'),
]

```

طمئن شوید که سرور شما در حال اجرا است. آدرس Swagger اکنون در آدرس زیر در دسترس است:

<http://127.0.0.1:8000/swagger/>

سپس تأیید کنید که نمای ReDoc نیز در <http://127.0.0.1:8000/redoc> کار می کند.

Blog API (v1)

Download OpenAPI specification: [Download](#)

E-mail: hello@example.com | License: BSD License | Terms of Service

A sample API for learning DRF

Authentication

Basic

Security scheme type:	HTTP
HTTP Authorization Scheme	basic

اسناد drf-yasg کاملاً جامع هستند و بسته به نیاز API های شما سفارشی سازی میشوند.

نتیجه

افزودن مستندات بخش مهمی از هر API است. این معمولاً اولین چیزی است که یک توسعه دهنده همکار به آن نگاه می کند، چه در یک تیم و چه در پروژه های منبع باز. با تشکر از ابزارهای خودکار پوشش داده شده در این فصل، اطمینان از اینکه API شما دارای اسناد دقیق و به روز است، فقط به مقدار کمی پیکربندی نیاز دارد.

نتیجه گیری

در حال حاضر ما در انتهای کتاب قرار داریم اما تنها آغاز چیزی است که می‌توان با "Django REST Framework" انجام داد. در طول سه پروژه مختلف (projects-Library API, Todo API, and Blog API) به تدریج Web API های پیچیده تری از صفر ایجاد کردیم. و تصادفی نیست که در هر مرحله از راه، Django REST Framework ویژگی‌ها و فیچرهای داخلی خود را برای آسان‌تر کردن کار ما فراهم می‌کند.

اگر قبل‌از هرگز API‌ها را با فریمورک دیگری ایجاد نکردید با خبر باشید که بدعادت شده اید و اگر تجربه با فریمورک دیگری برای Web API داشته اید مطمئن باشید که این کتاب فقط به صورت سطحی آنچه را که Django REST Framework می‌تواند انجام دهد را نمایش داده است. [دکیومنت](#) رسمی بهترین منبع برای کاوش بیشتر می‌باشد. اکنون شما به اصول اولیه دست یافته اید.

قدم بعدی

واسیع ترین موضوعی که ارزش مطالعه و کاوش بیشتر را دارد موضوع تست کردن است. تست‌های خود جنگو را می‌توان برای هر API اعمال کرد اما همچنین یک [مجموعه](#) ابزار مناسب در Django REST Framework فقط برای تست کردن درخواست‌های API وجود دارد.

قدم خوب بعدی پیاده سازی یک pastebin API است که [آموزش رسمی DRF](#) آن را پوشش داده شده است. من حتی یک [راهنمای جدید برای تازه کارها](#) نوشته ام که به آن دستور گام‌به‌گام را اضافه کرده ام.

بسته‌های ثالث به همان اندازه که برای جنگو لازم بودن برای Django REST Framework هم لازم می‌باشد. یک لیست کاملی از [پکیج‌های جنگو](#) و یک لیست مدیریت شده در ریپازیتوری [گیتهاپ awesome-django](#) می‌توان پیدا کرد.

سپاس گزاری

در حالی که کامپونیتی و جامعه جنگو بسیار بزرگ است و به کار بعضی از افراد حقیقی وابسته است Django REST Framework در مقایسه با آن بسیار جامعه کوچکی دارد. این فریمورک در ابتدا توسط [Tom Christie](#) توسعه داده شد که یک مهندس نرم افزار انگلیسی زبان است که در حال حاضر هم به صورت فول تایم به لطف منابع مالی منبع باز (open source) بر روی فریمورک جنگو رست کار می‌کند. اگر از کار با Django REST Framework لذت می‌برید، لطفاً یک لحظه شخصاً وقت بگذارید [از او در توییتر تشکر کنید](#). و از شما ممنونم که همراهی کردید و از کار من حمایت کردید اگر این کتاب را از در آمازون خریداری کردید لطفاً یک بازخورد صادقانه ثبت کنید این بازخورد تأثیر بسیار زیادی بر فروش کتاب می‌گذارد و به من کمک می‌کند تا به تولید کتاب و محتوای رایگان جنگو ادامه دهم که من عاشق انجام آن هستم.

ممنون از افرادی که در ترجمه این کتاب مشارکت داشتند

