

# LARAVEL 11.

Laravel 11. neslužbena dokumentacija

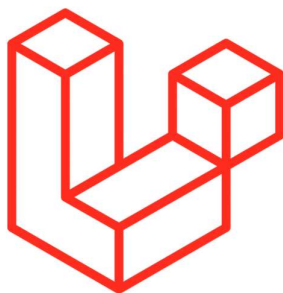
## SADRŽAJ

Laravel je besplatna PHP web radna okolina otvorenog koda, koji je stvorio Taylor Otwell i namijenjen je razvoju web aplikacija i temeljen je na Symfony-ju. Neka od svojstava su modularni sistem pakiranja namjenskim upraviteljem zavisnosti (Composer), različiti načini za pristup relacijskim bazama podataka , uslužni programi koji pomažu u implementaciji i održavanju aplikacije. Originalni i kod Laravela nalazi se na GitHubu i licenciran prema uvjetima MIT licence.



User

[Course title]



# Laravel

Uvod .....	26
Instalacija.....	26
Upoznajte Laravel.....	26
Zašto Laravel?.....	26
Progresivna radna okolina.....	26
Skalabilni radna okolina .....	26
Radna okolina zajednice.....	27
Kreiranje Laravel projekta .....	27
Početna konfiguracija.....	28
Konfiguracija temeljena na okolini.....	28
Baze podataka i migracije.....	28
Konfiguracija direktorija.....	29
Lokalna instalacija pomoću Herd-a .....	29
Herd na macOS-u .....	29
Herd na Windowsima.....	30
Instalacija Dockera pomoću Saila.....	30
Sail na macOS .....	31
Sail na Windows-ima .....	32
Razvoj unutar WSL2 .....	33
Sail na Linuxu.....	33
Odabir vaše Sail usluge.....	34
IDE podrška .....	34
Sljedeći koraci.....	34
Laravel Full Stack radna okolina .....	34
Laravel Pozadina API-ja .....	35
Konfiguracija.....	35
Uvod .....	35
about naredba.....	35
Konfiguracija okoline.....	36
Sigurnost datoteka okoline .....	36
Dodatne datoteke okoline .....	36
Vrste varijabli okoline.....	37
Dohvaćanje konfiguracije okoline .....	37
Određivanje trenutne okoline.....	37
Šifriranje datoteka okoline .....	38
Šifriranje .....	38

Dešifriranje .....	39
Pristup konfiguracijskim vrijednostima .....	39
Konfiguriranje keširanja .....	40
Objavljivanje konfiguracije .....	40
Debug način rada .....	41
Održavanje kao način rada .....	41
Zaobilaženje održavanja kao načina rada .....	42
Način održavanja na više servera .....	42
Prethodan prikaz u održavanju kao načinu rada .....	43
Preusmjeravanje zahtjeva u održavanju kao načinu rada .....	43
Isključivanje održavanja kao načina rada .....	43
Održavanje kao način rada i redovi čekanja .....	43
Alternative održavanju kao načinu rada .....	43
Struktura direktorija .....	44
Uvod .....	44
Root direktorij .....	44
app direktorij .....	44
bootstrap direktorij .....	44
config direktorij .....	44
database direktorij .....	44
public direktorij .....	44
resources direktorij .....	44
routes direktorij .....	44
storage direktorij .....	45
tests direktorij .....	45
vendor direktorij .....	45
app direktorij .....	45
Broadcasting direktorij .....	46
Console direktorij .....	46
Events direktorij .....	46
Exceptions direktorij .....	46
Http direktorij .....	46
Jobs direktorij .....	47
Listeners direktorij .....	47
Mail direktorij .....	47
Models direktorij .....	47

Notifications direktorij .....	47
Policies direktorij.....	47
Providers direktorij.....	48
Rules direktorij .....	48
Frontend.....	48
Uvod .....	48
Korišćenje PHP-a.....	48
PHP i Blade .....	48
Rastuća očekivanja .....	49
Livewire .....	49
Kompleti za početnike.....	50
Korišćenje Vue / React.....	50
Inertia .....	51
Renderiranje na strani servera .....	52
Kompleti za početnike.....	52
Paket materijala (engl. Asset bundling) .....	53
Paketi za početnike .....	53
Uvod .....	53
Laravel Breeze .....	53
Laravel Bootcamp.....	54
Instalacija.....	54
Breeze & Blade .....	55
Breeze & Livewire.....	55
Breeze & React / Vue .....	56
Breeze & Next.js / API .....	56
Next.js referentna implementacija .....	57
Laravel Jetstream .....	57
Implementacija.....	58
Uvod .....	58
Zahtjevi servera .....	58
Konfiguracija servera.....	58
Nginx.....	58
FrankenPHP .....	59
Dopuštenja direktorija .....	59
Optimizacija.....	60
Autoloader konfiguracija.....	60

Keširanje događaja (engl. events) .....	60
Keširanje ruta .....	60
Keširanje pogleda .....	60
Debug mod .....	61
Ruta zdravlja .....	61
Jednostavna implementacija s Forge / Vapor .....	61
Lavarel Forge .....	61
Lavarel Vapor .....	62
Koncepti arhitekture .....	63
Životni ciklus zahtjeva .....	63
Uvod .....	63
Pregled životnog ciklusa .....	63
Prvi koraci .....	63
HTTP / jezgre konzole .....	63
Pružatelji usluga .....	63
Usmjeravanje (engl. routing) .....	64
Dovršavanje .....	64
Usredotočite se na pružatelje usluga .....	65
Kontejner usluga .....	65
Uvod .....	65
Nulta rezolucija konfiguracije .....	66
Kada koristiti kontejner .....	67
Uvezivanje (engl. Binding) .....	67
Osnove uvezivanja .....	67
Jednostavno uvezivanje .....	67
Uvezivanje singleton-a .....	68
Uvezivanje dosega singletons-a .....	69
Uvezivanje instanci .....	69
Povezivanje interfejsa s implementacijama .....	69
Kontekstualno uvezivanje .....	70
Uvezivanje jednostavnih vrijednosti .....	71
Uvezivanje tipom određenih argumenata čiji je broj promjenjiv (engl. Binding Typed Variadics) .....	71
Zavisnosti tagova argumenata čiji je broj promjenjiv (engl. Variadic Tag Dependencies) ....	72
Tagiranje (engl. Tagging) .....	73
Proširivanje uvezivanja .....	73

Rješavanje .....	74
<b>make</b> metoda.....	74
Automatsko injektiranje.....	75
Metode pozivanja i injektiranja.....	76
Kontejnerski događaji.....	77
PSR-11 .....	77
Pružatelji usluga .....	78
Uvod .....	78
Pisanje pružatelja usluga .....	78
Metoda registracije .....	78
<b>bindings</b> i <b>singletons</b> svojstva .....	79
<b>boot</b> metoda.....	80
Injektiranje zavisnosti <b>boot</b> metode.....	81
Registracija pružatelja usluga.....	81
Odgođeni pružatelji.....	81
Fasade .....	83
Uvod .....	83
Pomoćne funkcije.....	83
Kada koristiti fasade .....	84
Fasade nasuprot ubrizgavanja zavisnosti .....	84
Fasade nasuprot Pomoćnih (engl. Helper) funkcija .....	85
Kako funkcioniraju fasade .....	86
Fasade u stvarnom vremenu.....	87
Referenca klase fasade.....	89
Osnove.....	92
Model-View-Controller (MVC) .....	92
Komponente.....	92
Model .....	92
Pogled (engl. View).....	92
Kontroler .....	93
Usluga.....	93
Usmjeravanje (engl. Routing).....	93
Zadane datoteke rute.....	93
API rute.....	94
Dostupne metode rutera .....	94
Injektiranje zavisnosti.....	95

CSRF zaštita .....	95
Redirekcija ruta .....	96
Prikaz ruta .....	96
Izlistajte svoje rute .....	96
Prilagođavanje ruta .....	97
Parametri ruta .....	98
Obavezni parametri.....	98
Parametri i injektiranje zavisnosti .....	99
Opcionalni parametri .....	99
Ograničenja regularnog izraza.....	99
Globalna ograničenja .....	100
Kodiranje kosih crta.....	101
Imenovane rute .....	101
Generiranje URL-ova za imenovane rute .....	102
Pregled trenutne rute.....	103
Grupe ruta .....	103
Middleware .....	103
Kontroleri .....	104
Rutanje poddomena.....	104
Prefiksi ruta .....	104
Prefiksi naziva rute .....	104
Uvezivanje modela rute .....	105
Implicitno uvezivanje .....	105
Meko izbrisani modeli .....	106
Prilagodba ključa .....	106
Prilagođeni ključevi i doseg.....	106
Prilagođavanje ponašanja modela koji nedostaje.....	107
Implicitno uvezivanje enuma .....	108
Eksplicitno uvezivanje .....	108
Prilagođavanje rezolucijske logike .....	109
Zamjenske rute.....	110
Ograničenje brzine .....	111
Definiranje ograničenja brzine .....	111
Ograničenja stope segmentiranja .....	112
Višestruka ograničenja brzine .....	113
Postavljanje ograničenja brzine na rutama.....	113

Reguliranje s Redis-om.....	113
Lažiranje metoda formeSpoofing metode obrasca.....	114
Pristup trenutnoj ruti .....	114
Dijeljenje resursa s različitim izvorima (engl. Cross-Origin Resource Sharing -CORS) .....	115
Keširanje rute .....	115
Middleware .....	116
Uvod .....	116
Definiranje Middleware-a .....	116
Middleware i odgovori .....	117
Registriranje Middleware -a.....	118
Globalni Middleware.....	118
Ručno upravljanje Laravelovim zadanim globalnim middleware-om .....	118
Pridruživanje Middleware rutama .....	119
Isključivanje Middleware.....	119
Middleware grupe.....	120
Laravelove zadane middleware grupe .....	121
Ručno upravljanje Laravelovim zadanim middleware grupama .....	122
Middleware aliasi .....	122
Sortiranje Middleware-a .....	123
Middleware parametri .....	124
Ograničeni Middleware.....	125
CSRF zaštita .....	127
Uvod .....	127
Objašnjenje ranjivosti.....	127
Sprječavanje CSRF zahtjeva .....	127
CSRF tokeni i SPA.....	128
Isključivanje URI-ja iz CSRF zaštite.....	128
X-CSRF-token .....	129
X-XSRF-token .....	129
Kontroleri .....	130
Uvod .....	130
Kontroleri zapisivanja .....	130
Osnovni kontroleri.....	130
Kontroleri s jednostrukim djelovanjem.....	131
Middleware kontroleri .....	132
Kontroleri resursa.....	133



Radnje kojima upravljaju kontroleri resursa .....	133
Prilagođavanje ponašanja modela koji nedostaje.....	134
Meko izbrisani modeli .....	134
Određivanje modela resursa .....	134
Generiranje zahtjeva za formu .....	134
Djelomične rute resursa .....	135
Rute API resursa .....	135
Ugniježđeni resursi.....	135
Određivanje dosega ugniježđenih resursa .....	136
Plitko gniježđenje (engl. shallow nesting).....	136
Imenovanje ruta resursa .....	136
Imenovanje parametara rute resursa .....	137
Određivanje dosega ruta resursa .....	137
Lokaliziranje URI-ja izvora .....	137
Dopunjavanje kontrolera resursa.....	138
Singleton kontroleri resursa .....	138
Singleton resursi koji se mogu kreirati .....	139
API Singleton resursi.....	140
Uvođenje zavisnosti i kontroleri.....	140
Injektiranje konstruktora .....	140
Metoda injektiranja.....	140
HTTP zahtjevi.....	143
Uvod .....	143
Interakcija sa zahtjevom.....	143
Pristupanje zahtjevu.....	143
Injektiranje zavisnosti i parametri rute .....	144
Putanja zahtjeva, host i metoda.....	144
Dohvaćanje putanje zahtjeva .....	144
Provjera putanje/rute zahtjeva .....	145
Dohvaćanje URL-a zahtjeva .....	145
Dohvaćanje host-a zahtjeva .....	145
Dohvaćanje metode zahtjeva.....	146
Zaglavlja zahtjeva .....	146
IP adresa zahtjeva .....	146
Pregovaranje o sadržaju.....	147
PSR-7 zahtjevi .....	147

Unos .....	148
Dohvaćanje unosa .....	148
Dohvaćanje svih ulaznih podataka .....	148
Dohvaćanje ulazne vrijednosti .....	148
Dohvaćanje unosa iz stringa upita .....	149
Dohvaćanje JSON ulaznih vrijednosti .....	149
Dohvaćanje stringable ulaznih vrijednosti .....	149
Dohvaćanje Boolean-ovih ulaznih vrijednosti .....	150
Dohvaćanje datumskih ulaznih vrijednosti .....	150
Dohvaćanje Enum ulaznih vrijednosti .....	150
Dohvaćanje unosa pomoću dinamičkih svojstava .....	151
Dohvaćanje dijela ulaznih podataka .....	151
Prisutnost unosa .....	151
Spajanje dodatnog unosa .....	153
Stari unos .....	153
Blic unos u sesiju .....	154
Blicani unos pa preusmjeravanje .....	154
Dohvaćanje starog unosa .....	154
Kolačići .....	155
Dohvaćanje kolačića iz zahtjeva .....	155
Obrezivanje i normalizacija unosa .....	155
Onemogućivanje normalizacije unosa .....	155
Datoteke .....	156
Dohvaćanje snimljenih datoteka .....	156
Provjera uspješnih uploada .....	156
Putanje i ekstenzije datoteka .....	156
Ostale metode datoteka .....	157
Pohranjivanje datoteka za snimanje .....	157
Konfiguriranje proxy-ja kojim vjerujete .....	157
Povjerenje u sve proxy-je .....	158
Konfiguriranje pouzdanih host-ova .....	158
HTTP odgovori .....	160
Kreiranje odgovora .....	160
Stringovi i matrice .....	160
Objekti odgovora .....	160
Eloquent modeli i kolekcije .....	160

Pridruživanje zaglavlja odgovorima.....	161
Kontrola Middleware keša .....	161
Pridruživanje kolačića uz odgovore.....	162
Generiranje instanci kolačića .....	162
Kolačići koji istječu ranije .....	162
Kolačići i šifriranje.....	163
Preusmjeravanja (redirekcije) .....	163
Preusmjeravanje na imenovane rute .....	164
Popunjavanje parametara pomoću Eloquent modela .....	164
Preusmjeravanje na Radnje kontrolera.....	164
Preusmjeravanje na vanjske domene .....	165
Preusmjeravanje s blicanim podacima o sesiji.....	165
Preusmjeravanje s unosom .....	165
Druge vrste odgovora.....	166
View odgovori .....	166
JSON odgovori .....	166
Preuzimanja datoteka .....	166
Datoteka Response .....	167
Streaming Response.....	167
Stream-ani JSON odgovori .....	168
Makronaredbe odgovora .....	168
Pogledi.....	170
Uvod .....	170
Pisanje pogleda u React / Vue.....	170
Kreiranje i renderiranje pogleda .....	170
Ugniježđeni View direktoriji .....	171
Kreiranje prvog dostupnog pogleda.....	171
Utvrdjivanje postoji li pogled .....	172
Prijenos podataka ka pogledima .....	172
Dijeljenje podataka sa svim pogledima.....	172
Pogled Composer-a .....	173
Pridruživanje konstruktora na više pogleda.....	175
Kreatori pogleda.....	175
Optimiziranje pogleda.....	176
Blade predlošci .....	177
Uvod .....	177

Brži i snažniji Blade s Livewire .....	177
Prikaz podataka .....	177
Kodiranje HTML entiteta .....	178
Prikaz unescaping podataka .....	178
Blade i JavaScript radne okoline .....	179
Izvođenje JSON .....	179
@verbatim direktiva .....	180
Blade direktive .....	180
If izjave .....	180
Direktive za autentifikaciju .....	181
Direktive okoline .....	181
Direktive blokova .....	182
Sesijske direktive .....	182
Switch Izjave .....	182
Petlje .....	183
loop varijabla .....	184
Uslovne klase i stilovi .....	185
Dodatni atributi .....	186
Uključivanje podpregleda .....	187
Renderiranje pogleda za kolekcije .....	188
@once direktiva .....	188
Sirovi PHP .....	189
Komentari .....	189
Komponente .....	189
Ručno registriranje komponenti paketa .....	190
Komponente prikazivanja .....	191
Prosljeđivanje podataka komponentama .....	192
Konvencije imenovanja .....	193
Skraćena sintaksa atributa .....	193
Prikazivanja atributa Escaped sekvenci .....	193
Metode komponenti .....	194
Pristup atributima i slotovima unutar klasa komponenti .....	194
Dodatne zavisnosti .....	195
Skrivanje atributa / metoda .....	196
Atributi komponente .....	196
Defalut / Merged atributi .....	197

Uslovno spajanje klasa .....	197
Spajanje atributa bez klase .....	198
Dohvaćanje i filtriranje atributa .....	198
Rezervirane ključne riječi .....	199
Slotovi.....	200
Doseg slotova .....	200
Opseg slotova .....	201
Atributi slotova.....	202
Umetanje komponenti pogleda .....	202
Generiranje umetnutih komponenti pogleda .....	203
Dinamičke komponente .....	203
Ručno registriranje komponenti .....	203
Automatsko učitavanje komponenti paketa .....	204
Anonimne komponente .....	204
Anonimne komponente indeksa .....	205
Svojstva podataka/atributi .....	205
Pristup roditeljskim (nadređenim) podacima .....	206
Anonimne putanje komponenti .....	207
Izgradnja plana pozicija .....	208
Plan pozicija koji koriste komponente .....	208
Definiranje komponenti plana pozicija .....	208
Primjena plana pozicija komponenti.....	208
Plan pozicija koji koriste nasljeđivanje predložaka .....	209
Definiranje plana pozicija .....	209
Proširivanje plana pozicija.....	210
Forme (Obrasce).....	211
CSRF polje .....	211
method polje .....	211
Greške provjere ispravnosti (validacije) .....	211
Stekovi .....	212
Injektiranje usluge .....	213
Prikazivanje ugrađenih Blade predložaka .....	213
Prikazivanje Blade fragmenata .....	213
Produživanje Blade-a.....	214
Prilagođeni echo rukovatelji.....	215
Prilagođene If izjave .....	216

Paket materijala (Vite) .....	218
Uvod .....	218
Izbor između Vitea i Laravel Mixa .....	218
Migracija nazad na Mix .....	218
Instalacija i podešavanje .....	218
Instaliranje čvora .....	218
Instaliranje Vite i Laravel plugin-a .....	219
Konfiguriranje Vite .....	219
Rad sa sigurnim razvojnim serverom .....	220
Pokretanje razvojnog servera u Sailu na WSL2 .....	221
Učitavanje vaših skripti i stilova .....	221
Umetnuti materijali .....	222
Pokretanje Vite .....	223
Rad s JavaScriptom .....	223
Alias .....	223
Vue .....	224
React .....	225
Inertija .....	225
URL procesiranje .....	226
Rad s tablicama stilova (engl. Stylesheets) .....	227
Rad s Blade-om i rutama .....	227
Obrada statičkih materijala s Vite-om .....	227
Osvježavanje kod snimanja .....	228
Alias .....	229
Ranije dohvaćanje materijala (engl. Asset Prefetching) .....	230
Prilagođeni osnovni URL-ovi .....	231
Varijable okoline .....	232
Onemogućavanje Vite-a u testovima .....	232
Prikazivanje na strani servera (engl. Server-Side Rendering - SSR) .....	233
Skripta i atributi tagova stila .....	234
Politika sigurnosti sadržaja (Content Security Policy - CSP) Nonce .....	234
Integritet podresursa (SRI) .....	235
Proizvoljni atributi .....	236
Napredna prilagodba .....	237
Ispravljanje URL-ova razvojnog servera .....	238
Generiranje URL-ova .....	239

Uvod .....	239
Osnove.....	239
Generiranje URL-ova .....	239
Pristupanje trenutnom URL-u .....	240
URL-ovi za imenovane rute .....	240
Eloquent modeli .....	241
Potpisani URL-ovi .....	241
Potvrđivanje potpisanih zahtjeva za rutu.....	242
Odgovaranje na rute koje su označene kao nevažeće .....	243
URL-ovi za radnje kontrolera.....	243
Zadane vrijednosti.....	243
Zadane postavke URL-a i prioritet middleware-a .....	244
HTTP sesija .....	246
Uvod .....	246
Konfiguracija.....	246
Preduvjeti za driver .....	246
Baza podataka .....	246
Redis .....	247
Interakcija sa sesijom .....	247
Dohvaćanje podataka.....	247
Globalni pomoćnik sesije.....	248
Dohvaćanje svih podataka sesije.....	248
Dohvaćanje dijela podataka sesije .....	248
Određivanje da li stavka postoji u sesiji .....	249
Pohranjivanje podataka .....	249
Guranje vrijednosti u matricu sesije .....	249
Dohvaćanje i brisanje stavke.....	249
Uvećavanje i umanjivanje vrijednosti sesije.....	250
Flash podataka .....	250
Brisanje podataka.....	250
Regeneriranje ID-a sesije.....	251
Blokiranje sesije.....	252
Dodavanje prilagođenih upravljačkih programa sesije .....	252
Implementacija upravljačkog programa .....	252
Registracija driver-a.....	254
Validacija .....	255

Uvod .....	255
Brzi početak provjere ispravnosti.....	255
Definiranje ruta .....	255
Kreiranje kontrolera .....	255
Zapisivanje i provjera ispravnosti logike .....	256
Zaustavljanje nakon prve neuspjele provjere validnosti.....	257
Napomena o ugniježđenim atributima .....	257
Prikaz validacijskih grešaka .....	258
Prilagođavanje poruka o grešci .....	258
XHR zahtjevi i validacija.....	259
@error direktiva .....	259
Ponovno popunjavanje formi.....	260
Napomena o opcionalnim poljima .....	260
Format odgovora validacijske greške.....	260
Validacija zahtjeva forme .....	261
Izrada zahtjeva za formu .....	261
Izvođenje dodatne provjere validnosti .....	262
Zaustavljanje nakon prve neuspjele provjere validnosti.....	264
Prilagođavanje lokacije preusmjeravanja .....	264
Autorizacija zahtjeva formi .....	264
Prilagođavanje poruka o grešci .....	266
Prilagođavanje poruke o grešci .....	266
Priprema unosa za provjeru validnosti.....	266
Ručno kreiranje validatora .....	267
Zaustavljanje nakon prve neuspjele provjere validnosti.....	268
Automatsko preusmjeravanje.....	268
Imenovane torbe s greškama (engl. Named Error Bags) .....	269
Prilagođavanje poruka o grešci .....	269
Određivanje prilagođene poruke za dani atribut.....	269
Određivanje prilagođenih vrijednosti atributa .....	270
Izvođenje dodatne provjere validacije .....	270
Rad s validiranim unosom .....	271
Rad s porukama o greškama .....	272
Dohvaćanje prve poruke o grešci za polje.....	272
Dohvaćanje svih poruka o grešci za polje.....	272
Dohvaćanje svih poruka o grešci za sva polja .....	272



Utvrđivanje postoji li poruka za polje .....	273
Određivanje prilagođenih poruka u jezičnim datotekama .....	273
Prilagođene poruke za određene atribute .....	273
Određivanje atributa u jezičnim datotekama .....	273
Određivanje vrijednosti u jezičnim datotekama .....	274
Dostupna pravila provjere ispravnosti .....	275
accepted .....	275
accepted If: drugo_polje, vrijednost, ... ..	275
active URL .....	276
after:(datum) .....	276
after_or_equal:date .....	276
alpha .....	276
alpha Dash .....	277
alpha_num .....	277
array .....	277
Ascii .....	278
bail .....	278
before (datum) .....	278
before_or_equal (datum) .....	278
between:min, max .....	278
boolean .....	279
confirmed .....	279
current_password .....	279
date .....	279
date_equals:datum .....	279
date_format:format, ... ..	280
decimal:min, max .....	280
declined .....	280
declined_if: drugopolje, vrijednost, ... ..	280
different:polje .....	280
digits .....	280
digits_between:min,max .....	280
dimensions .....	280
distinct .....	281
doesnt_start_with:foo,bar... ..	282
doesnt_end_with:foo,bar .....	282

email.....	282
ends_with:foo,bar .....	283
enum .....	283
exclude .....	283
exclude_if:drugopolje,vrijednost .....	284
exclude_unless:drugopolje, vrijednost .....	284
exclude_with:anotherfield .....	284
exclude_without:anotherfield .....	284
exists:tablica,stupac .....	284
Osnovna upotreba exist pravila .....	284
Određivanje naziva prilagođenog stupca.....	285
extensions:foo,bar,.....	286
file.....	286
filed.....	286
gt:field .....	286
gte:field .....	286
hex_color.....	286
image.....	286
in:foo,bar.....	286
in_array:drugopolje.* .....	287
integer .....	288
ip.....	288
ip4.....	288
ip6.....	288
json .....	288
lt:polje .....	288
lte:polje .....	288
lowercase .....	288
mac_address .....	289
max:vrijednost.....	289
max_digits:vrijednost.....	289
mimetypes:text/plain,.....	289
mimes:foo,bar.....	289
MIME vrste i ekstenzije .....	289
min:vrijednost .....	289
min_digits:vrijednost.....	289

missing.....	290
missing_if:drugopolje,vrijednost,.....	290
missing_unless:drugopolje,vrijednost.....	290
missing_with:foo,bar,.....	290
missing_with_all:foo,bar,.....	290
not_in:foo,bar.....	290
not_regex:uzorak .....	290
nullable .....	291
numeric .....	291
present .....	291
present_if:drugopolje,vrijednost,.....	291
present_unless:drugopolje,vrijednost .....	291
present_with:foo,bar... ..	291
present_with_all:foo,bar.....	291
prohibited.....	291
prohibited_if:drugopolje,vrijednost,.....	291
prohibited_unless:drugopolje,vrijednost,.....	292
prohibits:drugopolje,.....	292
regex:uzorak.....	292
required.....	293
required_if:drugopolje,vrijednost,.....	293
required_if_accepted:drugopolje,.....	294
required_unless:drugopolje,vrijednost,.....	294
required_with:foo,bar,.....	294
required_with_all:foo,bar,.....	294
required_without:foo,bar,.....	294
required_without_all:foo,bar,.....	295
required_array_keys:foo,bar,.....	295
same:polje .....	295
size:vrijednost .....	295
starts_with:foo,bar,.....	295
string.....	295
timezone.....	295
unique:table,column .....	296
Određivanje prilagođene veze s bazom podataka.....	296
Prisiljavanje Unique pravila da ignorira dani ID .....	296

Dodavanje dodatnih Where klauzula .....	297
uppercase .....	297
url .....	297
ulid.....	298
uuid.....	298
Pravila uslovnog dodavanja.....	298
Preskakanje provjere validnosti kada polja imaju određene vrijednosti .....	298
Validacija kada je prisutna.....	298
Složena uslovna validacija .....	299
Složena uslovna validacija matrice.....	300
Validacija matrice .....	300
Validacija unosa ugniježđenog polja .....	301
Pristup podacima ugniježdene matrice.....	301
Indeksi i pozicije poruka o grešci.....	302
Validacija datoteka.....	303
Veličina datoteka.....	304
Tipovi datoteka.....	304
Validacija šifara .....	304
Definiranje podrazumijevanih pravila za šifre.....	305
Prilagođena pravila validnosti .....	306
Korištenje objekata pravila.....	306
Prevođenje poruka validnosti .....	307
Pristup dodatnim podacima .....	308
Korištenje anonimnih funkcija (engl. closures) .....	309
Implicitna pravila .....	310
Upravljački sklop za rukovanje greškama (engl. Error Handling) .....	311
Uvod .....	311
Konfiguracija.....	311
Upravljački sklop za rukovanje iznimkama (engl. Handling Exception) .....	311
Izveštavanje o iznimkama .....	311
Globalni kontekst dnevnika.....	312
Kontekst log izuzetaka.....	312
report pomoćnik.....	313
Uklanjanje dvostruko prijavljenih iznimaka .....	313
Nivoi zapisnika izuzetaka.....	314
Ignoriranje iznimaka prema tipu .....	315

Iznimke kod prikazivanja .....	316
Prikazivanje iznimki kao JSON .....	317
Iznimke koje je moguće prijaviti i prikazati .....	317
Prigušivanje prijavljenih iznimaka .....	319
HTTP iznimke .....	321
Prilagođene stranice s HTTP greškama .....	321
Zamjenske HTTP stranice s greškama .....	322
Evidentiranje .....	323
Uvod .....	323
Konfiguracija .....	323
Dostupni driveri kanala .....	323
Konfiguriranje naziva kanala .....	324
Preduslovi kanala .....	324
Konfiguriranje pojedinačnih i dnevnih kanala .....	324
Konfiguriranje Papertrail kanala .....	325
Konfiguriranje Slack kanala .....	325
Upozorenja o zastarjelosti evidentiranja .....	325
Izgradnja evidencije steka .....	326
Nivoi log-ova .....	326
Pisanje log poruka .....	327
Kontekstualne informacije .....	328
Zapisivanje na određene kanale .....	330
Kanali na zahtjev .....	330
Monolog prilagođavanje kanala .....	330
Prilagođavanje Monologa za kanale .....	330
Kreiranje Monolog Handler kanala .....	332
Monolog formateri .....	332
Monolog procesori .....	333
Kreiranje prilagođenih kanala pomoću Factories .....	333
Praćenje evidentiranih poruka pomoću Pail-a .....	334
Instalacija .....	335
Korištenje .....	335
Filtriranje zapisa .....	335
Dublje kopanje .....	337
Artisan konzola .....	337
Uvod .....	337

Laravel Sail.....	337
Tinker (REPL) .....	337
Instalacija.....	337
Korištenje .....	337
Lista dopuštenih naredbi.....	338
Klase koje ne bi trebale imati aliase .....	338
Pisanje naredbi .....	338
Generiranje naredbi .....	338
Struktura naredbi .....	339
Izlazni kodovi .....	340
Naredbe anonimnih funkcija (engl. closures) .....	340
Zavisnosti nagovještavanjem tipa (engl. type-hint) .....	340
Opisi naredbi anonimnih funkcija (engl. closures) .....	341
Naredbe koje se mogu izolirati .....	341
Zaključavanje ID .....	342
Vrijeme kada istječe zaključavanje.....	342
Definiranje ulaznih očekivanja .....	343
Argumenti.....	343
Opcije .....	343
Opcije s vrijednostima.....	344
Prečice opcija .....	344
Ulazne matrice .....	344
Opcijske matrice.....	345
Opisi unosa .....	345
Upit za nedostajući unos .....	345
Komandni I/O .....	348
Dohvaćanje unosa .....	348
Traženje unosa .....	348
Traženje potvrde .....	349
Automatsko dovršavanje.....	349
Pitanja s višestrukim izborom .....	350
Zapisivanje izlaza .....	350
Tablice .....	351
Trake koje prikazuju napredovanje.....	351
Registriranje naredbi.....	352
Programsko izvršavanje naredbi .....	353

Prosljeđivanje vrijednosti matrice.....	353
Prosljeđivanje Boolean vrijednosti.....	353
Stavljanje u redoslijed (engl. queuing) Artisan naredbi.....	354
Pozivanje naredbi iz drugih naredbi.....	354
Rukovanje signalima.....	355
Stub prilagođavanje .....	355
Događaji .....	355
Emitiranje (engl. Broadcasting) .....	357
Uvod .....	357
Podržani driveri .....	357
Instalacija na strani servera.....	357
Konfiguracija.....	357
Emitovanje davatelja usluga .....	358
Konfiguracija reda čekanja .....	358
Reverb .....	358
Pusher kanali .....	358
Pusher alternative otvorenog koda.....	360
Alby.....	360
Instalacija na strani klijenta.....	361
Reverb .....	361
Pusher kanali .....	362
Korištenje postojećih instanci klijenta .....	363
Alby.....	363
Pregled koncepta.....	364
Korištenje primjera aplikacije.....	365
<b>ShouldBroadcast</b> Interface .....	365
Autoriziranje kanala .....	366
Slušanje prijenosa događaja.....	367
Definiranje događaja emitiranja .....	367
Ime emitiranja .....	369
Emitirani podaci (engl. Broadcast data) .....	369
Red čekanja za emitiranje (engl. Broadcast Queue) .....	370
Uslovi emitiranja .....	371
Transakcije emitiranja i baze podataka.....	371
Autorizacija kanala .....	372
Definiranje autorizacijskih ruta .....	372

Definiranje klasa kanala .....	373
Emitiranje događaja .....	374
Only za druge.....	375
Prilagođavanje konekcije .....	376
Anonimni događaji .....	376
Primanje emitiranja.....	378
Slušanje događaja.....	378
Napuštanje kanala.....	379
Imenski prostori (engl. Namespaces).....	379
Kanali prisutnosti.....	379
Autoriziranje kanala prisutnosti.....	379
Pridruživanje kanalima prisutnosti.....	380
Emitiranje na kanalima prisutnosti .....	380
Model emitiranja (Broadcasting) .....	381
Model konvencija emitiranja .....	383
Slušanje modela emitiranja.....	385
Klijentski događaji .....	386
Obavijesti.....	387
Keš.....	388
Uvod .....	388
Konfiguracija.....	388
Preduvjeti za driver .....	388
Baza podataka .....	388
Memcached.....	388
Redis .....	389
DynamoDB.....	389
Upotreba keša .....	390
Dobivanje keš instance.....	390
Pristup višestrukim keš memorijama .....	391
Dohvaćanje stavki iz keša .....	391
Utvrđivanje postojanja predmeta .....	391
Povećanje/smanjenje vrijednosti.....	391
Dohvaćanje i pohranjivanje.....	392
Dohvati i izbriši .....	392
Pohranjivanje stavki u keš .....	392
Uklanjanje stavki iz keša.....	393



Keš pomoćnik .....	394
Atomska zaključavanja .....	394
Upravljanje zaključavanjima.....	394
Upravljanje zaključavanjem procesa .....	396
Dodavanje prilagođenih driver-a za keš.....	397
Pisanje driver-a.....	397
Registracija driver-a.....	398
Događaji .....	399
Eloquent .....	400
Početak rada.....	400
Uvod .....	400
Laravel Bootcamp.....	400
Generiranje klasa modela .....	400
Provjera modela .....	401
Konvencija Eloquent modela.....	401
Nazivi tablica .....	402
Primarni ključevi.....	402
UUID i ULID ključevi.....	404
Vremenske oznake (engl. Timestamps) .....	405
Veze s bazom podataka.....	406
Zadane vrijednosti atributa .....	407
Konfiguriranje Eloquent Strictness.....	408
Dohvaćanje modela.....	408
Izgradnja upita (engl. Buliding Queries) .....	408
Osvježavanje modela .....	409
Kolekcije .....	409
Ulančavanje rezultata.....	410
Kursori .....	411
Napredni podupiti .....	412
Odabir podupita (engl. Subquery Selects) .....	412
Dohvaćanje pojedinačnih modela/agregata .....	413
Dohvaćanje ili kreiranje modela.....	414
Dohvaćanje agregata.....	415
Umetanje i ažuriranje modela.....	415
Umetci (engl. Inserts) .....	415
Ažuriranja .....	416

Masovna ažuriranja (engl. Mass Updates).....	417
Upsert-i.....	421
Brisanje modela.....	421
Meko brisanje (engl. Soft Deleting).....	423
Upit meko obrisanim modelima.....	424
Modeli rezidbe (engl. Pruning Models).....	425
Repliciranje modela.....	427
Doseg upita.....	428
Globalni dosezi .....	428
Lokalni doseg.....	431
Usporedba modela .....	433
Događaji .....	434
Korišćenje Closures.....	435
Promatrači.....	436
Isključivanje događaja .....	438

## Uvod

### Instalacija

#### Upoznajte Laravel

Laravel je radna okolina web aplikacije s izražajnom, elegantnom sintaksom. Web radna okolina pruža strukturu i početnu točku za stvaranje vaše aplikacije, omogućujući vam da se usredotočite na stvaranje nečeg nevjerojatnog dok se mi znojimo oko detalja.

Laravel nastoji pružiti nevjerojatno razvojno iskustvo uz pružanje snažnih svojstava kao što su temeljito ubrizgavanje zavisnosti (engl. dependency injection), izražajan sloj apstrakcije baze podataka (engl. database abstraction layer), redovi čekanja (engl. queues) i planirani poslovi (engl. scheduled jobs), jedinice (engl. units) i testiranje integracije i drugo.



Bez obzira na to jeste li tek upoznali PHP web radnom okolinom ili imate godine iskustva, Laravel je radna okolina koja može rasti s vama. Pomoći ćemo vam da napravite svoje prve korake kao web programer ili ćemo vas potaknuti dok svoju stručnost podižete na viši nivo. Jedva čekamo vidjeti što ćete napraviti.

#### NAPOMENA:



Novi ste u Laravel-u? Provjerite [Laravel Bootcamp](#) za praktični obilazak radne okoline dok vas vodimo kroz izradu vaše prve Laravel aplikacije.

#### Zašto Laravel?

Kada izrađujete web aplikacije dostupni su vam razni alati i radna okoline. Međutim, vjerujemo da je Laravel najbolji izbor za izradu modernih, full-stack web aplikacija.

#### Progresivna radna okolina

Laravel volimo nazivati “progresivnim” radna okolinom. Pod tim mislimo da Laravel raste s vama. Ako tek radite svoje prve korake u web razvoju, Laravel-ova ogromna biblioteka dokumentacije, vodiča i video uputa pomoći će vam da naučite sve bez preopterećenja.

Ako ste senior programer, Laravel vam daje robusne alate za uvođenje injektiranja zavisnosti (engl. dependency injection), testiranje jedinica (engl. unit testing), redove čekanja (engl. queues), događaje u stvarnom vremenu (engl. real-time events) i više. Laravel je fino podešen za izradu profesionalnih web aplikacija i spreman je nositi se s radnim opterećenjima u firmama.

#### Skalabilni radna okolina

Laravel je nevjerojatno skalabilan. Zahvaljujući prirodi prilagođenoj skaliranju PHP-a i Laravel-ovoj ugrađenoj podršci za brze, distribuirane keš sisteme kao što je Redis, horizontalno skaliranje s Laravel-om je povjetarac. U stvari, Laravel aplikacije su lako skalirane za obradu stotina miliona zahtjeva mjesečno.

Trebate ekstremno skaliranje? Platforme kao što je [Laravel Vapor](#) omogućavaju pokretanje Vaše Laravel aplikacije u gotovo neograničenom opsegu na najnovijoj tehnologiji AWS-a bez servera.



### *Radna okolina zajednice*

Laravel kombinira najbolje pakete u PHP ekosistemu kako bi ponudio najrobusniju dostupnu radnu okolinu prilagođenu razvojnim programerima. Osim toga, tisuće talentiranih programera iz cijelog svijeta pridonijeli su radnoj okolini. Tko zna, možda čak i postanete Laravel suradnik.

### Kreiranje Laravel projekta

Prije stvaranja vašeg prvog Laravel projekta, provjerite ima li vaše lokalni kompjuter instaliran PHP i [Composer](#).

Composer je upravitelj zavisnosti (engl. dependency manager) na nivou aplikacije namijenjen upravo PHP-u, koji pruža standardni format za upravljanje zavisnostima PHP softvera i potrebnih biblioteka. Razvili su ga Nils Adermann i Jordi Boggiano, koji su nastavili upravljati projektom. Composer je snažno inspiriran [Node.js](#)<sup>1</sup>-ovim NPM<sup>2</sup> i Rubyjevim „bundler-om“. Composer se pokreće iz komandne linije i instalira zavisnosti (npr. biblioteke) za aplikaciju. Također omogućava korisnicima da instaliraju PHP aplikacije koje su dostupne na „[Packagist](#)“ koji je njegov glavni repozitorij koji sadrži dostupne pakete. Također pruža mogućnosti automatskog učitavanja za biblioteke koje navode informacije o automatskom učitavanju kako bi se olakšalo korištenje koda treće strane.

Ako razvijate na macOS-u, PHP i Composer mogu se instalirati za nekoliko minuta pomoću Laravel Herd-a. Osim toga, preporučujemo instaliranje Node i NPM.

Nakon što ste instalirali PHP i Composer, možete kreirati novi Laravel projekt pomoću Composer-ove `create-project` naredbe:

```
composer create-project laravel/laravel primjer-app
```

Ili, možete kreirati nove Laravel projekte globalnom instalacijom [Laravel programa za instalaciju](#) pomoću Composer-a:

```
composer global require laravel/installer
```

```
laravel new primjer-app
```

Nakon što je projekt kreiran, pokrenite Laravel-ov lokalni razvojni server pomoću Laravel Artisan `serve` naredbe:

```
cd primjer-app
```

```
php artisan serve
```

---

<sup>1</sup> Node.js je višepatformska okolina za izvršavanje JavaScript-a s otvorenim kodom. Omogućava programerima korištenje JavaScript-a iz komandne linije i skriptiranje na strani servera. Koristi se za generiranje dinamičkog sadržaja web stranice na serveru prije nego što se stranica pošalje browseru korisnika.

<sup>2</sup> NPM ili Node.js packet manager je upravitelj paketa za pakete ili module. NPM se instalira na vaš kompjuter zajedno sa Node.js. Na [www.npmjs.com](http://www.npmjs.com) postoji ogroman broj besplatnih paketa za skinuti

Nakon što pokrenete Artisan<sup>3</sup> razvojni server, vaša će aplikacija biti dostupna u vašem web browseru na adresi <http://localhost:8000>. Sada ste spremni započeti s poduzimanjem sljedećih koraka u Laravel ekosistemu. Naravno, možda ćete također htjeti konfigurirati bazu podataka.

**NAPOMENA:**



Ako želite prednost kod razvoju svoje Laravel aplikacije, razmislite o korištenju jednog od naših paketa za početnike (engl. starter kit). Laravel-ovi kompleti za početnike pružaju backend i frontend autentifikacijsku skelu (engl. scaffolding) za vašu novu Laravel aplikaciju.

### Početna konfiguracija

Sve konfiguracijske datoteke za Laravel radnu okolinu pohranjene su u `config` direktoriju. Svaka opcija je dokumentirana, zato slobodno pregledajte datoteke i upoznajte se s opcijama koje su vam dostupne.

Laravel ne treba gotovo nikakvu dodatnu konfiguraciju izvan okvira. Slobodno započnite s razvojem! Međutim, možda ćete htjeti pregledati `config/app.php` datoteku i njezinu dokumentaciju. Sadrži nekoliko opcija kao što su `timezone` i `locale` koje možda želite promijeniti prema svojoj aplikaciji.

### Konfiguracija temeljena na okolini

Budući da mnoge Laravel-ove vrijednosti konfiguracijskih opcija mogu varirati ovisno o tome izvodi li se vaša aplikacija na vašem lokalnom kompjuteru ili na produkcijskom web serveru, mnoge važne konfiguracijske vrijednosti definirane su pomoću datoteke `.env` koja postoji u početnom direktoriju (engl. root) vaše aplikacije.

Vaša `.env` datoteka ne bi trebala biti dodijeljena kontroli izvora vaše aplikacije, budući da svaki programer/server koji koristi vašu aplikaciju može zahtijevati drugačiju konfiguraciju okoline. Nadalje, to bi bio sigurnosni rizik u slučaju da uljez dobije pristup vašem repozitoriju kontrole izvora, jer bi svi osjetljivi akreditivi (engl. credentials) bili izloženi.

**BILJEŠKA:**



Za više informacija o `.env` datoteci i konfiguraciji temeljenoj na okolini (engl. Environment Based Configuration), pogledajte potpunu konfiguracijsku dokumentaciju.

### Baze podataka i migracije

Sada kada ste kreirali svoju Laravel aplikaciju, vjerojatno želite pohraniti neke podatke u bazu podataka. Prema podrazumijevanim postavkama, vaša aplikacijska `.env` konfiguracijska datoteka specifikira da će Laravel biti u interakciji s MySQL bazom podataka i da će pristupiti bazi podataka na adresi `127.0.0.1`.

**NAPOMENA:**



Ako razvijate na macOS-u i trebate lokalno instalirati MySQL, Postgres ili Redis, razmislite o korištenju DBngin-a.

---

<sup>3</sup> Artisan je Laravel-ov CLI tj. interface komandne linije. Najčešće se koristi za upravljanje migracijama i dodavanje lažnih podataka u bazu podataka za potrebe testiranja (sijanje baze podataka), generiranje koda za nove migracije i kontrolere. Još jedna popularna osobina je mogućnost automatizacije za ponavljajuće zadatke.

Ako ne želite instalirati MySQL ili Postgres na vašem lokalnom kompjuteru, uvijek možete koristiti SQLite bazu podataka. SQLite je mali, brzi, samostalni motor baze podataka. Za početak ažurirajte svoju `.env` konfiguracijsku datoteku da biste koristili Laravel-ov `sqlite` upravljački program baze podataka. Možete ukloniti druge opcije konfiguracije baze podataka:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

Nakon što ste konfigurirali svoju SQLite bazu podataka, možete pokrenuti migracije baze podataka vaše aplikacije, koje će kreirati tablice baze podataka vaše aplikacije:

```
php artisan migrate
```

Ako SQLite baza podataka ne postoji za vašu aplikaciju, Laravel će vas pitati želite li da se baza podataka izradi. Tipično, SQLite datoteka baze podataka bit će kreirana na `database/database.sqlite`.

### *Konfiguracija direktorija*

Laravel bi se uvijek trebao posluživati iz početnog direktorija „web direktorija” konfiguriranog za vaš web server. Ne biste trebali pokušavati posluživati Laravel aplikaciju iz poddirektorija „web imenika”. Pokušaj da to učinite mogao bi otkriti osjetljive datoteke unutar vaše aplikacije.

### *Lokalna instalacija pomoću Herd-a*

Laravel Herd je munjevito brzo, prirodno Laravel i PHP razvojno okruženje za macOS i Windows. Herd uključuje sve što vam je potrebno za početak razvoja Laravela, uključujući PHP i Nginx.

Nakon što instalirate Herd, spremni ste za početak razvoja s Laravelom. Herd uključuje alate naredbenog retka za `php`, `composer`, `laravel`, `expose`, `node`, `npm` i `nvm`.

### **NAPOMENA:**



[Herd Pro](#) proširuje Herd s dodatnim moćnim svojstvima, kao što je mogućnost kreiranja i upravljanja lokalnim bazama podataka MySQL, Postgres i Redis, kao i pregledavanje lokalne pošte i nadzor log-ova.

### *Herd na macOS-u*

Ako razvijate na Macu možete preuzeti program za instalaciju Herd s [web stranice Herd](#). Instalacijski program automatski preuzima najnoviju verziju PHP-a i konfigurira vaš Mac da uvijek pokreće [Nginx](#) u pozadini.

Herd za macOS koristi [dnsmasq](#) za podršku "parkiranih" direktorija. Svaku Laravel aplikaciju u parkiranom imeniku automatski će poslužiti Herd. Prema zadanim postavkama, Herd stvara parkirani direktorij na `~/Herd` i možete pristupiti bilo kojoj Laravel aplikaciji u ovom direktoriju na `.test` domeni koristeći naziv direktorija.

Nakon instaliranja Herda, najbrži način za stvaranje novog Laravel projekta je korištenje Laravel CLI-ja, koji je u paketu s Herdom:

```
cd ~/Herd
laravel new my-app
cd my-app
herd open
```

Naravno, uvijek možete upravljati svojim parkiranim direktorijima i drugim PHP postavkama preko Herdovog korisničkog interface-a, koje se može otvoriti iz izbornika Herd na vašem sistem tray-u.

Više o Herdu možete saznati ako pogledate [dokumentaciju Herda](#).

### *Herd na Windowsima*

Možete preuzeti Windows instalaciju za Herd na [Herd web stranici](#). Nakon što instalacija završi, možete pokrenuti Herd kako biste dovršili proces integracije i prvi put pristupili korisničkom sučelju Herd.

Korisničkom interface-u Herd-a može se pristupiti klikom lijeve tipke miša na ikonu Herdove systemske trake (engl. system tray). Desni klik otvara brzi meni s pristupom svim alatima koji su vam potrebni svakodnevno.

Tokom instalacije, Herd stvara "parkirani" direktorij u vašem matičnom direktoriju na `%USERPROFILE%\Herd`. Svaku Laravel aplikaciju u parkiranom direktoriju automatski će poslužiti Herd, a bilo kojoj Laravel aplikaciji u ovom direktoriju na `.test` domeni možete pristupiti koristeći njezin naziv direktorija.

Nakon instaliranja Herda, najbrži način za stvaranje novog Laravel projekta je korištenje Laravel CLI-ja, koji je u paketu s Herdom. Za početak otvorite Powershell i pokrenite sljedeće naredbe:

```
cd ~\Herd
laravel new my-app
cd my-app
herd open
```

Više o Herdu možete saznati ako pogledate [Herd dokumentaciju za Windows](#).

### *Instalacija Dockera pomoću Saila*

Želimo da početak rada s Laravelom bude što lakši bez obzira na operativni sistem koji preferirate. Dakle, postoji niz opcija za razvoj i pokretanje Laravel projekta na vašem lokalnom kompjuteru. Iako biste možda željeli istražiti ove opcije kasnije, Laravel nudi Sail , ugrađeno rješenje za pokretanje vašeg Laravel projekta pomoću [Dockera](#).

Docker je alat za pokretanje aplikacija i usluga u malim, laganim "kontejnerima" koji ne ometaju instalirani softver ili konfiguraciju vašeg lokalnog kompjutera. To znači da ne morate brinuti o konfiguraciji ili postavljanju kompliciranih razvojnih alata kao što su web serveri i baze podataka na vašem lokalnom računalu. Za početak trebate samo instalirati [Docker Desktop](#).

Laravel Sail je lagani interface komandnog reda za interakciju s Laravelovom zadanom konfiguracijom Dockera. Sail pruža sjajnu početnu točku za izradu Laravel aplikacije pomoću PHP-a, MySQL-a i Redisa bez potrebe za prethodnim iskustvom s Dockerom.

**NAPOMENA:**



Već ste stručnjak za Docker? Ne brinite! Sve o Sailu može se prilagoditi pomoću `docker-compose.yml` datoteke uključene u Laravel.

### *Sail na macOS*

Ako razvijate na Macu i Docker Desktop je već instaliran, možete koristiti jednostavnu naredbu terminala za stvaranje novog Laravel projekta. Na primjer, da biste stvorili novu Laravel aplikaciju u direktoriju pod nazivom "`primjer-app`", možete pokrenuti sljedeću naredbu na svom terminalu:

```
curl -s "https://laravel.build/primjer-app" | bash
```

Naravno, možete promijeniti "`primjer-app`" u ovom URL-u u što god želite - samo pazite da naziv aplikacije sadrži samo alfanumeričke znakove, crtice (engl. dashes) i donje crtice (engl. underscores). Direktorij Laravel aplikacije bit će kreiran unutar direktorija iz kojeg izvršavate naredbu.

Sail instalacija može potrajati nekoliko minuta dok se Sailovi aplikacijski kontejneri izgrade na vašem lokalnom kompjuteru.

Nakon što je projekt izrađen, možete otići do direktorija aplikacije i pokrenuti Laravel Sail. Laravel Sail pruža jednostavan interface komandnog reda za interakciju s Laravelovom zadanom Docker konfiguracijom:

```
cd example-app  
  
./vendor/bin/sail up
```

Nakon što se pokrenu Docker kontejneri aplikacije, trebali biste pokrenuti migracije baze podataka svoje aplikacije :

```
./vendor/bin/sail artisan migrate
```

Konačno, aplikaciji možete pristupiti u svom web browser-u na: `http://localhost`.

**NAPOMENA:**



Kako biste nastavili učiti više o Laravel Sail, pregledajte njegovu kompletnu dokumentaciju .



### *Sail na Windows-ima*

Prije nego što stvorimo novu Laravel aplikaciju na vašem Windows računalu, svakako instalirajte Docker Desktop . Zatim biste trebali osigurati da je Windows podsistem za Linux 2 (WSL2) instaliran i omogućen. WSL vam omogućuje izvorno pokretanje Linux binarnih izvršnih datoteka u sustavu Windows 10. Informacije o tome kako instalirati i omogućiti WSL2 mogu se pronaći unutar Microsoftove [dokumentacije za razvojnu okolinu](#).

#### **NAPOMENA:**



Nakon instaliranja i omogućavanja WSL2, trebali biste provjeriti je li Docker Desktop konfiguriran za korištenje WSL2 pozadine.

Zatim ste spremni za izradu svog prvog Laravel projekta. Pokrenite [Windows terminal](#) i započnite novu sesiju terminala za vaš operativni sistem WSL2 Linux. Zatim možete koristiti jednostavnu naredbu terminala za stvaranje novog Laravel projekta. Na primjer, da biste stvorili novu Laravel aplikaciju u direktoriju pod nazivom "primjer-app", možete pokrenuti sljedeću naredbu na svom terminalu:

```
curl -s https://laravel.build/primjer-app | bash
```

Naravno, možete promijeniti "primjer-app" u ovom URL-u u što god želite - samo pazite da naziv aplikacije sadrži samo alfanumeričke znakove, crtice i crtice za podvlačenje. Direktorij Laravel aplikacije bit će kreiran unutar direktorija iz kojeg izvršavate naredbu.

Sail instalacija može potrajati nekoliko minuta dok se Sailovi spremnici aplikacija izgrade na vašem lokalnom kompjuteru.

Nakon što je projekt izrađen, možete otići do direktorija aplikacije i pokrenuti Laravel Sail. Laravel Sail pruža jednostavan interface komandnog reda za interakciju s Laravelovom zadanom Docker konfiguracijom:

```
cd example-app  
  
./vendor/bin/sail up
```

Nakon što se pokrenu Docker kontejneri aplikacije, trebali biste pokrenuti migracije baze podataka svoje aplikacije:

```
./vendor/bin/sail artisan migrate
```

Konačno, aplikaciji možete pristupiti u svom web pregledniku na: <http://localhost> .

**NAPOMENA:**



Kako biste nastavili učiti više o Laravel Sail, pregledajte njegovu [kompletnu dokumentaciju](#).

*Razvoj unutar WSL2*

Naravno, morat ćete moći modificirati Laravel aplikacijske datoteke koje su stvorene unutar vaše WSL2 instalacije. Da biste to postigli, preporučujemo korištenje editora koda Microsoft [Visual Studio](#) i njihovog dodatka za [daljnji razvoj](#).

Nakon što su ovi alati instalirani, možete otvoriti bilo koji Laravel projekt izvršavanjem naredbe `code` iz root direktorija vaše aplikacije pomoću Windows terminala.

*Sail na Linuxu*

Ako razvijate na Linuxu i [Docker Compose](#) je već instaliran, možete koristiti jednostavnu naredbu terminala za stvaranje novog Laravel projekta.

Prvo, ako koristite Docker Desktop za Linux, trebali biste izvršiti sljedeću naredbu. Ako ne koristite Docker Desktop za Linux, možete preskočiti ovaj korak:

```
docker context use default
```

Zatim, da biste stvorili novu Laravel aplikaciju u direktoriju pod nazivom „primjer-app“, možete pokrenuti sljedeću naredbu na svom terminalu:

```
curl -s https://laravel.build/primjer-app | bash
```

Naravno, možete promijeniti „primjer-app“ u ovom URL-u u što god želite - samo pazite da naziv aplikacije sadrži samo alfanumeričke znakove, crtice i donje crtice. Direktorij Laravel aplikacije bit će kreiran unutar direktorija iz kojeg izvršavate naredbu.

Sail instalacija može potrajati nekoliko minuta dok se Sailovi kontejneri aplikacija izgrade na vašem lokalnom kompjuteru.

Nakon što je projekt izrađen, možete otići do direktorija aplikacije i pokrenuti Laravel Sail. Laravel Sail pruža jednostavan interface iz komandne linije za interakciju s Laravelovom zadanom konfiguracijom Dockera:

```
cd primjer-app  
  
./vendor/bin/sail up
```

Nakon što se pokrenu Docker kontejneri aplikacije, možete pristupiti aplikaciji u svom web browser-u na lokaciji: <http://localhost>.

**NAPOMENA:**



Kako biste nastavili učiti više o Laravel Sail, pregledajte njegovu [kompletnu dokumentaciju](#).

### Odabir vaše Sail usluge

Kada kreirate novu Laravel aplikaciju pomoću Saila, možete koristiti `with` upit s string varijablom da odaberete koje usluge treba konfigurirati u datoteci vaše nove aplikacije `docker-compose.yml`. Dostupne usluge uključuju `mysql`, `pgsql`, `mariadb`, `redis`, `memcached`, `meilisearch`, `minio`, `selenium` i `mailpit`:

```
curl -s "https://laravel.build/primjer-app?with=mysql,redis" | bash
```

Ako ne navedete koje usluge želite konfigurirati, bit će konfiguriran zadana gomila `mysql`, `redis`, `meilisearch`, `mailpit` i `selenium`.

Možete uputiti Sail da instalira zadani [Devcontainer](#) dodavanjem `devcontainer` parametra u URL:

```
curl -s "https://laravel.build/primjer-app?with=mysql,redis&devcontainer" | bash
```

### IDE podrška

Slobodno možete koristiti bilo koji editor koda kada razvijate Laravel aplikacije; međutim, [PhpStorm](#) nudi opsežnu podršku za Laravel i njegov ekosistem, uključujući [Laravel Pint](#).

Dodatno, dodatak [Laravel Idea](#) PhpStorm koji održava zajednica nudi mnoštvo korisnih IDE proširenja, uključujući generiranje koda, Eloquent dovršavanje sintakse, dovršavanje pravila validacije i više.

### Sljedeći koraci

Sada kada ste izradili svoj Laravel projekt, možda se pitate što ćete sljedeće naučiti. Prvo, toplo preporučujemo da se upoznate s načinom na koji Laravel radi čitanjem sljedeće dokumentacije:

- Životni ciklus zahtjeva
- Konfiguracija
- Struktura imenika
- Frontend
- Kontejner usluga
- Fasade

Način na koji želite koristiti Laravel također će diktirati sljedeće korake na vašem putu. Postoje različiti načini korištenja Laravel-a, a mi ćemo istražiti dva primarna slučaja upotrebe radne okoline u nastavku.

#### NAPOMENA:



Novi ste u Laravelu? Provjerite [Laravel Bootcamp](#) za praktični obilazak radne okoline dok vas vodimo kroz izradu vaše prve Laravel aplikacije.

### Laravel Full Stack radna okolina

Laravel može poslužiti kao full stack radna okolina. Pod „full stack” radnom okolinom mislimo da ćete koristiti Laravel za usmjeravanje zahtjeva vašoj aplikaciji i renderiranje vašeg interface-a pomoću Blade template-a ili hibridne tehnologije jednostrane aplikacije kao što je [Inertia](#). Ovo je najčešći način korištenja Laravel radne okoline i, po našem mišljenju, najproduktivniji način korištenja Laravela.

Ako je ovo način na koji planirate koristiti Laravel, možda biste trebali provjeriti našu dokumentaciju o frontend razvoju, usmjeravanju (engl. routing), pogledima (engl. views) ili Eloquent ORM<sup>4</sup>-u. Osim toga, moglo bi vas zanimati učenje o paketima zajednice kao što su [Livewire](#) i [Inertia](#). Ovi paketi vam omogućuju da koristite Laravel kao full-stack radnu okolinu dok uživate u mnogim prednostima korisničkog interface-a koje pružaju JavaScript aplikacije na jednoj stranici.

Ako koristite Laravel kao full stack framework, također vas snažno potičemo da naučite kako kompajlirati CSS i JavaScript vaše aplikacije pomoću Vite-a.

**NAPOMENA:**

Ako želite unaprijed započeti s izradom svoje aplikacije, pogledajte jedan od naših službenih paketa za početak aplikacije.

### *Laravel Pozadina API-ja*

Laravel također može poslužiti kao API backend za JavaScript aplikaciju na jednoj stranici ili mobilnu aplikaciju. Na primjer, možete koristiti Laravel kao API backend za svoju [Next.js](#) aplikaciju. U tom kontekstu, možete koristiti Laravel za provjeru autentifikaciju i pohranjivanje/dohvaćanje podataka za svoju aplikaciju, dok također iskorištavate Laravel-ove moćne usluge kao što su redovi čekanja (engl. queues), e-pošta, notifikacije i više.

Ako je to način na koji planirate koristiti Laravel, možda biste trebali provjeriti našu dokumentaciju o usmjeravanju (engl. routing), Laravel Sanctum i Eloquent ORM.

**BILJEŠKA:**

Trebate prednost u postavljanju skela (engl. scaffolding) za Laravel backend i Next.js frontend? Laravel Breeze nudi API stack kao i frontend implementaciju Next.js tako da možete započeti za nekoliko minuta.

## Konfiguracija

### Uvod

Sve konfiguracijske datoteke za Laravel okvir pohranjene su u [config](#) direktoriju. Svaka opcija je dokumentirana, zato slobodno pregledajte datoteke i upoznajte se s opcijama koje su vam dostupne.

Ove konfiguracijske datoteke vam omogućuju da konfigurirate stvari kao što su informacije o povezivanju s vašom bazom podataka, informacije o vašem serveru e-pošte, kao i razne druge osnovne konfiguracijske vrijednosti kao što su vremenska zona vaše aplikacije i ključ za šifriranje (engl. encryption key).

### *about naredba*

U žurbi? Možete dobiti brzi pregled konfiguracije svoje aplikacije, upravljačkih programa i okoline pomoću [about](#) naredbe Artisan:

---

<sup>4</sup> Objektno-relacijsko mapiranje (engl. Object–relational mapping – ORM) u je tehnika programiranja za pretvaranje podataka između relacijske baze podataka i gomile (engl. heap) objektno orijentiranog programskog jezika. Time se zapravo stvara baza podataka virtualnih objekata koja se može koristiti unutar programskog jezika. Rezultat je virtualna objektna baza podataka koja se može koristiti unutar programskog jezika. Eloquent je ORM uključen u Laravel radnu okolinu.

### php artisan about

Ako vas zanima samo određeni odlomak izlaza pregleda aplikacije, možete filtrirati taj odlomak pomoću `--only` opcije:

### php artisan about --only=environment

Ili, da biste detaljno istražili vrijednosti određene konfiguracijske datoteke, možete upotrijebiti Artisan naredbu `config:show`:

### php artisan config:show database

## Konfiguracija okoline

Često je korisno imati različite konfiguracijske vrijednosti na temelju okoline (engl. environment) u kojem se aplikacija izvodi. Na primjer, možda ćete htjeti lokalno koristiti drugačiji cache driver od onoga koji koristite na svom produkcijskom serveru.

Kako bi ovo bilo jednostavno, Laravel koristi [DotEnv](#) PHP biblioteku. U novoj instalaciji Laravel-a, početni direktorij vaše aplikacije sadržavat će datoteku `.env.example` koja definira mnoge uobičajene varijable okoline. Tokom procesa instalacije Laravel-a, ova će se datoteka automatski kopirati u `.env`.

Laravel-ova zadana `.env` datoteka sadrži neke uobičajene konfiguracijske vrijednosti koje se mogu razlikovati ovisno o tome izvodi li se vaša aplikacija lokalno ili na produkcijskom web serveru. Te se vrijednosti zatim dohvaćaju iz različitih Laravel konfiguracijskih datoteka unutar `config` direktorija pomoću Laravel-ove `env` funkcije.

Ako razvijate s timom, možda ćete htjeti nastaviti uključivati datoteku `.env.example` u svoju aplikaciju. Stavljanjem rezerviranih mjesta u primjer konfiguracijske datoteke, drugi programeri u vašem timu mogu jasno vidjeti koje su varijable okoline potrebne za pokretanje vaše aplikacije.

### NAPOMENA:



Sve varijable u vašoj `.env` datoteci mogu nadjačati (engl. overridden) varijable vanjske okoline kao što su varijable okoline na nivou servera ili sistema.

## Sigurnost datoteka okoline

Vaša `.env` datoteka ne bi trebala biti dodijeljena kontroli izvora vaše aplikacije, budući da svaki programer/server koji koristi vašu aplikaciju može zahtijevati drugačiju konfiguraciju okoline. Nadalje, to bi bio sigurnosni rizik u slučaju da uljez dobije pristup vašem repozitoriju kontrole izvora, jer bi svi osjetljivi akreditivi (engl. credentials) bili izloženi.

Međutim, moguće je šifrirati vašu datoteku okoline pomoću Laravel-ove ugrađene enkripcije okoline. Datoteke šifrirane okoline mogu se sigurno staviti u kontrolu izvora.

## Dodatne datoteke okoline

Prije učitavanja varijabli okoline vaše aplikacije, Laravel utvrđuje je li `APP_ENV` varijabla okoline dostavljena izvana ili je `--env` naveden argument u interface-u komandne linije (engl. command line

interface - CLI)<sup>5</sup>. Ako je tako, Laravel će pokušati učitati `.env.[APP_ENV]` datoteku ako postoji. Ako ne postoji, `.env` učitat će se zadana datoteka.

### Vrste varijabli okoline

Sve varijable u vašim `.env` datotekama obično se analiziraju (engl. parsed) kao stringovi, pa su stvorene neke rezervirane vrijednosti koje vam omogućuju vraćanje šireg raspona tipova iz `env()` funkcije:

<code>.env</code> vrijednost	<code>env()</code> vrijednost
true	(bool) true
(true)	(bool) true
false	(bool) false
(false)	(bool) false
empty	(string) ""
(empty)	(string) ""
null	(null) null
(null)	(null) null

Ako trebate definirati varijablu okoline s vrijednošću koja sadrži razmake, to možete učiniti tako da vrijednost stavite u dvostruke navodnike:

```
APP_NAME="Moja aplikacija "
```

### Dohvaćanje konfiguracije okoline

Sve varijable navedene u `.env` datoteci bit će učitane u `$_ENV` PHP super-global kada vaša aplikacija primi zahtjev. Međutim, možete koristiti ovu `env` funkciju za dohvaćanje vrijednosti iz ovih varijabli u vašim konfiguracijskim datotekama. U stvari, ako pregledate Laravel konfiguracijske datoteke, primijetit ćete da mnoge opcije već koriste ovu funkciju:

```
'debug' => env('APP_DEBUG', false),
```

Druga vrijednost proslijeđena funkciji `env` je „zadana vrijednost“. Ova vrijednost će biti vraćena ako ne postoji varijabla okoline za dani ključ.

### Određivanje trenutne okoline

Trenutačna okolina aplikacije određeno je pomoću `APP_ENV` varijable iz vaše `.env` datoteke. Ovoj vrijednosti možete pristupiti pomoću `environment` metode na `App` facade (fasadi):

```
use Illuminate\Support\Facades\App;

$environment = App::environment();
```

<sup>5</sup> CLI je tekstualni interface u koje možete unijeti naredbe koje su u interakciji s operativnim sistemom kompjutera. CLI radi uz pomoć zadane ljuške, koja se nalazi između operativnog sistema i korisnika. Kod Windowsa je to CMD.

Možete također proslijediti argumente metodi `environment` da odredite odgovara li okolina danoj vrijednosti. Metoda će se vratiti `true` ako okolina odgovara bilo kojoj od zadanih vrijednosti:

```
if (App::environment('local')) {
    // Environment je local
}

if (App::environment(['local', 'staging'])) {
    // Environment (Okolina) je ili local ILI staging (izvođenje)...
}
```

#### NAPOMENA:



Trenutno otkrivanje okoline aplikacije može se nadjačati definiranjem `APP_ENV` varijable okoline na nivou servera.

#### Šifriranje datoteka okoline

Nešifrirane datoteke okoline nikad se ne smiju pohranjivati u kontroli izvora. Međutim, Laravel vam omogućava šifriranje datoteka vaše okoline tako da se mogu sigurno dodati u kontrolu izvora s ostatkom vaše aplikacije.

#### Šifriranje

Za enkripciju (šifriranje) datoteke okoline, možete koristiti `env:encrypt` naredbu:

```
php artisan env:encrypt
```

Pokretanje `env:encrypt` naredbe će šifrirati vašu `.env` datoteku i smjestiti šifrirani sadržaj u `.env.encrypted` datoteku. Ključ za dešifriranje predstavljen je u izlazu naredbe i trebao bi biti pohranjen u sigurnom upravitelju šifri. Ako želite dati svoj vlastiti ključ za šifriranje, možete koristiti opciju `--key` kada pozivate naredbu:

```
php artisan env:encrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

#### NAPOMENA:



Dužina dostavljenog ključa trebala bi odgovarati dužini ključa koju zahtijeva šifra šifriranja koja se koristi. Prema podrazumijevanim postavkama, Laravel će koristiti `AES-256-CBC` šifru koja zahtijeva ključ od 32 znaka. Slobodno koristite bilo koju šifru koju podržava Laravelov kriptor prosljeđivanjem `--cipher` opcije kada pozivate naredbu.

Ako vaša aplikacija ima više datoteka okoline, kao što su `.env` i `.env.staging`, možete navesti datoteku okoline koja bi trebala biti šifrirana davanjem naziva okoline pomoću `--env` opcije:

```
php artisan env:encrypt --env=staging
```

### Dešifriranje

Za dešifriranje datoteke okoline, možete koristiti `env:decrypt` naredbu. Ova naredba zahtijeva ključ za dešifriranje koji će Laravel dohvatiti iz `LARAVEL_ENV_ENCRYPTION_KEY` varijable okoline:

```
php artisan env:decrypt
```

Ili, ključ se može dati direktno naredbi pomoću `--key` opcije:

```
php artisan env:decrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

Kada se naredba `env:decrypt` pozove, Laravel će dekriptirati sadržaj datoteke `.env.encrypted` i smjestiti dešifrirani sadržaj u `.env` datoteku.

Opcija `--cipherse` može dati naredbi `env:decrypt` kako bi se koristila prilagođena šifra za šifriranje:

```
php artisan env:decrypt --key=qUWuNRdfuImXcKxZ --cipher=AES-128-CBC
```

Ako vaša aplikacija ima više datoteka okoline, kao što su `.env` i `.env.staging`, možete navesti datoteku okoline koja bi se trebala dešifrirati davanjem naziva okoline pomoću `--env` opcije:

```
php artisan env:decrypt --env=staging
```

Kako biste prebrisali postojeću datoteku okoline, možete unijeti opciju `--force` za `env:decrypt` naredbu:

```
php artisan env:decrypt --force
```

### Pristup konfiguracijskim vrijednostima

Svojim konfiguracijskim vrijednostima možete jednostavno pristupiti pomoću globalne `config` funkcije s bilo kojeg mjesta u vašoj aplikaciji. Vrijednostima konfiguracije može se pristupiti pomoću sintakse "točka", koja uključuje naziv datoteke i opciju kojoj želite pristupiti. Također se može navesti podrazumijevana vrijednost koja će biti vraćena ako opcija konfiguracije ne postoji:

```
use Illuminate\Support\Facades\Config;

$value = Config::get('app.timezone');

$value = config('app.timezone');

// Dohvati podrazumijevanu vrijednost ako vrijednost konfiguracije ne postoji...

$value = config('app.timezone', 'Asia/Seoul');
```



Za postavljanje konfiguracijskih vrijednosti tokom izvođenja, možete pozvati `Config` fasade `set` metodu ili proslijediti matricu `config` funkciji:

```
Config::set('app.timezone', 'Europe/Zagreb');

config(['app.timezone' => 'Europe/Zagreb']);
```

Kao pomoć kod statičke analize, `Config` fasada također nudi metode dohvaćanja tipizirane konfiguracije. Ako dohvaćena vrijednost konfiguracije ne odgovara očekivanom tipu, izbacit (engl. thrown) će se iznimka (engl. exception):

```
Config::string('config-key');
Config::integer('config-key');
Config::float('config-key');
Config::boolean('config-key');
Config::array('config-key');
```

### Konfiguriranje keširanja

Da biste ubrzali svoju aplikaciju, trebali biste keširati sve konfiguracijske datoteke u jednu datoteku pomoću `config:cache` Artisan naredbe. Ovo će kombinirati sve opcije konfiguracije za vašu aplikaciju u jednu datoteku koju možete brzo učitati u radnu okolinu.

Naredbu biste obično trebali pokrenuti `php artisan config:cache` kao dio produkcijskog razvojnog procesa. Naredba se ne bi trebala pokretati to kom lokalnog razvoja jer će konfiguracijske opcije često trebati mijenjati tokom razvoja vaše aplikacije.

Nakon što se konfiguracija kešira, `.env` datoteka neće se učitati u radnu okolinu tokom zahtjeva ili Artisan naredbi; zato će `env` funkcija vratiti samo vanjske varijable okoline na nivou sistema.

Iz tog razloga, trebali biste osigurati da pozivate `env` funkciju samo iz konfiguracijskih (`config`) datoteka vaše aplikacije. Možete vidjeti mnoge primjere ovoga ispitujući Laravelove zadane konfiguracijske datoteke. Konfiguracijskim vrijednostima može se pristupiti s bilo kojeg mjesta u vašoj aplikaciji koristeći iznad opisanu `config` funkciju.

Naredba `config:clear` se može koristiti za čišćenje keširane konfiguracije:

```
php artisan config:clear
```

### UPOZORENJE:



Ako izvršite `config:cache` naredbu tokom procesa razvoja, trebali biste biti sigurni da pozivate funkciju samo `env` iz konfiguracijskih datoteka. Nakon što se konfiguracija spremi u keš, `.env` datoteka se neće učitati; zato će `env` funkcija vratiti samo vanjske varijable okoline na nivou sistema.

### Objavljivanje konfiguracije

Većina konfiguracijskih datoteka Laravela već je objavljena u `config` direktoriju vaše aplikacije; međutim, određene konfiguracijske datoteke kao što je `cors.php` i `view.php` nisu objavljene prema zadanim postavkama, budući da ih većina aplikacija nikada neće morati mijenjati.

Međutim, možete upotrijebiti `config:publish` naredbu Artisan za objavljivanje konfiguracijskih datoteka koje nisu objavljene prema zadanim postavkama:

```
php artisan config:publish

php artisan config:publish --all
```

### Debug način rada

`debug` opcija u vašoj `config/app.php` konfiguracijskoj datoteci određuje koliko se informacija o grešci u stvari prikazuje korisniku. Prema podrazumijevanim postavkama, ova je opcija postavljena tako da poštuje vrijednost varijable `APP_DEBUG` okoline koja je pohranjena u vašoj `.env` datoteci.

#### UPOZORENJE:



Za lokalni razvoj trebali biste postaviti `APP_DEBUG` varijablu okoline na `true`. U vašoj produkcijskoj okolini ova bi vrijednost uvijek trebala biti `false`. Ako je varijabla postavljena na `true` u produkciji, rizikujete izlaganje osjetljivih konfiguracijskih vrijednosti krajnjim korisnicima vaše aplikacije.

### Održavanje kao način rada

Kada je vaša aplikacija u načinu rada za održavanje (engl. maintenance mode), prikazat će se prilagođeni prikaz za sve zahtjeve vaše aplikacije. To olakšava "onemogućavanje" vaše aplikacije dok se ažurira ili kada obavljate održavanje. Provjera načina održavanja uključena je u podrazumijevani stack za vašu aplikaciju. Ako je aplikacija u načinu rada za održavanje, `Symfony\Component\HttpFoundation\Exception\HttpException` instanca<sup>6</sup> će biti izbačena sa statusnim kodom 503.

Da biste omogućili način rada za održavanje, izvršite `down` Artisan naredbu:

```
php artisan down
```

Ako želite da `Refresh` HTTP zaglavlje se šalje sa svim odgovorima održavanja kao načina rada, možete koristiti opciju `refresh` kod pozivanja `down` naredbe. Zaglavlje `Refresh` će uputiti browser da automatski osvježi stranicu nakon određenog broja sekundi:

```
php artisan down --refresh=15
```

Također možete koristiti `retry` opciju za `down` naredbu, koja će biti postavljena kao `Retry-After` vrijednost HTTP zaglavlja, iako browser-i općenito ignoriraju ovo zaglavlje:

```
php artisan down --retry=60
```

---

<sup>6</sup> Nakon deklariranja klase potrebno je napraviti objekt. Taj postupak se naziva pravljenje instance ili instanciranje klase.

### Zaobilaženje održavanja kao načina rada

Da biste omogućili zaobilaženje održavanja kao načina rada (engl. bypassing maintenance mode) pomoću tajnog tokena, možete upotrijebiti opciju `secret` da bi ste odredili token za zaobilaženje održavanja kao načina:

```
php artisan down --secret="1630542a-246b-4b66-afa1-dd72a4c43515"
```

Nakon postavljanja aplikacije u način rada za održavanje, možete otići do URL-a aplikacije koji odgovara ovom tokenu i Laravel će vašem browseru izdati zaobilazni kolačić za zaobilaženje održavanja kao načina rada:

```
https://primjer.com/1630542a-246b-4b66-afa1-dd72a4c43515
```

Ako želite da Laravel generira tajni token za vas, možete upotrijebiti tu `with-secret` opciju. Tajna će vam se prikazati kada aplikacija bude u načinu rada za održavanje:

```
php artisan down --with-secret
```

Kada pristupate ovoj skrivenoj ruti, bit ćete preusmjereni (redirekcija) na /rutu aplikacije. Nakon što se kolačić izda vašem browser-u, moći ćete normalno pregledavati aplikaciju kao da nije u načinu rada za održavanje.

#### NAPOMENA:



Vaša tajna (engl. secret) održavanja kao načina rada obično bi se trebala sastojati od alfanumeričkih znakova i, po izboru, crtica. Trebali biste izbjegavati korištenje znakova koji imaju posebno značenje u URL-ovima kao što su `?` ili `&`.

### Način održavanja na više servera

Prema zadanim postavkama, Laravel određuje je li vaša aplikacija u načinu rada za održavanje (engl. maintenance mode) pomoću sistema temeljenog na datotekama. To znači da se za aktiviranje načina održavanja `php artisan down` naredba mora izvršiti na svakom serveru koji hostira vašu aplikaciju.

Alternativno, Laravel nudi metodu koja se temelji na metodi temeljenoj na keširanju za rukovanje u načinu rada za održavanje (engl. maintenance mode). Ova metoda zahtijeva pokretanje `php artisan down` naredbe na samo jednom serveru. Da biste koristili ovaj pristup, promijenite postavku "driver" u `config/app.php` datoteci svoje aplikacije u `cache`. Zatim odaberite keš store kojoj mogu pristupiti svi vaši serveri. Ovo osigurava dosljedan načinu rada u modu za održavanje (engl. maintenance mode) na svakom serveru:

```
'maintenance' => [  
    'driver' => 'cache',  
    'store' => 'database',  
],
```

### *Prethodan prikaz u održavanju kao načinu rada*

Ako koristite `php artisan down` naredbu tokom implementacije, vaši korisnici još uvijek mogu povremeno naići na greške ako pristupe aplikaciji dok se vaše zavisnosti o Composeru ili druge komponente infrastrukture ažuriraju. To se događa jer se značajan dio Laravel radne okoline mora pokrenuti kako bi se utvrdilo da je vaša aplikacija u održavanju kao načinu rada i prikazao (renderirao) prikaz (eng. view) održavanja kao načina rada pomoću mehanizma za izradu predložaka.

Iz tog razloga, Laravel vam omogućava da unaprijed renderirate prikaz održavanja kao načina rada koji će se vratiti na samom početku ciklusa zahtjeva. Ovaj se prikaz renderira prije nego što se bilo koja od zavisnosti vaše aplikacije učita. Možete unaprijed prikazati predložak po vlastitom izboru koristeći `down` naredbu `render` opcije:

```
php artisan down --render="errors::503"
```

### *Preusmjeravanje zahtjeva u održavanju kao načinu rada*

Dok je u održavanju kao načinu rada, Laravel će prikazati prikaz održavanja kao načina rada za sve URL-ove aplikacija kojima korisnik pokuša pristupiti. Ako želite, možete dati instrukciju Laravel-u da preusmjeri sve zahtjeve na određeni URL. To se može postići pomoću `redirect` opcije. Na primjer, možda želite preusmjeriti sve zahtjeve na `/` URI:

```
php artisan down --redirect=/"
```

### *Isključivanje održavanja kao načina rada*

Da bi ste isključili održavanje kao način rada, koristite `up` naredbu:

```
php artisan up
```

#### **NAPOMENA:**



Zadani predložak održavanja kao načina rada možete prilagoditi definiranjem vlastitog predloška na `resources/views/errors/503.blade.php`.

### *Održavanje kao način rada i redovi čekanja*

Dok je vaša aplikacija u održavanju kao načinu rada, neće se rukovati poslovima na čekanju (engl. queued jobs). Poslovi će se nastaviti normalno obrađivati nakon što aplikacija izađe iz održavanja kao načinu rada.

### *Alternative održavanju kao načinu rada*

Budući da održavanje kao način rada zahtijeva da vaša aplikacija ima nekoliko sekundi zastoja, razmislite o alternativama kao što su [Laravel Vapor](#) i [Envoyer](#) kako biste s Laravelom postigli implementaciju bez zastoja.

## Struktura direktorija

### Uvod

Zadana Laravel struktura aplikacije namijenjena je pružanju sjajne početne točke za velike i male aplikacije. Ali možete slobodno organizirati svoju prijavu kako god želite. Laravel ne nameće gotovo nikakva ograničenja na to gdje se neka klasa nalazi - sve dok Composer može automatski učitati klasu.

#### NAPOMENA:



Novi ste u Laravelu? Provjerite Laravel Bootcamp za praktični obilazak radne okoline dok vas vodimo kroz izradu vaše prve Laravel aplikacije.

### Root direktorij

#### *app direktorij*

Direktorij `app` sadrži osnovni kod vaše aplikacije. Uskoro ćemo detaljnije istražiti ovaj direktorij; međutim, gotovo sve klase u vašoj aplikaciji bit će u ovom direktoriju.

#### *bootstrap direktorij*

Direktorij `bootstrap` sadrži `app.php` datoteku koja pokreće (engl. bootstrap) radnu okolinu. Ovaj direktorij također sadrži `cache` direktorij koji sadrži datoteke generirane radnom okolinom za optimizaciju performansi kao što su ruta i datoteke keširanja usluga. Obično ne biste trebali mijenjati datoteke unutar ovog direktorija.

#### *config direktorij*

Direktorij `config`, kao što naziv implicira, sadrži sve konfiguracijske datoteke vaše aplikacije. Sjajna je ideja pročitati sve te datoteke i upoznati se sa svim opcijama koje su vam dostupne.

#### *database direktorij*

Direktorij `database` sadrži vaše migracije baza podataka, tvornice modela i početne vrijednosti. Ako želite, možete također koristiti ovaj direktorij za držanje SQLite baze podataka.

#### *public direktorij*

Direktorij `public` sadrži `index.php` datoteku koja je ulazna točka za sve zahtjeve koji ulaze u vašu aplikaciju i konfigurira automatsko učitavanje (engl. autoloading). Ovaj direktorij također sadrži vašu imovinu kao što su slike, JavaScript i CSS.

#### *resources direktorij*

`resources` direktorij sadrži vaše prikaze (engl. views) kao i vaše neobrađene, nekompajlirane resurse kao što su CSS ili JavaScript.

#### *routes direktorij*

`routes` direktorij sadrži sve definicije ruta za vašu aplikaciju. Prema podrazumijevanim postavkama, nekoliko datoteka rute uključeno je u Laravel: `web.php` i `console.php`.

Datoteka `web.php` sadrži rute koje Laravel smješta u `web` middleware grupu, koja pruža stanje sesije, CSRF zaštitu i enkripciju kolačića. Ako vaša aplikacija ne nudi RESTful API bez stanja (engl. stateless - ne sprema podatke klijenta generirane u jednoj sesiji za korištenje u sljedećoj sesiji), tada će sve vaše rute najvjerojatnije biti definirane u datoteci `web.php`.

`console.php` datoteka je mjesto gdje možete definirati sve vaše anonimne funkcije (engl. closure)<sup>7</sup> na temelju naredbe konzole. Svaka [anonimna funkcija \(engl. closure\)](#) vezana je za instancu naredbe što omogućava jednostavan pristup interakciji s IO metodama svake naredbe. Iako ova datoteka ne definira HTTP rute, ona definira ulazne točke (rute) temeljene na konzoli u vašoj aplikaciji. Također možete rasporediti zadatke u `console.php` datoteci.

Po izboru, možete instalirati dodatne datoteke ruta za API rute (`api.php`) i kanale emitiranja (`channels.php`), pomoću naredbi `install:api` i `install:broadcasting` Artisan.

Datoteka `api.php` sadrži rute koje Laravel smješta u `api` middleware<sup>8</sup> grupu. Namjera je da ove rute budu bez stanja (engl. stateless), tako da je za zahtjeve koji ulaze u aplikaciju tim rutama namijenjena provjera autentičnosti pomoću tokena i neće imati pristup stanju sesije.

Datoteka `channels.php` je mjesto gdje možete registrirati sve kanale za emitiranje događaja koje vaša aplikacija podržava.

### *storage direktorij*

`storage` direktorij sadrži vaše log-ove, kompajlirane Blade predloške (engl. templates), datoteke temeljene na sesijama, keš datoteke i druge datoteke koje generira radna okolina. Ovaj direktorij je podijeljen na `app`, `framework`, i `logs` direktorije. Direktorij `app` se može koristiti za pohranjivanje svih datoteka koje je generirala vaša aplikacija. Direktorij `framework` se koristi za pohranjivanje datoteka i keša radne okoline. Na kraju, `logs` direktorij sadrži datoteke log-ova vaše aplikacije.

`storage/app/public` direktorij se može koristiti za pohranjivanje datoteka koje su generirali korisnici, kao što su avatari profila, koji bi trebali biti javno dostupni. Trebali biste stvoriti simboličku vezu `public/storage` koja upućuje na ovaj direktorij. Poveznicu možete stvoriti pomoću `php artisan storage:link` Artisan naredbe.

### *tests direktorij*

`tests` direktorij sadrži vaše automatizirane testove. Primjeri `Pest` ili `PHPUnit` jediničnih testova i testova svojstava isporučuju se odmah. Svaka test klasa trebala bi imati sufiks s riječi `Test`. Svoje testove možete pokrenuti pomoću naredbi `phpunit` ili `php vendor/bin/phpunit`. Ili, ako želite detaljniji i ljepši prikaz rezultata testa, možete pokrenuti svoje testove pomoću `php artisan test` Artisan naredbe.

### *vendor direktorij*

Direktorij `vendor` sadrži vaše zavisnosti (engl. dependencies) o Composeru.

### *app direktorij*

Većina vaših aplikacija nalazi se u `app` direktoriju. Prema podrazumijevanim postavkama, ovaj direktorij je raspoređen pod `App` i Composer ga automatski (engl. autoloaded) učitava pomoću [PSR-4 standarda za automatsko učitavanje](#).

Prema zadanim postavkama, `app` direktorij sadrži niz dodatnih direktorija kao što su, `Http`, `Models` i `Providers`. Međutim, tokom vremena, razni drugi direktoriji će se generirati unutar direktorija

---

<sup>7</sup> Anonimne funkcije (engl. closures) dopuštaju stvaranje funkcija koje nemaju specificirano ime. Najkorisniji su kao vrijednost parametara koji se mogu pozvati, ali imaju i mnoge druge namjene. Closure se se također mogu koristiti kao vrijednosti varijabli; PHP automatski pretvara takve izraze u instance interne klase Closure. Closure također mogu naslijediti varijable iz nadređenog opsega. Closure je klasa.

<sup>8</sup> Middleware je softver koji se nalazi između operativnog sistema i aplikacija koje se na njemu izvode. U osnovi funkcionirajući kao skriveni sloj prevođenja, međuprogramska oprema omogućava komunikaciju i upravljanje podacima za distribuirane aplikacije.

aplikacije dok koristite `make` Artisan naredbe za generiranje klasa. Tako, na primjer, `app/Jobs` direktorij neće postojati sve dok ne izvršite `make:command` naredbu Artisan za generiranje klase posla.

I direktoriji `Console` i `Http` dodatno su objašnjeni u svojim odlomcima u nastavku, ali zamislite da direktoriji `Console` i `Http` pružaju API u jezgri vaše aplikacije. HTTP protokol i CLI su mehanizmi za interakciju s vašom aplikacijom, ali zapravo ne sadrže logiku aplikacije. Drugim riječima, to su dva načina izdavanja naredbi vašoj aplikaciji. Direktorij `Console` sadrži sve vaše Artisan naredbe, dok `Http` direktorij sadrži vaše kontrolere, middleware i zahtjeve.

**NAPOMENA:**

Mnoge klase u `app` direktoriju može generirati Artisan pomoću naredbi. Da biste pregledali dostupne naredbe, pokrenite `php artisan list make` naredbu na svom terminalu.

### *Broadcasting direktorij*

`Broadcasting` direktorij sadrži sve klase kanala emitiranja za vašu aplikaciju. Ove klase se generiraju pomoću `make:channel` naredbe. Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas kada stvorite svoj prvi kanal. Kako biste saznali više o kanalima, pogledajte dokumentaciju o prijenosu događaja.

### *Console direktorij*

Direktorij `Console` sadrži sve prilagođene Artisan naredbe za vašu aplikaciju. Ove se naredbe mogu generirati pomoću `make:command` naredbe.

### *Events direktorij*

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će ga za vas kreirati Artisan naredbe `event:generate` i `make:event`. U direktoriju `Events` se nalaze klase događaja (engl. Event classes). Događaji se mogu koristiti za upozorenje drugim dijelovima vaše aplikacije da se određena radnja dogodila, pružajući veliku fleksibilnost i odvajanje.

### *Exceptions direktorij*

`Exceptions` direktorij sadrži sve prilagođene iznimke za vašu aplikaciju. Ove se iznimke mogu generirati pomoću `make:exception` naredbe.

### *Http direktorij*

Direktorij `Http` sadrži vaše kontrolere, middleware i zahtjeve formi. Gotovo sva logika za upravljačkog sklopa za rukovanje zahtjevima (engl. handle requests) koji ulaze u vašu aplikaciju bit će smještena u ovaj direktorij.

### *Jobs direktorij*

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas ako izvršite Artisan naredbu `make:job`. `Jobs` direktorij sadrži poslove koji su u redu čekanja (engl. queueable jobs) za vašu aplikaciju. Vaša aplikacija može staviti poslove u red čekanja ili se mogu izvoditi sinkrono unutar trenutnog životnog ciklusa zahtjeva. Poslovi koji se izvode sinkrono tokom trenutnog zahtjeva ponekad se nazivaju „naredbe“ budući da su implementacija [uzorka naredbi](#).

### *Listeners direktorij*

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas ako izvršite Artisan naredbu `event:generate` ili `make:listener`. Direktorij `Listeners` sadrži klase koje rukuju vašim događajima. Slušačelji (engl. Listeners) događaja primaju instancu događaja i izvode logiku kao odgovor na događaj koji se pokreće. Na primjer, `UserRegistered` događajem može upravljati slušačelj `SendWelcomeEmail`.

### *Mail direktorij*

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas ako izvršite Artisan naredbu `make:mail`. `Mail` direktorij sadrži sve vaše klase koje predstavljaju e-poštu koju šalje vaša aplikacija. Objekti pošte omogućuju vam da obuhvatite svu logiku izgradnje e-pošte u jednu, jednostavnu klasu koja se može poslati pomoću `Mail::send` metode.

### *Models direktorij*

`Models` direktorij sadrži sve vaše Eloquent klase modela. Eloquent ORM uključen u Laravel pruža prekrasnu, jednostavnu ActiveRecord implementaciju za rad s vašom bazom podataka. Svaka tablica baze podataka ima odgovarajući "Model" koji se koristi za interakciju s tom tablicom. Modeli vam omogućavaju postavljanje upita (engl. query) za podatke u vašim tablicama, kao i umetanje novih slogova u tablicu.

### *Notifications direktorij*

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas ako izvršite Artisan naredbu `make:notification`. `Notifications` direktorij sadrži sve „transakcijske“ obavijesti koje šalje vaša aplikacija, kao što su jednostavne obavijesti (notifikacije) o događajima koji se događaju unutar vaše aplikacije. Laravel-ova svojstvo notifikacije apstrahira slanje obavijesti pomoću različitih driver-a kao što su e-pošta, Slack, SMS ili pohranjenih u bazi podataka.

### *Policies direktorij*

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas ako izvršite Artisan naredbu `make:policy`. `Policies` direktorij sadrži klase politike autorizacije (engl. authorization policy classes) za vašu aplikaciju. Pravila se koriste za određivanje može li korisnik izvršiti određenu radnju protiv resursa.



### Providers direktorij

**Providers** direktorij sadrži sve pružatelje usluga (engl. service providers) za vašu aplikaciju. Pružatelji usluga pokreću vašu aplikaciju povezivanjem usluga u kontejneru usluge, registracijom događaja ili izvođenjem bilo kojih drugih zadataka kako bi pripremili vašu aplikaciju za dolazne zahtjeve.

U novoj Laravel aplikaciji, ovaj direktorij će već sadržavati nekoliko pružatelja usluga. Po potrebi možete slobodno dodati svoje pružatelje u ovaj direktorij.

### Rules direktorij

Ovaj direktorij ne postoji prema podrazumijevanim postavkama, ali će biti kreiran za vas ako izvršite `make:rule` Artisan naredbu. **Rules** direktorij sadrži objekte s prilagođenim pravilima validacije za vašu aplikaciju. Pravila se koriste za ućahurenje (engl. encapsulate) komplicirane logike provjere validacije u jednostavan objekt. Za više informacija pogledajte dokumentaciju za provjeru validacije.

## Frontend

### Uvod

Laravel je backend radna okolina koja omogućava sve karakteristike koje su vam potrebne za izgradnju modernih web aplikacija, kao što su usmjeravanje, provjera validnosti, keširanje, redove čekanja (engl. queues), pohranu datoteka i još više. Međutim, ekipa iz Laravel-a vjeruje da je važno programerima ponuditi prekrasan full-stack doživljaj, uključujući moćne pristupe za izgradnju vašeg aplikacijskog frontend-a.

Postoje dva primarna načina za rješavanje razvoja frontend-a kada izrađujete aplikacije s Laravelom, a koji pristup odabirete ovisi o tome želite li izgraditi svoji frontend korištenjem PHP-a ili korištenjem JavaScript radne okoline kao što su Vue i React. U nastavku ćemo razmotriti obje ove mogućnosti kako biste mogli donijeti informiranu odluku o najboljem pristupu razvoju frontend-a za svoju aplikaciju.

### Korištenje PHP-a

#### PHP i Blade

U prošlosti je većina PHP aplikacija prikazivala HTML u browser-u pomoću jednostavnih HTML predložaka prošaranih PHP `echo` izjavama koje su prikazivale podatke koji su dohvaćeni iz baze podataka tokom zahtjeva:

```
<div>
    <?php foreach ($users as $user): ?>
        Hello, <?php echo $user->name; ?> <br />
    <?php endforeach; ?>
</div>
```

U Laravelu se ovaj pristup prikazivanju HTML-a još uvijek može postići korištenjem pogleda (engl. views) i Bladea. Blade je iznimno lagan jezik za izradu predložaka koji pruža praktičnu, kratku sintaksu za prikaz podataka, iteraciju kroz podatke i više od toga:

```
<div>
    @foreach ($users as $user)
        Hello, {{ $user->name }} <br />
    @endforeach
</div>
```

&lt;/div&gt;

Kada se aplikacije izrađuju na ovaj način, podnošenje formi i druge interakcije sa stranicama obično primaju potpuno novi HTML dokument sa servera, a browser ponovno prikazuje cijelu stranicu. Čak i danas, mnoge aplikacije mogu biti savršeno prikladne za konstruirane svojih frontend-ova na ovaj način pomoću jednostavnih Blade predložaka.

### *Rastuća očekivanja*

Međutim, kako su očekivanja korisnika u pogledu web aplikacija sazrijevala, mnogi programeri su otkrili potrebu za izgradnjom dinamičnijih frontend-ova s interakcijama koje izgledaju ugrađenije. U svjetlu toga, neki programeri odlučuju početi graditi frontend-ove svojih aplikacija koristeći JavaScript radne okoline kao što su Vue i React.

Drugi, radije držeći se backend jezika koji im odgovara, razvili su rješenja koja omogućuju izgradnju suvremenih UI-ova web aplikacija, dok još uvijek prvenstveno koriste svoj backend jezik po izboru. Na primjer, u [Rails](#) ekosistemu, to je potaknulo stvaranje biblioteka kao što su [Turbo Hotwire](#) i [Stimulus](#).

Unutar ekosistema Laravel, potreba za stvaranjem modernih, dinamičnih frontend-ova primarno korištenjem PHP-a dovela je do stvaranja [Laravel Livewire](#) i [Alpine.js](#).

### *Livewire*

Laravel Livewire je radna okolina za izgradnju frontend-a pokretanih Laravelom koja djeluju dinamično, moderno i živo baš kao frontend-ovi izgrađeni s modernim JavaScript radnim okolinama kao što su Vue i React.

Kada koristite Livewire, stvorit ćete Livewire „komponente“ koje renderiraju diskretni dio vašeg UI i izlažu metode i podatke koji se mogu pozvati i s kojima se može komunicirati iz vašeg aplikacijskog frontenda. Na primjer, jednostavna komponenta "Counter" može izgledati ovako:

```
<?php

namespace App\Http\Livewire;

use Livewire\Component;

class Counter extends Component
{
    public $count = 0;

    public function increment()
    {
        $this->count++;
    }

    public function render()
```

```
{  
    return view('livewire.counter');  
}  
}
```

A odgovarajući predložak za brojač bio bi napisan ovako:

```
<div>  
    <button wire:click="increment"></button>  
    <h1>{{ $count }}</h1>  
</div>
```

Kao što možete vidjeti, Livewire vam omogućava pisanje novih HTML atributa kao što je `wire:click` koji povezuju frontend i backend vaše Laravel aplikacije. Osim toga, možete prikazati trenutno stanje svoje komponente pomoću jednostavnih Blade izraza.

Za mnoge, Livewire je revolucionirao razvoj frontend-a s Laravelom, omogućujući im da ostanu unutar udobnosti Laravela dok konstruiraju moderne, dinamične web aplikacije. Tipično, programeri koji koriste Livewire također će koristiti [Alpine.js](#) za "posipanje" JavaScripta na svoj frontend samo tamo gdje je to potrebno, kao što je za renderiranje dijalognog prozora.

Ako ste novi u Laravelu, preporučujemo da se upoznate s osnovnom upotrebom pogleda (engl. views) i Bladea. Zatim konzultirajte službenu [Laravel Livewire dokumentaciju](#) kako biste saznali kako podići svoju aplikaciju na viši nivo s interaktivnim Livewire komponentama.

### *Kompleti za početnike*

Ako želite izgraditi svoj frontend pomoću PHP-a i Livewire-a, možete iskoristiti naše početne komplete Breeze ili Jetstream kako biste pokrenuli razvoj svoje aplikacije. Oba ova početna kompleta pružaju skelu (engl. Scaffolding) backendu i frontendu vaše aplikacije koristeći Blade i [Tailwind](#) tako da jednostavno možete početi graditi svoju sljedeću veliku ideju.

### Korištenje Vue / React

Iako je moguće izgraditi moderni frontend koristeći Laravel i Livewire, mnogi programeri još uvijek radije koriste snagu JavaScript radne okoline kao što su Vue ili React. To programerima omogućava da iskoriste prednosti bogatog ekosistema JavaScript paketa i alata dostupnih pomoću NPM-a.

Međutim, bez dodatnog alata, uparivanje Laravela s Vueom ili Reactom ostavilo bi nam potrebu za rješavanjem niza kompliciranih problema kao što su routing na strani klijenta, hidratacija podataka<sup>9</sup> i autentifikacija. Usmjeravanje na strani klijenta često se pojednostavljuje korištenjem samouvjerenih

---

<sup>9</sup> Hidracija podataka ili hidratacija jezera podataka je uvoz podataka u objekt. Kada objekt čeka da ga podaci popune, ovaj objekt čeka da bude hidratiziran. Izvor te hidratacije može biti jezero podataka ili drugi izvor podataka. U web razvoju, hidratacija je tehnika u kojoj JavaScript na strani klijenta pretvara statičnu HTML web stranicu, isporučenu pomoću statičkog hostinga ili renderiranja na strani servera, u dinamičku web stranicu pričvršćivanjem rukovatelja događajima na HTML elemente.

Vue/React okvira kao što su [Nuxt](#) i [Next](#); međutim, hidratacija podataka i autentifikacija ostaju komplicirani i glomazni problemi koje treba riješiti kada uparujete backend radne okoline kao što je Laravel s ovim frontend radnim okolinama.

Osim toga, programeri moraju održavati dva odvojena repozitorija koda, često moraju koordinirati održavanje, verzije i implementacije u oba repozitorija. Iako ti problemi nisu nepremostivi, u Laravel-u ne vjerujemo da je to produktivan ili ugodan način razvoja aplikacija.

### *Inertia*

Srećom, Laravel nudi najbolje od oba svijeta. [Inertia](#) premošćuje jaz između vaše Laravel aplikacije i vašeg modernog Vue ili React sučelja, omogućujući vam da izgradite potpuna, moderni frontend koristeći Vue ili React dok iskorištavate Laravel rute i kontrolere za usmjeravanje, hidrataciju podataka i autentifikaciju — sve unutar jednog repozitorija koda. S ovim pristupom možete uživati u punoj snazi i Laravela i Vue / Reacta bez osakaćivanja mogućnosti bilo kojeg alata.

Nakon instaliranja Inertie u vašu Laravel aplikaciju, pisat ćete rute i kontrolere kao i obično. Međutim, umjesto vraćanja Blade predloška s vašeg kontrolera, vratit ćete Inertia stranicu:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Inertia\Inertia;
use Inertia\Response;

class UserController extends Controller
{
    /**
     * Prikažite profil određenog korisnika
     */
    public function show(string $id): Response
    {
        return Inertia::render('Users/Profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Inertia stranica odgovara Vue ili React komponenti, obično pohranjenoj unutar `resources/js/Pages` direktorija vaše aplikacije. Podaci dani stranici pomoću `Inertia::render` metode koristit će se za hidrataciju<sup>10</sup> "props" komponente stranice:

```
<script setup>
import Layout from '@/Layouts/Authenticated.vue';
import { Head } from '@inertiajs/vue3';

const props = defineProps(['user']);
</script>

<template>
  <Head title="User Profile" />

  <Layout>
    <template #header>
      <h2 class="font-semibold text-xl text-gray-800 leading-tight">
        Profile
      </h2>
    </template>

    <div class="py-12">
      Hello, {{ user.name }}
    </div>
  </Layout>
</template>
```

Kao što možete vidjeti, Inertia vam omogućava da iskoristite punu snagu Vue ili Reacta kada gradite svoje frontend, istovremeno pružajući lagani most između pozadine koju pokreće Laravel i interface koje pokreće JavaScript.

### *Renderiranje na strani servera*

Ako ste zabrinuti oko uranjanja u Inertia jer vaša aplikacija zahtijeva prikaz sa servera, ne brinite. Inertia nudi podršku za [renderiranje na strani servera](#). A kada implementirate svoju aplikaciju pomoću [Laravel Forgea](#), lako je osigurati da Inertia proces renderiranja na strani servera uvijek radi.

### *Kompleti za početnike*

Ako želite izgraditi svoji frontend koristeći Inertia i Vue / React, možete iskoristiti naše komplete za početnike Breeze ili Jetstream kako biste pokrenuli razvoj svoje aplikacije. Oba ova kompleta za početnike pružaju skelu (engl. scaffold) vašom frontendu i backendu aplikacije autentifikacije toka vaše koristeći Inertia, Vue / React, [Tailwind](#) i [Vite](#) tako da možete početi graditi svoju sljedeću veliku ideju.

---

<sup>10</sup> Hidratacija se odnosi na proces popunjavanja (ili "punjenja") svojstava modela podacima iz baze podataka. Kada se podaci dohvate iz baze putem Eloquent ORM-a (Object-Relational Mapping), Laravel automatski stvara instance modela i "hidratizira" ih podacima koji su dobijeni iz baze.

### Paket materijala (engl. Asset bundling)

Bez obzira odlučite li razviti svoj frontend koristeći Blade i Livewire ili Vue / React i Inertia, vjerojatno ćete morati spojiti vaš aplikacijski CSS u materijal spremna za produkciju (sredstva, engl. assets)<sup>11</sup>. Naravno, ako odlučite izgraditi svoj aplikacijski frontend s Vue ili Reactom, također ćete morati spojiti svoje komponente u browser spreman za JavaScript imovinu.

Prema podrazumijevanim postavkama, Laravel koristi Vite za pakiranje vaših materijala. Vite pruža munjevito brzo vrijeme izrade i gotovo trenutnu zamjenu vrućih modula (engl. Hot Module Replacement - HMR) tokom lokalnog razvoja. U svim novim Laravel aplikacijama, uključujući one koje koriste naše pakete za početnike, pronaći ćete `vite.config.js` datoteku koja učitava naš jednostavni Laravel Vite dodatak koji čini Vite užitkom koristiti s Laravel aplikacijama.

Najbrži način da počnete koristiti Laravel i Vite je započinjanje razvoja vaše aplikacije pomoću Laravel Breeze, našeg najjednostavnijeg kompleta za početnike koji pokreće vašu aplikaciju pružajući skele za autentifikaciju frontend-a i backend-a.

#### NAPOMENA:



Za detaljniju dokumentaciju o korištenju Vitea u kombinaciji s Laravelom, pogledajte našu namjensku dokumentaciju o povezivanju i kompajliranju vaše imovine.

### Paketi za početnike

#### Uvod

Kako bismo vam omogućili početak izgradnje vaše nove Laravel aplikacije, sa zadovoljstvom vam nudimo autentifikaciju i početne pakete aplikacija. Ovi setovi automatski postavljaju vašu aplikaciju s rutama, kontrolerima i prikazima (engl. views) koji su vam potrebni za registraciju i autentifikaciju korisnika vaše aplikacije.

Iako možete slobodno koristiti ove početne komplete, oni nisu potrebni. Slobodno možete izraditi vlastitu aplikaciju iz temelja jednostavnom instalacijom nove kopije Laravel-a. U svakom slučaju, znamo da ćete izgraditi nešto sjajno!

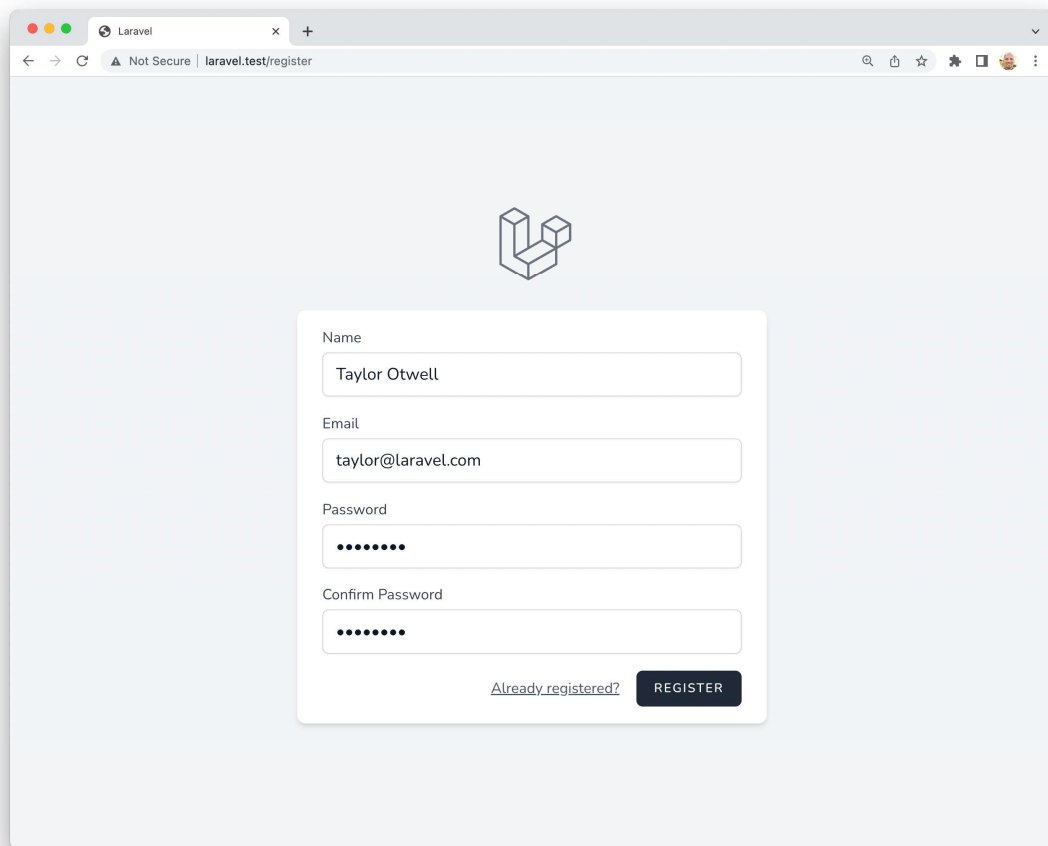
#### Laravel Breeze

Laravel Breeze je minimalna, jednostavna implementacija svih Laravel-ovih svojstava provjere autentičnosti, uključujući prijavu (logiranje), registraciju, reset šifre, provjeru e-pošte i potvrdu šifre. Osim toga, Breeze uključuje jednostavnu stranicu „profila“ na kojoj korisnik može ažurirati svoje ime, adresu e-pošte i šifru.

Zadani sloj prikaza (engl. view layer) Laravel Breeze sastoji se od jednostavnih Blade predložaka stiliziranih Tailwind CSS-om. Dodatno, Breeze nudi opcije koje pružaju skelu (engl. scaffolding) na temelju Livewire ili Inertia, uz izbor upotrebe Vue ili Reacta za skele temeljene na Inertia.

---

<sup>11</sup> Asset ili materijal je sve što vam je potrebno za izradu vašeg projekta (slike, tekst, video, audio itd.). Bundle ili paket je kada se te datoteke zatim komprimiraju zajedno u jednu datoteku.



### Laravel Bootcamp

Ako ste novi s Laravel-om, slobodno uskočite u Laravel Bootcamp. Laravel Bootcamp će vas provesti kroz izradu vaše prve Laravel aplikacije koristeći Breeze. To je sjajan način da obidete sve što Laravel i Breeze imaju za ponuditi.

### Instalacija

Najprije biste trebali izraditi novu Laravel aplikaciju, konfigurirati svoju bazu podataka i pokrenuti migracije baze podataka. Nakon što ste izradili novu Laravel aplikaciju, možete instalirati Laravel Breeze koristeći Composer:

```
composer require laravel/breeze --dev
```

Nakon što Composer instalira Laravel Breeze paket, možete pokrenuti `breeze:install` naredbu Artisan. Ova naredba objavljuje preglede provjere autentičnosti (engl. authentication views), rute, kontrolere i druge resurse u vašoj aplikaciji. Laravel Breeze objavljuje sav svoj kod u vašoj aplikaciji tako da imate potpunu kontrolu i pregled nad njegovim karakteristikama i implementacijom.

`breeze:install` naredba će od vas tražiti željeni frontend stack i testna radna okolina:

```
php artisan breeze:install
```

```
php artisan migrate  
npm install  
npm run dev
```

### *Breeze & Blade*

Zadani Breezeov „stack“ je Blade stack, koji koristi jednostavne Blade predloške za renderiranje frontend-a vaše aplikacije. Blade stack može se instalirati pozivanjem naredbe `breeze:install` bez drugih dodatnih argumenata i odabirom Blade frontend stack-a. Nakon instaliranja Breeze-ove skele (engl. scaffolding), trebali biste kompajlirati vaše materijale (engl. assets):

```
php artisan breeze:install  
  
php artisan migrate  
npm install  
npm run dev
```

Zatim se možete otići do svoje aplikacije `/login` ili `/register` URL-ovima u svom web browseru. Sve Breezeove rute definirane su unutar `routes/auth.php` datoteke.

#### **NAPOMENA:**



Kako biste saznali više o sastavljanju CSS-a i JavaScripta svoje aplikacije, pogledajte dokumentaciju Vite za Laravel.

### *Breeze & Livewire*

Laravel Breeze također nudi Livewire dodatne skele (engl. scaffolding). Livewire je moćan način izgradnje dinamičkih, reaktivnih, frontend UI koristeći samo PHP.

Livewire je odličan za timove koji primarno koriste Blade predloške i traže jednostavniju alternativu SPA radnim okolinama<sup>12</sup> pokretanim JavaScriptom kao što su Vue i React.

Da biste koristili Livewire stack, možete odabrati Livewire frontend stack kada izvršavate `breeze:install` Artisan naredbu. Nakon instaliranja Breezeove skele (engl. scaffolding), trebali biste pokrenuti migracije baze podataka:

```
php artisan breeze:install  
  
php artisan migrate
```

---

<sup>12</sup> Radna okolina aplikacije s jednom stranicom (engl. Single-Page Application Frameworks) je web aplikacija koja se korisniku prikazuje pomoću jedne HTML stranice kako bi bolje reagirala i bolje replicirala desktop aplikaciju ili native app (iOS ili Android)



### *Breeze & React / Vue*

Laravel Breeze također nudi React i Vue dodatnu skelu (engl. scaffolding), kroz implementaciju [Inertia](#) frontend-a. Inertia vam omogućava da izgradite moderne React i Vue aplikacije na jednoj stranici koristeći klasično rutanje na serverskoj strani i kontrolere.

Inertia vam omogućava da uživate u frontend snazi Reacta i Vuea u kombinaciji s nevjerojatnom pozadinskom produktivnošću Laravela i munjevito brzom [Vite](#) kompajleru. Da biste koristili Inertia stack, možete odabrati Vue ili React frontend stackove kada izvršavate `breeze:install` Artisan naredbu.

Kada odabirete Vue ili React frontend stack-a, Breeze instalacijski program također će vas pitati da odredite želite li podršku za [Inertia SSR](#)<sup>13</sup> ili TypeScript<sup>14</sup>. Nakon instaliranja Breezeove skele, trebali biste kompajlirati i vašu aplikacijsku frontset materijale (engl. assets):

```
php artisan breeze:install

php artisan migrate
npm install
npm run dev
```

Zatim se možete pomaknuti do svoje aplikacije `/login` ili `/register` URL-ovima u svom web browseru. Sve Breezeove rute definirane su unutar `routes/auth.php` datoteke.

### *Breeze & Next.js / API*

Laravel Breeze također može postaviti dodatnu skelu (engl. scaffolding) za autentifikacijski API koji je spreman za autentifikaciju modernih JavaScript aplikacija kao što su one koje pokreću Next, Nuxt i ostale. Za početak odaberite API stack kao željeni stack kada izvršavate `breeze:install` Artisan naredbe:

```
php artisan breeze:install

php artisan migrate
```

Tokom instalacije, Breeze će dodati `FRONTEND_URL` varijablu okoline u `.env` datoteku vaše aplikacije. Ovaj URL trebao bi biti URL vaše JavaScript aplikacije. To će obično biti `http://localhost:3000` tokom lokalnog razvoja. Osim toga, trebali biste osigurati da je vaš `APP_URL` postavljen na `http://localhost:8000`, što je zadani URL koji koristi `serve` Artisan naredba.

---

<sup>13</sup> Renderiranje na strani servera unaprijed renderira vaše JavaScript stranice na serveru, omogućujući vašim posjetiteljima da prime potpuno prikazani HTML kada posjete vašu aplikaciju. Budući da vaša aplikacija posluži potpuno prikazani HTML, tražilicama je također lakše indeksirati vašu stranicu.

<sup>14</sup> Typescript dodaje sintaksu iznad JavaScript-a omogućavajući programerima dodavanje tipova podataka jer kod statički tipiziranih jezika tip podatka varijable mora specificirati tip podataka varijable za vrijeme njegove deklaracije. Moramo unaprijed definirati tip povrata funkcije kao i tip varijable koju uzima ili prihvata za daljnje procjene. TypeScript omogućava određivanje tipa podataka koji se proslijeđuju unutar koda i ima mogućnost izvještavanja o greškama kada se tipovi ne podudaraju.

### *Next.js referentna implementacija*

Konačno, spremni ste upariti ovaj backend s frontend-om po vašem izboru. Sljedeća referentna implementacija Breeze frontend-a [dostupna je na GitHubu](#). Ovaj backend održava Laravel i sadrži isti UI kao i tradicionalni stacks Blade i Inertia koje nudi Breeze.

### Laravel Jetstream

Dok Laravel Breeze pruža jednostavnu i minimalnu početnu točku za izradu Laravel aplikacije, Jetstream povećava tu funkcionalnost s robusnijim mogućnostima i dodatnim frontend skupovima tehnologije. Za one koji su tek počeli sudjelovati u Laravel-u, preporučujemo da nauče osnove uz Laravel Breeze prije nego prijeđu na Laravel Jetstream.

Jetstream pruža lijepo dizajniranu aplikacijsku skelu (engl. Scaffolding) za Laravel i uključuje prijavu, registraciju, provjeru e-pošte, dvofaktorsku autentifikaciju, upravljanje sesijom, API podršku pomoću Laravel Sanctum-a i opcionalno upravljanje timom. Jetstream je dizajniran korištenjem [Tailwind CSS-a](#) i nudi vaš izbor [Livewire](#) ili [Inertia](#) vođene frontend skele.

Potpuna dokumentacija za instaliranje Laravel Jetstreama može se pronaći [unutar službene Jetstream dokumentacije](#).

## Implementacija

### Uvod

Kada budete spremni implementirati svoju Laravel aplikaciju u produkciju, postoje neke važne stvari koje možete učiniti kako biste bili sigurni da vaša aplikacija radi što je moguće učinkovitije. U ovom ćemo poglavlju pokriti neke sjajne početne točke za osiguravanje da je vaša Laravel aplikacija pravilno postavljena.

### Zahtjevi servera

Laravel radna okolina ima nekoliko sistemskih zahtjeva. Trebali biste osigurati da vaš web server ima sljedeću minimalnu PHP verziju i ekstenzije:

- PHP >= 8.1
- ctype PHP ekstenzija
- cURL PHP ekstenzija
- DOM PHP proširenje
- Fileinfo PHP ekstenzija
- Filtrirajte PHP proširenje
- Hash PHP ekstenzija
- Mbstring PHP proširenje
- OpenSSL PHP proširenje
- PCRE PHP proširenje
- PDO PHP proširenje
- Sesijska PHP ekstenzija
- Tokenizer PHP proširenje
- XML PHP ekstenzija

### Konfiguracija servera

#### Nginx

Ako implementirate svoju aplikaciju na serveru koji pokreće Nginx, možete koristiti sljedeću konfiguracijsku datoteku kao početnu točku za konfiguriranje vašeg web servera. Najvjerojatnije će ovu datoteku trebati prilagoditi ovisno o konfiguraciji vašeg servera. Ako biste željeli pomoć u upravljanju svojim serverom, razmislite o korištenju usluge upravljanja i postavljanja Laravel servera kao što je [Laravel Forge](#).

Provjerite, poput konfiguracije u nastavku, vaš web server usmjerava sve zahtjeve na `public/index.php` datoteku vaše aplikacije. Nikada ne biste trebali pokušati premjestiti `index.php` datoteku u početni direktorij vašeg projekta, jer će posluživanje aplikacije iz početnog direktorija projekta izložiti mnoge osjetljive konfiguracijske datoteke javnom Internetu:

```
server {  
    listen 80;  
    listen [::]:80;  
    server_name primjer.com;  
    root /srv/primjer.com/public;  
  
    add_header X-Frame-Options "SAMEORIGIN";  
    add_header X-Content-Type-Options "nosniff";
```

```
index index.php;

charset utf-8;

location / {
    try_files $uri $uri/ /index.php?$query_string;
}

location = /favicon.ico { access_log off; log_not_found off; }
location = /robots.txt { access_log off; log_not_found off; }

error_page 404 /index.php;

location ~ /\.php$ {
    fastcgi_pass unix:/var/run/php/php8.2-fpm.sock;
    fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
    include fastcgi_params;
}

location ~ /\.(!well-known).* {
    deny all;
}
}
```

### FrankenPHP

[FrankenPHP](#) se također može koristiti za posluživanje vaših Laravel aplikacija. FrankenPHP je moderan PHP server aplikacija napisan u Go programskom jeziku. Za posluživanje Laravel PHP aplikacije koristeći FrankenPHP, možete jednostavno pozvati njegovu `php-server` naredbu:

```
frankenphp php-server -r public/
```

Kako biste iskoristili moćnija svojstva koje podržava FrankenPHP, kao što je njegova Laravel Octane integracija, HTTP/3, moderna kompresija ili mogućnost pakiranja Laravel aplikacija kao samostalnih binarnih datoteka, pogledajte FrankenPHP-a [Laravel dokumentaciju](#).

### Dopuštenja direktorija

Laravel će morati pisati u direktorije `bootstrap/cache` i `storage`, pa biste trebali osigurati da vlasnik procesa web servera ima dozvolu za pisanje u te direktorije.

## Optimizacija

### *Autoloader konfiguracija*

Prilikom implementacije vaše aplikacije u produkciju, trebali biste provjeriti jeste li pokrenuli naredbu `config:cache` Artisan tokom procesa implementacije:

```
php artisan config:cache
```

Ova naredba će kombinirati sve Laravelove konfiguracijske datoteke u jednu, keširanu datoteku, što uvelike smanjuje broj putova koje radna okolina mora napraviti do datotečnog sistema prilikom učitavanja vaših konfiguracijskih vrijednosti.

**UPOZORENJE:**



Ako izvršite naredbu `config:cache` tokom procesa implementacije, trebali biste biti sigurni da pozivate `env` funkciju samo iz vaših konfiguracijskih datoteka. Nakon što se konfiguracija predmemorira, datoteka `.env` neće se učitati i svi pozivi funkciji `env` za varijable `.env` vratit će `null`.

### *Keširanje događaja (engl. events)*

Trebali biste keširati automatski otkriveni događaj (engl. event discovery), svoje aplikacije u mapiranja slušatelja (engl. listener mappings) tokom procesa implementacije. To se može postići pozivanjem naredbe `event:cache Artisan` tokom implementacije:

```
php artisan event:cache
```

### *Keširanje ruta*

Ako gradite veliku aplikaciju s mnogo ruta, trebali biste provjeriti pokrećete li Artisan naredbu `route:cache` tokom procesa implementacije:

```
php artisan route:cache
```

Ova naredba reducira sve vaše registracije ruta u jedan poziv metode unutar keširane datoteke, poboljšavajući izvedbu registracije ruta kod registriranja stotina ruta.

### *Keširanje pogleda*

Kada implementirate svoju aplikaciju u produkciju, provjerite jeste li pokrenuli `view:cache` Artisan naredbu tokom procesa implementacije:

```
php artisan view:cache
```

Ova naredba reducira sve vaše registracije ruta u jedan poziv metode unutar keširane datoteke, poboljšavajući izvedbu registracije ruta kod registriranja stotina ruta.

## Debug mod

Debug opcija `config/app.php` konfiguracijskoj datoteci određuje koliko se informacija o grešci stvarno prikazuje korisniku. Prema podrazumijevanim postavkama, ova je opcija postavljena tako da poštuje vrijednost `APP_DEBUG` varijable okoline koja je pohranjena u `.env` datoteci vaše aplikacije.

### UPOZORENJE:



U vašoj produkcijskoj okolini ova bi vrijednost uvijek trebala biti `false`. Ako je `APP_DEBUG` varijabla postavljena na `true` u produkciji, riskirate izlaganje osjetljivih konfiguracijskih vrijednosti krajnjim korisnicima vaše aplikacije.

## Ruta zdravlja

Laravel uključuje ugrađenu rutu provjere zdravlja koja se može koristiti za praćenje statusa vaše aplikacije. U produkciji se ova ruta može koristiti za izvješćivanje o statusu vaše aplikacije uređaju za praćenje neprekidnog rada, balansu opterećenja ili sistemu orkestracije kao što je Kubernetes.

Prema zadanim postavkama, ruta provjere zdravlja posluhuje se kod `/up` i vratit će HTTP 200 odgovor ako se aplikacija pokrenula bez iznimke. U suprotnom će biti vraćen odgovor HTTP 500. Možete konfigurirati URI za ovu rutu u `bootstrap/` datoteci aplikacije svoje aplikacije:

```
->withRouting(
    web: __DIR__.'/../routes/web.php',
    commands: __DIR__.'/../routes/console.php',
    health: '/up',
    health: '/status',
)
```

Kada se HTTP zahtjevi upućuju ovoj ruti, Laravel će također poslati `Illuminate\Foundation\Events\DiagnosingHealth` događaj, omogućujući vam da izvršite dodatne zdravstvene provjere relevantne za vašu aplikaciju. Unutar slušača (engl. listener) za ovaj događaj možete provjeriti bazu podataka svoje aplikacije ili status keša. Ako otkrijete problem sa svojom aplikacijom, možete jednostavno izbaciti iznimku od slušatelja.

## Jednostavna implementacija s Forge / Vapor

### Laravel Forge

Ako niste sasvim spremni upravljati vlastitom konfiguracijom servera ili vam nije ugodno konfigurirati sve različite usluge potrebne za pokretanje robusne Laravel aplikacije, Laravel Forge je izvrsna alternativa.

Laravel Forge može kreirati servere na različitim pružateljima infrastrukture (engl. providers) kao što su DigitalOcean, Linode, AWS i drugi. Osim toga, Forge instalira i upravlja svim alatima potrebnim za izgradnju robusnih Laravel aplikacija, kao što su Nginx, MySQL, Redis, Memcached, Beanstalk i još mnogo toga.

**NAPOMENA:**



Želite potpuni vodič za implementaciju s Laravel Forgeom? Pogledajte Laravel Bootcamp i Forge video seriju koja je dostupna na Laracasts.

*Laravel Vapor*

Ako želite platformu za implementaciju potpuno bez servera s automatskim skaliranjem podešenu za Laravel, pogledajte Laravel Vapor. Laravel Vapor je platforma za implementaciju bez servera za Laravel, koju pokreće AWS. Pokrenite svoju Laravel infrastrukturu na Vaporu i zaljubite se u skalabilnu jednostavnost bez servera. Kreatori Laravela su fino podesili Laravel Vapor za besprijekoran rad s radnom okolinom tako da možete nastaviti pisati svoje Laravel aplikacije točno onako kako ste navikli.

## Koncepti arhitekture

### Životni ciklus zahtjeva

#### Uvod

Kada koristite bilo koji alat u „stvarnom svijetu“, osjećate se sigurnije ako razumijete kako taj alat radi. Razvoj aplikacija nije drugačiji. Kada shvatite kako funkcioniraju vaši razvojni alati, osjećate se ugodnije i sigurnije ih upotrebljavati.

Cilj ovog dokumenta je dati vam dobar pregled na visokom nivou o tome kako funkcionira Laravel radna okolina. Boljim upoznavanjem cjelokupne radne okoline sve će se činiti manje „čarobnim“ i bit ćete sigurniji u izgradnji svojih aplikacija. Ako odmah ne razumijete sve pojmove, nemojte klonuti duhom! Samo pokušajte steći osnovni uvid u ono što se događa i vaše će znanje rasti kako budete istraživali druge dijelove dokumentacije.

#### Pregled životnog ciklusa

##### Prvi koraci

Ulazna točka za sve zahtjeve prema Laravel aplikaciji je datoteka `public/index.php`. Sve zahtjeve usmjerava na ovu datoteku konfiguracija vašeg web (Apache / Nginx) servera. Datoteka `index.php` ne sadrži mnogo koda. Umjesto toga, to je početna točka za učitavanje ostatka radne okoline.

Datoteka `index.php` učitava definiciju automatskog učitavanja koju je generirao Composer, a zatim dohvaća instancu Laravel aplikacije iz `bootstrap/app.php`. Prva radnja koju poduzima sam Laravel je kreiranje instance aplikacije/ kontejnera usluge.

##### HTTP / jezgre konzole

Zatim se dolazni zahtjev šalje ili HTTP jezgri (engl. kernel) ili jezgri konzole, ovisno o vrsti zahtjeva koji ulazi u aplikaciju. Ove dvije jezgre služe kao središnja lokacija kroz koju prolaze svi zahtjevi. Za sada se usredotočimo samo na HTTP jezgru, koji se nalazi u `app/Http/Kernel.php`.

HTTP jezgra proširuje `Illuminate\Foundation\Http\Kernel` klasu, koja definira matricu `bootstrappers` koji će se pokrenuti prije nego što se zahtjev izvrši. Ovi programi za pokretanje (engl. bootstrappers) konfiguriraju upravljački sklop za rukovanje greškama (engl. error handling), konfiguriraju logiranje, detektiraju aplikacijsku okolinu (engl. application environment) i izvode druge zadatke koje je potrebno obaviti prije nego što se zahtjevom rukuje (tj. stvarno se obradi). Obično ove klase rukuju unutrašnjom Laravel konfiguracijom o kojoj ne morate brinuti.

HTTP kernel također definira lista HTTP middleware kroz koji svi zahtjevi moraju proći prije nego što ih aplikacija obradi. Ovaj middleware rukuje čitanjem i pisanjem HTTP sesije, utvrđuje da li aplikacija u načinu održavanja (engl. maintenance mode), provjeru CSRF tokena i još toga. Uskoro ćemo više o njima.

Potpis metode za `handle` metodu HTTP kernela prilično je jednostavan: prima `Request` i vraća `Response`. Zamislite jezgru kao veliku crnu kutiju koja predstavlja cijelu vašu aplikaciju. Pošaljite mu HTTP zahtjeve i on će vratiti HTTP odgovore.

##### Pružatelji usluga

Jedna od najvažnijih radnji pokretanja jezgre je učitavanje pružatelja usluga (engl. service providers) za vašu aplikaciju. Pružatelji usluga odgovorni su za bootstrapping radne okoline, kao što su baza podataka, red čekanja (engl. queue), komponente provjere validnosti i usmjeravanja (engl. routing). Svi davatelji usluga za aplikaciju konfigurirani su u matrici `providers` konfiguracijske datoteke `config/app.php`.



Laravel će iterirati kroz ovaj listu pružatelja i instancirati<sup>15</sup> svakog od njih. Nakon instanciranja pružatelja, `register` metoda će biti pozvana na svim pružateljima (engl. providers). Zatim, nakon što su svi pružatelji registrirani, `boot` metoda će biti pozvana na svakom pružatelju usluga. To je tako da pružatelji usluga mogu ovisiti o tome da je svako vezivanje (engl. binding) kontejnera registrirano i dostupno do trenutka kada se njihova `boot` metoda izvrši.

U biti svako glavno svojstvo koju nudi Laravel je podignuto (engl. bootstrapped) i konfigurirano od strane davatelja usluga (engl. service providers). Budući da oni bootstrap-iraju i konfiguriraju toliko puno svojstava koje nudi radna okolina, pružatelji usluga su najvažniji aspekt cjelokupnog procesa bootstrap-a Laravela.

Iako radna okolina interno koristi desetke pružatelja usluga (engl. service providers), također imate mogućnost kreiranja vlastitog. U datoteci možete pronaći popis korisnički definiranih pružatelja usluga ili nezavisnih pružatelja usluga koje vaša aplikacija koristi u `bootstrap/providers.php` datoteci.

### *Usmjeravanje (engl. routing)*

Nakon što se aplikacija podigne (engl. bootstrapped) i svi davatelji usluga (engl. service providers) registriraju, Zahtjev će biti predan ruteru na slanje. Ruter će poslati zahtjev ruti ili kontroleru, kao i pokrenuti bilo koji middleware specifičan za rutu.

Middleware pruža prikladan mehanizam za filtriranje ili ispitivanje HTTP zahtjeva koji ulaze u vašu aplikaciju. Na primjer, Laravel uključuje middleware koji provjerava je li korisnik vaše aplikacije autentificiran. Ako korisnik nije autentificiran, middleware će ga preusmjeriti na ekran za prijavu. Međutim, ako je korisnik autentificiran, middleware će dopustiti da se zahtjev nastavi dalje u aplikaciju. Neki middleware su dodijeljeni svim rutama unutar aplikacije, kao što je `PreventRequestsDuringMaintenance`, dok su neki samo dodijeljeni određenim rutama ili grupama ruta. Možete saznati više o middleware čitanjem kompletne dokumentacije middleware.

Ako zahtjev prolazi kroz sve podudarne rute dodijeljene middleware-u, metoda rute ili kontrolera će se izvršiti, a odgovor vraćen od strane metode rute ili kontrolera bit će poslan natrag kroz lanac middleware rute.

### *Dovršavanje*

Jednom kada metoda rute ili kontrolera vrati odgovor, odgovor će putovati natrag kroz middleware rute, dajući aplikaciji priliku da modificira ili ispita odlazni odgovor.

Konačno, nakon što odgovor putuje natrag kroz middleware, `handle` metoda HTTP kernela vraća objekt odgovora i `index.php` datoteka poziva `send` metodu na vraćeni odgovor. Metoda `send` šalje sadržaj odgovora web browseru korisnika. Završili smo naše putovanje kroz cijeli životni ciklus Laravel zahtjeva!

---

<sup>15</sup> Objekt koji pripada klasi naziva se instance te klase. Kada konstruktor klase stvori objekt te klase, kažemo da je objekt instanca klase (klasa je instancirana), a članske varijable specifične za taj objekt nazivaju se varijablama instance. Varijable instance specifične su za stvoreni objekt, za razliku od varijabli klase, gdje sve instance dijele iste podatke.

## Usredotočite se na pružatelje usluga

Pružatelji usluga doista su ključ za bootstrapping Laravel aplikacije. Instanca aplikacije je kreirana, davatelji usluga su registrirani, a zahtjev je predan bootstrapped aplikaciji. Stvarno je tako jednostavno!

Vrlo je vrijedno imati čvrsto razumijevanje načina na koji se Laravel aplikacija gradi i bootstrap-a pomoću pružatelja usluga. Zadani pružatelji usluga vaše aplikacije pohranjeni su u `app/Providers` direktoriju.

Prema podrazumijevanim postavkama, `AppServiceProvider` je prilično prazan. Ovaj pružatelj je odlično mjesto za dodavanje vlastitog pokretanja (engl. Bootstrapping) aplikacije i vezanja kontejnera usluge. Za velike aplikacije, možda ćete htjeti stvoriti nekoliko pružatelja usluga, svaki s preciznijim pokretanjem (engl. Bootstrapping) za specifične usluge koje koristi vaša aplikacija.

## Kontejner usluga

### Uvod

Laravel kontejner usluge moćan je alat za upravljanje zavisnostima klasa i izvođenje ubrizgavanja zavisnosti. Injekcija zavisnosti je otmjena fraza koja u suštini znači ovo: zavisnosti klase se „injektiraju“ ili „ubrizgavaju“ u klasu kroz konstruktor ili, u nekim slučajevima, „setter“ metodu.

Pogledajmo jednostavan primjer:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Repositories\UserRepository;
use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Kreira novi kontroler instance.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Prikazuje profil za danog korisnika.
     */
    public function show(string $id): View
    {
        $user = $this->users->find($id);
```

```

        return view('user.profile', ['user' => $user]);
    }
}

```

U ovom primjeru, `UserController` potreban je da bi dohvatio korisnike iz izvora podataka. Dakle, injektirat ćemo uslugu koja može dohvatiti korisnike. U tom kontekstu, naš `UserRepository` najvjerojatnije koristi `Eloquent` za dohvaćanje korisničkih podataka iz baze podataka. Međutim, budući da je repozitorij umetnut, možemo ga lako zamijeniti drugom implementacijom. Također smo u mogućnosti jednostavno „oponašati“ ili stvoriti lažnu implementaciju `UserRepository` kada testiramo svoje aplikacije.

Duboko razumijevanje kontejnera usluge Laravel ključno je za izgradnju snažne, velike aplikacije, kao i za doprinos samoj Laravel jezgri.

### *Nulta rezolucija konfiguracije*

Ako klasa nema zavisnosti ili zavisi samo o drugim konkretnim klasama (ne o interface-ima), kontejner ne mora dobiti upute o tome kako riješiti tu klasu. Na primjer, možete staviti sljedeći kod u svoju `routes/web.php` datoteku:

```

<?php

class Service
{
    // ...
}

Route::get('/', function (Service $service) {
    die($service::class);
});

```

U ovom primjeru, odabir rute vaše aplikacije `/` automatski će riješiti `Service` klasu i injektirati je u upravljački sklop za rukovanje rutama (engl. route's handler). Ovo mijenja igru. To znači da možete razviti svoju aplikaciju i iskoristiti prednost uvođenja zavisnosti bez brige o napuhanim konfiguracijskim datotekama.

Srećom, mnoge klase koje ćete pisati kada izrađuje Laravel aplikacije automatski primaju svoje zavisnosti pomoću kontejnera, uključujući kontrolere, slušatelje događaja (engl. event listeners), middleware i više. Dodatno, možete nagovijestiti tip (engl. type-hint)<sup>16</sup> zavisnosti u `handle` metodi

<sup>16</sup> Nagovještaj tipa (engl. type hinting) ili PHP deklaracije tipa daje Vam mogućnost definiranja tipa koje se mogu proslijediti za svaki argument funkcije ili metode. Nagovještavanje tipa nije obvezno, ali kada se koristi, prisiljava argumente da budu određene vrste ili ne. Ako nisu, zaustavlja izvršavanje programa i izbacuje grešku. PHP deklaracije tipa (ili „type hints“ daju podatke o tipovima podataka funkcije i argumentima metoda. Ako je zadana vrijednost pogrešnog tipa, generirat će se `TypeError` greška. Da bi ste odredili deklaraciju tipa, ime tipa treba dodati prije naziva parametra. Deklaracija može biti napravljena tako da prihvati `NULL` vrijednosti ako je zadana vrijednost parametra postavljena na `NULL`.

poslova u čekanju (engl. *queued jobs*). Jednom kada okusite snagu automatike i mogućnost bez konfiguracije injektiranja zavisnosti, čini se da je nemoguće razvijati se bez njega.

### *Kada koristiti kontejner*

Zahvaljujući rješenju za nultom konfiguracijom, često ćete nagovijestiti tip (engl. *type-hint*) zavisnosti o rutama, kontrolerima, slušateljima događaja (engl. *event listeners*) i drugdje bez ikakve ručne interakcije sa kontejnerom. Na primjer, možete nagovijestiti tip (engl. *type-hint*) `Illuminate\Http\Request` objekt u svojoj definiciji rute tako da možete lako pristupiti trenutnom zahtjevu. Iako nikada ne moramo komunicirati sa kontejnerom da bismo napisali ovaj kod, on upravlja injektiranjem ovih zavisnosti iza scene:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

U mnogim slučajevima, zahvaljujući automatskom injektiranju zavisnosti i fasadama (engl. *facades*), možete izraditi Laravel aplikacije bez ručnog vezanja ili rješavanja bilo čega iz kontejnera. Dakle, kada biste ikada ručno stupili u interakciju sa kontejnerom? Razmotrimo dvije situacije.

Prvo, ako napišete klasu koja implementira interface i želite nagovijestiti tip (engl. *type-hint*) tog interface-a na ruti ili konstruktoru klase, morate reći kontejneru kako riješiti taj interface. Drugo, ako pišete Laravel paket koji planirate dijeliti s drugim Laravel programerima, možda ćete morati vezati usluge svog paketa u kontejner.

## Uvezivanje (engl. Binding)

### *Osnove uvezivanja*

#### *Jednostavno uvezivanje*

Gotovo sva vaša uvezivanja kontejnera usluga bit će registrirana unutar servera usluga, tako da će većina ovih primjera pokazati korištenje kontejnera u tom kontekstu.

Unutar pružatelja usluga, uvijek imate pristup kontejneru pomoću `$this->app` svojstva. Možemo registrirati uvezivanje korištenjem `bind` metode, prosljeđivanjem naziva klase ili interface-a koje želimo registrirati zajedno sa anonimnom funkcijom (engl. *closure*) koje vraća instancu klase:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->bind(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Primijetite da primamo sam kontejner kao argument za rješavač (engl. resolver). Zatim možemo koristiti kontejner za rješavanje pod-zavisnosti objekta koji gradimo.

Kao što je spomenuto, obično ćete komunicirati sa kontejnerom unutar pružatelja usluga; međutim, ako želite komunicirati sa kontejnerom izvan pružatelja usluga, to možete učiniti pomoću **App** fasade :

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function (Application $app) {
    // ...
});
```

Možete koristiti **bindIf** metodu za registraciju vezivanja kontejnera samo ako uvezivanje nije već registrirano za dani tip:

```
$this->app->bindIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

#### NAPOMENA:



Nema potrebe vezati (bind) klase u kontejner ako ne ovise ni o jednom interface-u. Kontejner ne treba biti instruiran u to kako izgraditi te objekte, budući da može automatski riješiti te objekte pomoću refleksije<sup>17</sup>.

#### Uvezivanje singleton-a

Metoda **singleton** uvezuje klasu ili interface u kontejner koji bi se trebao riješiti samo jednom. Nakon što se singleton<sup>18</sup> uvezivanje (engl. binding) riješi, ista će se instanca objekta vratiti na sljedeće pozive u kontejner:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->singleton(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

<sup>17</sup> Izraz „refleksija“ u razvoju softvera znači da program poznaje vlastitu strukturu tokom izvođenja i da je također može modificirati. Ova sposobnost se također naziva „introspekcija“. U području PHP-a refleksija se koristi kako bi se osigurala sigurnost tipa u programskom kodu.

<sup>18</sup> Singleton uzorak je uzorak dizajna softvera koji ograničava instanciranje klase na pojedinačnu instancu. Singleton uzorak omogućava objektima da: osigurava da imaju samo jednu instancu, omogućava jednostavan pristup toj instanci, kontrolira njihovu instancijaciju (na primjer, skrivanje konstruktora klase)

Možete upotrijebiti `singletonIf` metodu za registraciju singleton kontejnera povezanog samo ako povezivanje nije već registrirano za dani tip:

```
$this->app->singletonIf(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

#### *Uvezivanje dosjega singletons-a*

Metoda `scoped` veže klasu ili interface u kontejner koji bi trebao biti riješen samo jednom unutar zadanog Laravel zahtjeva/posla životnog ciklusa. Iako je ova metoda slična `singleton` metodi, instance registrirane pomoću `scoped` metode bit će ispražnjene svaki put kada Laravel aplikacija započne novi „životni ciklus“, kao što je kada Laravel Octane radnik obradi novi zahtjev ili kada Laravel radnik reda čekanja (engl. queue worker) obradi novi posao:

```
use App\Services\Transistor;
use App\Services\PodcastParser;
use Illuminate\Contracts\Foundation\Application;

$this->app->scoped(Transistor::class, function (Application $app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

#### *Uvezivanje instanci*

Također možete vezati postojeću instancu objekta u kontejner pomoću `instance` metode. Navedena instanca će uvijek biti vraćena na sljedećim pozivima u kontejner:

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);
```

#### *Povezivanje interfejsa s implementacijama*

Vrlo moćno svojstvo kontejnera usluga je njegova sposobnost vezanja interfejsa na danu implementaciju. Na primjer, pretpostavimo da imamo `EventPusher` interface i `RedisEventPusher` implementaciju. Nakon što smo kodirali našu `RedisEventPusher` implementaciju ovog interface-a, možemo ga registrirati u kontejneru usluge ovako:

```
use App\Contracts\EventPusher;
use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);
```

Ova izjava govori kontejneru da bi trebao injektirati `RedisEventPusher` kada klasa treba implementaciju `EventPusher`. Sada možemo nagovijestiti tip (engl. type-hint) `EventPusher` interface-u u konstruktoru klase koju je riješio kontejner. Upamtite, kontroleri, slušatelji događaja (engl. Event listeners), middleware i razne druge vrste klasa unutar Laravel aplikacija uvijek se rješavaju pomoću kontejnera:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 */
public function __construct(
    protected EventPusher $pusher
) {}
```

### *Kontekstualno uvezivanje*

Ponekad možete imati dvije klase koje koriste isti interface, ali želite injektirati različite implementacije u svaku klasu. Na primjer, dva kontrolera mogu ovisiti o različitim provedbama `Illuminate\Contracts\Filesystem\Filesystem` ugovora (engl. contract). Laravel pruža jednostavno, tečan interface za definiranje ovog ponašanja (engl. behavior):

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\UploadController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;
use Illuminate\Support\Facades\Storage;

$this->app->when(PhotoController::class)
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('local');
    });

$this->app->when([VideoController::class, UploadController::class])
    ->needs(Filesystem::class)
    ->give(function () {
        return Storage::disk('s3');
    });
```

### *Uvezivanje jednostavnih vrijednosti*

Ponekad možete imati klasu koja prima neke umetnute klase, ali također treba umetnutu jednostavnu vrijednost kao što je cijeli broj. Možete lako koristiti kontekstualno vezivanje za injektiranje bilo koje vrijednosti koja bi vašoj klasi mogla biti potrebna:

```
use App\Http\Controllers\UserController;

$this->app->when(UserController::class)
    ->needs('$variableName')
    ->give($value);
```

Ponekad klasa može ovisiti o matrici tagiranih instanci (engl. array of tagged instances). Koristeći ovu `giveTagged` metodu, možete jednostavno injektirati sva uvezivanja kontejnera s tim tagom:

```
$this->app->when(ReportAggregator::class)
    ->needs('$reports')
    ->giveTagged('reports');
```

Ako trebate injektirati vrijednost iz jedne od konfiguracijskih datoteka svoje aplikacije, možete koristiti `giveConfig` metodu:

```
$this->app->when(ReportAggregator::class)
    ->needs('$timezone')
    ->giveConfig('app.timezone');
```

### *Uvezivanje tipom određenih argumenata čiji je broj promjenjiv (engl. Binding Typed Variadics)*

Ponekad možete imati klasu koja prima matricu upisanih objekata koristeći promjenjivi (engl. variadic)<sup>19</sup> argument konstruktora:

```
<?php

use App\Models\Firewall;
use App\Services\Logger;

class Firewall
{
    /**
     * Filter instanci.
     *
     * @var array
     */
    protected $filters;
```

---

<sup>19</sup> Variadic function je funkcija sa neodređenim brojem argumenata (engl. arity).



```

/**
 * Kreira novu klasu instance.
 */
public function __construct(
    protected Logger $logger,
    Filter ...$filters,
) {
    $this->filters = $filters;
}
}

```

Upotrebom kontekstualnog uvezivanja (engl. contextual binding), ovu zavisnost možete riješiti tako da omogućite `give` metodi anonimnu funkciju (engl. closure) koje vraća matricu riješenih `Filter` instanci:

```

$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give(function (Application $app) {
        return [
            $app->make(NullFilter::class),
            $app->make(ProfanityFilter::class),
            $app->make(TooLongFilter::class),
        ];
    });

```

Radi praktičnosti, također možete samo dati matricu imena klasa koje će kontejner riješiti kad god `Firewall` zatreba `Filter` instance:

```

$this->app->when(Firewall::class)
    ->needs(Filter::class)
    ->give([
        NullFilter::class,
        ProfanityFilter::class,
        TooLongFilter::class,
    ]);

```

*Zavisnosti tagova argumenata čiji je broj promjenjiv (engl. Variadic Tag Dependencies)*

Ponekad klasa može imati varijabilnu zavisnost (engl. variadic dependency) koja je nagovjestila tip (engl. type-hint) kao data klasa (`Report ...$reports`). Koristeći `needs` i `giveTagged` metode, možete jednostavno injektirati sva vezivanja kontejnera s tim tagom za danu zavisnost:

```
$this->app->when(ReportAggregator::class)
    ->needs(Report::class)
    ->giveTagged('reports');
```

### Tagiranje (engl. Tagging)

Povremeno ćete možda trebati riješiti sve određene „kategorije“ uvezivanja. Na primjer, možda pravite analizator izvještaja koji prima matricu sa različitim **Report** implementacijama interface-a. Nakon registracije **Report** implementacije, možete im dodijeliti tag pomoću tag metode:

```
$this->app->bind(CpuReport::class, function () {
    // ...
});

$this->app->bind(MemoryReport::class, function () {
    // ...
});

$this->app->tag([CpuReport::class, MemoryReport::class], 'reports');
```

Nakon što su usluge označene (tagirane), možete ih sve jednostavno sve riješiti kroz **tagged** metodu kontejnera:

```
$this->app->bind(ReportAnalyzer::class, function (Application $app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

### Proširivanje uvezivanja

**extend** metoda omogućava modifikaciju riješenih usluga. Na primjer, kada je usluga riješena, možete pokrenuti dodatni kod za ukrašavanje ili konfiguraciju usluge. **extend** metoda prihvaća dva argumenta, klasu usluge koju proširujete (engl. extend)<sup>20</sup> i anonimnu funkciju (engl. closure) koja bi trebala vratiti modificiranu uslugu. Anonimna funkcija (engl. closure) prima uslugu koja se rješava i instancu kontejnera:

```
$this->app->extend(Service::class, function (Service $service, Application $app) {
    return new DecoratedService($service);
});
```

<sup>20</sup> Ključna riječ `extends` se u PHP koristi za izvođenje klase iz druge klase. To se zove nasljeđivanje (engl. inheritance). Izvedena klasa ima sva javna i zaštićena svojstva klase iz koje je izvedena.

## Rješavanje

### `make` metoda

Možete koristiti `make` metodu za rješavanje instance klase iz kontejnera. `make` metoda prihvaća naziv klase ili interface-a koje želite riješiti:

```
use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);
```

Ako se neke od vaših zavisnosti klase ne mogu riješiti pomoću kontejnera, možete ih injektirati proslijeđujući ih kao [asocijativnu matricu](#) (matrice koje koriste imenovane ključeve koje ste im dodijelili) u `makeWith` metodu. Na primjer, možemo ručno proslijediti `$idargument` konstruktora koji zahtijeva `Transistor` usluga:

```
use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);
```

`Bound` metoda se može koristiti za određivanje je li su klasa ili interface eksplicitno vezani u kontejneru:

```
if ($this->app->bound(Transistor::class)) {
    // ...
}
```

Ako ste izvan pružatelja usluga (engl. service provider) na lokaciji vašeg koda koja nema pristup varijabli `$app`, možete koristiti `App` fasadu (engl. facade) ili `app` pomoćnik (engl. helper) za rješavanje instance klase iz kontejnera:

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);

$transistor = app(Transistor::class);
```

Ako želite da se instanca Laravel kontejnera sama injektira u klasu koju kontejner rješava, možete nagovijestiti tip (engl. type-hint) `Illuminate\Container\Container` klase na svom konstruktoru klase:

```
use Illuminate\Container\Container;

/**
 * Create a new class instance.
 */
```

```
public function __construct(
    protected Container $container
) {}
```

### *Automatsko injektiranje*

Alternativno, i što je važno, možete nagovijestiti tip (engl. type-hint) zavisnost u konstruktoru klase koju je riješio kontejner, uključujući kontrolere, slušatelje događaja (engl. event listeners), middleware, i još mnogo toga. Dodatno, možete nagovijestiti tip (engl. type-hint) zavisnosti u `handle` metodi poslova u čekanju (engl. queued jobs). U praksi, ovo je način na koji bi kontejner trebao riješiti većinu vaših objekata.

Na primjer, možete nagovijestiti tip (engl. type-hint) repozitorija definiran vašom aplikacijom u konstruktoru kontrolera. Repozitorij će se automatski riješiti i injektirati u klasu:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Kreiraj novu instancu kontrolera.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Prikaži korisnika s danim ID-om.
     */
    public function show(string $id): User
    {
        $user = $this->users->findOrFail($id);

        return $user;
    }
}
```

## Metode pozivanja i injektiranja

Ponekad ćete možda poželjeti pozvati metodu na instanci objekta dok dopuštate kontejneru da automatski injektira zavisnosti te metode. Na primjer, s obzirom na sljedeću klasu:

```
<?php

namespace App;

use App\Repositories\UserRepository;

class UserReport
{
    /**
     * Generira novi korisnički izvještaj.
     */
    public function generate(UserRepository $repository): array
    {
        return [
            // ...
        ];
    }
}
```

Možete pozvati `generate` metodu pomoću kontejnera ovako:

```
use App\UserReport;
use Illuminate\Support\Facades\App;

$report = App::call([new UserReport, 'generate']);
```

Metoda `call` prihvaća bilo koji PHP callable<sup>21</sup>. Kontejner `call` metoda može se čak koristiti za pozivanje anonimne funkcije (engl. Closure) dok se automatski injektira njegove ovisnosti:

```
use App\Services\AppleMusic;
use Illuminate\Support\Facades\App;

$result = App::call(function (AppleMusic $apple) {
    // ...
});
```

---

<sup>21</sup> Callable je tip koji predstavlja referencu na funkciju ili metodu koja se može pozvati. Closure se može proslijediti kad god se očekuje callable, ali ne i obrnuto jer closure klasa implementira `__invoke` metodu.

### Kontejnerski događaji

Kontejner usluge pokreće događaj svaki put kada riješi objekt. Ovaj događaj možete poslušati na sljedeći `resolving` način:

```
use App\Services\Transistor;
use Illuminate\Contracts\Foundation\Application;

$this->app->resolving(Transistor::class, function (Transistor $transistor,
Application $app) {
    // Poziva se kada kontejner riješi objekte tipa "Transistor"...
});

$this->app->resolving(function (mixed $object, Application $app) {
    // Poziva se kada kontejner riješi objekte bilo kojeg tipa...
});
```

Kao što možete vidjeti, objekt koji se rješava bit će proslijeđen callback<sup>22</sup> funkciji, što vam omogućava da postavite sva dodatna svojstva na objektu prije nego što se on da svom potrošaču.

### PSR-11

Laravelov servisni kontejner implementira PSR-11 interface. Zato možete nagovijestiti tip (engl. type-hint) PSR-11 interface kontejnera da dobijete instancu kontejnera Laravel:

```
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);

    // ...
});
```

Iznimka se javlja ako se navedeni identifikator ne može riješiti. Iznimka će biti instanca `Psr\Container\NotFoundExceptionInterface` ako identifikator nikada nije bio povezan. Ako je identifikator bio vezan, ali nije mogao riješen, instanca

`Psr\Container\ContainerExceptionInterface` će biti izbačena.

---

<sup>22</sup> Callback funkcija je funkcija koja se proslijeđuje kao argument drugoj funkciji. Bilo koja postojeća funkcija može se koristiti kao funkcija povratnog poziva. Da biste koristili funkciju kao funkciju povratnog poziva, proslijedite string koji sadrži naziv funkcije kao argument druge funkcije.

## Pružatelji usluga

### Uvod

Pružatelji usluga (engl. Service providers) središnje su mjesto za bootstrapping svih Laravel aplikacija. Vaša vlastita aplikacija, kao i sve Laravel-ove temeljne usluge, bootstrap-iraju se pomoću pružatelja usluga.

Ali, što mislimo pod pokreću (engl. „bootstrapped“)? Općenito, mislimo na **registriranje** stvari, uključujući registriranje uvezanog kontejnera usluge (engl. registering service container bindings), slušatelja događaja (engl. event listeners), middleware, pa čak i ruta. Davatelji usluga središnje su mjesto za konfiguriranje vaše aplikacije.

Laravel koristi desetke pružatelja usluga (engl. service provider) interno za pokretanje (engl. Bootstrap) svojih osnovnih usluga, kao što su mailer, red čekanja (engl. queue), keširanje i drugi. Mnogi od ovih pružatelja su „odgođeni“ pružatelji usluga, što znači da se neće učitavati na svaki zahtjev, već samo kada su usluge koje pružaju stvarno potrebne.

Svi korisnički definirani pružatelji usluga registrirani su u datoteci `bootstrap/providers.php`. U dokumentaciji koja slijedi naučit ćete kako napisati vlastite pružatelje usluga i registrirati ih u svojoj Laravel aplikaciji.

#### NAPOMENA:



Ako želite saznati više o tome kako Laravel obrađuje zahtjeve i interno radi, pogledajte dokumentaciju o [životnom ciklusu Laravel zahtjeva](#).

### Pisanje pružatelja usluga

Svi pružatelji usluga (engl. service provider) proširuju `Illuminate\Support\ServiceProvider` klasu. Većina pružatelja usluga sadrži `register` i `boot` metodu. Unutar `register` metode trebali biste samo **uvezati stvari** u kontejner usluge. Nikada ne biste trebali pokušavati registrirati slušatelje događaja (engl. event listeners), rute ili bilo koji drugi dio funkcionalnosti unutar `register` metode.

Artisan CLI može generirati novog pružatelja pomoću `make:provider` naredbe. Laravel će automatski registrirati vaš novi pružatelj usluga u datoteci `bootstrap/providers.php` vaše aplikacije:

```
php artisan make:provider RiakServiceProvider
```

### Metoda registracije

Kao što je ranije spomenuto, unutar `register` metode trebali biste samo vezati stvari u [kontejner usluge](#). Nikada ne biste trebali pokušavati registrirati slušatelje događaja (engl. event listeners), rute ili bilo koji drugi dio funkcionalnosti unutar `register` metode. U suprotnom biste mogli slučajno upotrijebiti uslugu koju pruža davatelj usluge koja još nije učitana.

Pogledajmo osnovnog pružatelja usluga. Unutar bilo koje metode vašeg pružatelja usluga, uvijek imate pristup svojstvu `$app` koje omogućava pristup kontejneru usluge:

```
<?php

namespace App\Providers;
```

```
use App\Services\iak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\ServiceProvider;

class IakServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection(config('iak'));
        });
    }
}
```

Ovaj pružatelj usluga samo definira `register` metodu i koristi tu metodu za definiranje implementacije `App\Services\iak\Connection` u kontejneru usluge. Ako još niste upoznati s Laravelovim kontejnerom usluga, pogledajte [njegovu dokumentaciju](#).

#### *bindings i singletons svojstva*

Ako vaš pružatelj usluga registrira mnogo jednostavnih povezivanja, možda ćete htjeti koristiti svojstva `bindings` i `singletons` umjesto ručnog registriranja svakog povezivanja kontejnera. Kada radna okolina učita pružatelja usluge, on će automatski provjeriti ova svojstva i registrirati njihova povezivanja:

```
<?php

namespace App\Providers;

use App\Contracts\DowntimeNotifier;
use App\Contracts\ServerProvider;
use App\Services\DigitalOceanServerProvider;
use App\Services\PingdomDowntimeNotifier;
use App\Services\ServerToolsProvider;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Sva povezivanja kontejnera koja bi trebala biti registrirana.
     */
}
```



```
* @var array
*/
public $bindings = [
    ServerProvider::class => DigitalOceanServerProvider::class,
];

/**
 * Svi singleton kontejneri koji bi trebali biti registrirani.
 *
 * @var array
 */
public $singletons = [
    DowntimeNotifier::class => PingdomDowntimeNotifier::class,
    ServerProvider::class => ServerToolsProvider::class,
];
}
```

#### *boot metoda*

Dakle, što ako trebamo registrirati pogled composer-a (engl. view composer) unutar našeg pružatelja usluga (engl. Service provider)? To treba učiniti unutar `boot` metode. **Ova se metoda poziva nakon što su registrirani svi ostali pružatelji usluga**, što znači da imate pristup svim ostalim uslugama koje je radna okolina registrirala:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap bilo koje usluge aplikacije.
     */
    public function boot(): void
    {
        View::composer('view', function () {
            // ...
        });
    }
}
```

### Injektiranje zavisnosti `boot` metode

Možete nagovijestiti tip (engl. type-hint) zavisnosti za `boot` metodu vašeg davatelja usluga. [Kontejner usluge](#) automatski će ubaciti sve zavisnosti koje trebate:

```
use Illuminate\Contracts\Routing\ResponseFactory;

/**
 * Bootstrap bilo koje usluge aplikacije.
 */
public function boot(ResponseFactory $response): void
{
    $response->macro('serialized', function (mixed $value) {
        // ...
    });
}
```

### Registracija pružatelja usluga

Svi pružatelji usluga registrirani su u `bootstrap/providers.php` konfiguracijskoj datoteci. Ova datoteka vraća matricu kaoja sadrži nazive klasa svojih pružatelja usluga vaše aplikacije.

```
<?php

return [
    App\Providers\AppServiceProvider::class,
];
```

Kada pozovete naredbu `make:provider Artisan`, Laravel će automatski dodati generiranog pružatelja usluge u datoteku `bootstrap/providers.php`. Međutim, ako ste ručno kreirali klasu pružatelja usluge, trebali biste ručno dodati klasu pružatelja usluge u polje:

```
<?php

return [
    App\Providers\AppServiceProvider::class,
    App\Providers\ComposerServiceProvider::class,
];
```

### Odgodeni pružatelji

Ako vaš pružatelj **samo** registrira povezivanja u [kontejneru usluge](#), možete odlučiti odgoditi njegovu registraciju dok jedno od registriranih povezivanja zaista ne bude potrebno. Odgađanje učitavanja takvog pružatelja poboljšat će performanse vaše aplikacije, jer se ne učitava iz datotečnog sistema na svaki zahtjev.

Laravel sastavlja i pohranjuje listu svih usluga koje pružaju davatelji odgođenih usluga, zajedno s nazivom svoje klase davatelja usluga. Zatim, samo kada pokušate riješiti jednu od ovih usluga, Laravel učitava pružatelja usluge.

Za odgodu učitavanja pružatelja, implementirajte interface

`\Illuminate\Contracts\Support\DeferrableProvider` i definirajte `provides` metodu. `provides` metoda bi trebala vratiti povezivanje kontejnera usluge koje je registrirao pružatelj (engl. Provider):

```
<?php

namespace App\Providers;

use App\Services\Riak\Connection;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Contracts\Support\DeferrableProvider;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider implements DeferrableProvider
{
    /**
     * Registrira bilo koju uslugu aplikacije.
     */
    public function register(): void
    {
        $this->app->singleton(Connection::class, function (Application $app) {
            return new Connection($app['config']['riak']);
        });
    }

    /**
     * Dobij usluge koje pruža pružatelj usluga..
     *
     * @return array<int, string>
     */
    public function provides(): array
    {
        return [Connection::class];
    }
}
```

## Fasade

### Uvod

U cijeloj Laravel dokumentaciji vidjet ćete primjere koda koji je u interakciji s Laravel-ovim svojstvima preko „fasada“. Fasade pružaju „statičan“ interface klasama koje su dostupne u kontejneru usluga aplikacije. Laravel se isporučuje s mnogo fasada koje omogućavaju pristup gotovo svim Laravel-ovim svojstvima.

Laravel fasade služe kao „statički proxy-ji“ temeljnim klasama u kontejneru usluga, pružajući prednost sažete, ekspresivne sintakse uz zadržavanje veće mogućnosti testiranja i fleksibilnosti od tradicionalnih statičkih metoda. Sasvim je u redu ako ne razumijete potpuno kako funkcioniraju fasade - samo se prepustite toku i nastavite učiti o Laravel-u.

Sve Laravelove fasade definirane su u `Illuminate\Support\Facades` prostoru imena (engl. namespace). Dakle, možemo lako pristupiti ovakvoj fasadi:

```
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Route;

Route::get('/cache', function () {
    return Cache::get('key');
});
```

Kroz Laravel dokumentaciju, mnogi primjeri će koristiti fasade za demonstraciju različitih svojstava radne okoline.

### Pomoćne funkcije

Kako bi nadopunio fasade, Laravel nudi niz globalnih „pomoćnih funkcija“ koje dodatno olakšavaju interakciju s uobičajnim Laravel svojstvima. Neke od uobičajenih pomoćnih funkcija s kojima možete komunicirati su `view`, `response`, `url`, `config` i druge. Svaka pomoćna funkcija koju nudi Laravel dokumentirana je s pripadajućom karakteristikom; međutim, potpuna lista dostupna je unutar namjenske pomoćne dokumentacije.

Na primjer, umjesto korištenja `Illuminate\Support\Facades\Response` fasade za generiranje JSON odgovora, možemo jednostavno upotrijebiti `response` funkciju. Budući da su pomoćne funkcije globalno dostupne, ne morate uvoziti nijednu klasu da biste ih koristili:

```
use Illuminate\Support\Facades\Response;

Route::get('/users', function () {
    return Response::json([
        // ...
    ]);
});

Route::get('/users', function () {
    return response()->json([
        // ...
    ]);
});
```

### Kada koristiti fasade

Fasade imaju mnoge prednosti. Oni pružaju sažetu, pamtljivu sintaksu koja vam omogućava korištenje Laravel-ovih svojstava bez pamćenja dugih naziva klasa koje morate ručno umetnuti ili konfigurirati. Nadalje, zbog njihove jedinstvene upotrebe PHP-ovih dinamičkih metoda, lako ih je testirati.

Međutim, kod korištenja fasada treba biti oprezan. Primarna opasnost od fasada je klasa „scope creep“ (puseći doseg). Budući da su fasade tako jednostavne za korištenje i ne zahtijevaju injektiranje, može biti lako pustiti da vaše klase da nastavte rasti i koristiti mnogo fasada u jednoj klasi. Korištenjem injektiranja zavisnosti, ovaj potencijal je ublažen vizualnom povratnom informacijom koju vam veliki konstruktor je daje vaša klasa koja raste prevelika. Dakle, kada koristite fasade, obratite posebnu pažnju na veličinu vaše klase kako bi njegov doseg odgovornosti ostao uzak. Ako vaša klasa postaje prevelika, razmislite o tome da je podijelite u više manjih klasa.

### Fasade nasuprot ubrizgavanja zavisnosti

Objekti koje ćemo ovdje pogledati su:

- Prazni (engl. Dummy) koji se proslijeđuju okolo ali se nikada ne koriste. Obično se koriste samo za popunjavanje liste parametara.
- Lažni (engl. Fake) objekti zapravo imaju radne implementacije, ali obično koriste neke prečice što ih čini neprikladnima za produkciju (baza podataka u memoriji je dobar primjer).
- Zamjene (engl. Stubs) daju standardne odgovore na pozive upućene tokom testa, obično uopće ne odgovaraju ni na što izvan onoga što je programirano za test. Stubovi također mogu bilježiti informacije o pozivima, kao što je email gateway stub koji pamti poruke koje je 'poslao', ili možda samo koliko je poruka 'poslao'. Stubs daju unaprijed definirane odgovore na pozive i usredotočuju se na ishode.
- Imitiranja (engl. Mocks) su ono o čemu ovdje govorimo: objekti unaprijed programirani s očekivanjima koja formiraju specifikaciju poziva za koje se očekuje da će ih primiti. Mocks evidentiraju i validiraju interakcije među stvarnim objektima baze podataka, čime se koncentriraju na ponašanje (engl. behavior).

Jedna od primarnih prednosti ubrizgavanja zavisnosti je mogućnost zamjene implementacija injektirane klase. Ovo je korisno tokom testiranja budući da možete ubaciti mock ili stub i potvrditi da su različite metode pozvane na stub.

Tipično, ne bi bilo moguće da mock ili stub budu stvarno statične metode klase. Međutim, budući da fasade koriste dinamičke metode za proxy pozive metoda objektima razriješenim iz kontejnera usluge, u stvari možemo testirati fasade baš kao što bismo testirali umetnutu instancu klase. Na primjer, s obzirom na sljedeću rutu:

```
- use Illuminate\Support\Facades\Cache;
-
- Route::get('/cache', function () {
-     return Cache::get('key');
- });
```

Testiranje mocks je testiranje ponašanja (engl. behavioral) a testiranje stubs je testiranje stanja. U jednom testu može biti nekoliko stubs ali općenito samo jedan mock.

Koristeći Laravelove metode testiranja fasada, možemo napisati sljedeći test kako bismo potvrdili da je `Cache::get` metoda pozvana s argumentom koji smo očekivali:

```
use Illuminate\Support\Facades\Cache;

/**
 * Primjer osnovnog funkcionalnog testa.
 */
public function test_basic_primjer(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
}
```

#### *Fasade nasuprot Pomoćnih (engl. Helper) funkcija*

Uz fasade, Laravel uključuje razne „pomoćne“ funkcije koje mogu obavljati uobičajene zadatke kao što je generiranja pogleda (engl. views), pokretanja događaja, otpremanja poslova ili slanja HTTP odgovora. Mnoge od ovih pomoćnih funkcija imaju istu funkciju kao odgovarajuća fasada. Na primjer, ovaj poziv fasade i poziv pomagača su ekvivalentni:

```
return Illuminate\Support\Facades\View::make('profile');

return view('profile');
```

Ne postoji praktična razlika između fasada i pomoćnih funkcija. Kada koristite pomoćne funkcije, još uvijek ih možete testirati točno kao što biste testirali odgovarajuću fasadu. Na primjer, s obzirom na sljedeću rutu:

```
Route::get('/cache', function () {
    return cache('key');
});
```

`cache` pomoćnik će pozvati `get` metodu u klasi koja se nalazi ispod `Cache` fasade. Dakle, iako koristimo funkciju pomoćnika, možemo napisati sljedeći test kako bismo potvrdili da je metoda pozvana s argumentom koji smo očekivali:

```
use Illuminate\Support\Facades\Cache;

/**
```

```
* Primjer osnovnog funkcionalnog testa.
*/
public function test_basic_primjer(): void
{
    Cache::shouldReceive('get')
        ->with('key')
        ->andReturn('value');

    $response = $this->get('/cache');

    $response->assertSee('value');
}
```

Kako funkcioniraju fasade

U Laravel aplikaciji, fasada je klasa koja omogućava pristup objektu iz kontejnera. Mašina koja omogućava ovo funkcioniranje je u Facade razredu. Laravel-ove fasade i sve prilagođene fasade koje napravite proširit će osnovnu `Illuminate\Support\Facades\Facade` klasu.

Osnovna `Facade` klasa koristi `__callStatic()` čarobnu metodu za odgodu poziva s vaše fasade na objekt razriješen iz kontejnera. U primjeru ispod, upućen je poziv Laravel keš sistemu. Gledajući ovaj kod, moglo bi se pretpostaviti da se statička `get` metoda poziva u `Cache` klasi:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Cache;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Prikažite profil za određenog korisnika.
     */
    public function showProfile(string $id): View
    {
        $user = Cache::get('user:'.$id);

        return view('profile', ['user' => $user]);
    }
}
```

Primijetite da na vrhu primjera „uvozimo“ `Cache` fasadu. Ova fasada služi kao proxy za pristup temeljnoj implementaciji interfeasa `Illuminate\Contracts\Cache\Factory`. Svi pozivi koje napravimo pomoću fasade bit će proslijeđeni osnovnoj instanci Laravel-ove keš usluge.

Ako pogledamo tu `Illuminate\Support\Facades\Cache` klasu, vidjet ćete da ne postoji statična metoda `get`:

```
class Cache extends Facade
{
    /**
     * Dobijte registrirani naziv komponente.
     */
    protected static function getFacadeAccessor(): string
    {
        return 'cache';
    }
}
```

Umjesto toga, `Cache` fasada proširuje osnovnu `Facade` klasu i definira metodu `getFacadeAccessor()`. Posao ove metode je vraćanje imena povezanog kontejnera usluge. Kada korisnik referencira<sup>23</sup> bilo koju statičku metodu na `Cache` fasadi, Laravel rješava `cache` uvezivanje iz kontejnera usluge i pokreće traženu metodu (u ovom slučaju, `get`) protiv tog objekta.

### Fasade u stvarnom vremenu

Koristeći fasade u stvarnom vremenu, možete tretirati bilo koju klasu u svojoj aplikaciji kao da je fasada. Da bismo ilustrirali kako se to može koristiti, prvo ispitajmo neki kod koji ne koristi fasade u stvarnom vremenu. Na primjer, pretpostavimo da naš `Podcast` model ima `publish` metodu. Međutim, kako bismo objavili podcast, moramo injektirati `Publisher` instancu:

```
<?php
```

---

<sup>23</sup> Referenca je alias koji dopušta da dvije različite varijable pišu istu vrijednost. U PHP-u varijabla objekta ne sadrži sam objekt kao vrijednost nego sadrži samo identifikator objekta koji omogućava dodacima objekta da pronađu stvarni objekt. Kada je objekt poslan argumentom, vraćen ili dodijeljen drugoj varijabli, različite varijable nisu aliasi: one sadrže kopiju identifikatora koji upućuje na isti objekt. Postoji razlika između pointera (pokazivača) i reference.

Pointer pohranjuje memorijsku adresu za pristup objektu. Svaki put kada je objekt dodijeljen, generira se pokazivač. Prosljeđivanje varijable u funkciju vrši se prema zadanim postavkama kao prijenos vrijednosti, tj. radite s kopijom. Zadano prosljeđivanje vrši se prema vrijednosti. Stalno. Međutim, ŠTO se kopira i prosljeđuje je pointer. Kada koristite `"->"`, naravno, pristupat ćete istim internim elementima kao izvorna varijabla u funkciji pozivatelja. Samo korištenje `"="` igrat će se samo s kopijama. `"&"` automatski i trajno postavlja drugo ime/pokazivač na drugo ime varijable/pointer na istu memorijsku adresu kao nešto drugo dok ih ne odvojite. Ovdje je ispravno koristiti izraz "alias". Zamislite to kao spajanje dvaju pokazivača na kuku dok se nasilno ne razdvoje pomoću `„unset()“`. Ova funkcionalnost postoji i u istom dosegu i kada se argument proslijeđi funkciji. Često se proslijeđeni argument naziva „referenca“, zbog određenih razlika između „prijenosa po vrijednosti“ i „prijenosa po referenci“ koje su bile jasnije u C i C++.

Funkcijama se prosljeđuju pointeri na objekte a ne sami objekti. Ovi pointeri su kopije originala, osim ako ne koriste `„&“` u svojoj listi parametara za stvarno prosljeđivanje originala. Originali će se promijeniti samo kada kopate po unutrašnjosti objekta.



```
<php?

namespace App\Models;

use App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Objavi podcast.
     */
    public function publish(Publisher $publisher): void
    {
        $this->update(['publishing' => now()]);

        $publisher->publish($this);
    }
}
```

Injektiranje implementacije izdavača (engl. publisher implementation) u metodu omogućava nam jednostavno testiranje metode u izolaciji jer možemo oponašati (engl. mock) injektiranog izdavača. Međutim, zahtijeva od nas da uvijek proslijedimo instancu izdavača svaki put kada pozovemo `publish` metodu. Koristeći fasade u stvarnom vremenu, možemo održavati istu mogućnost testiranja bez potrebe da eksplicitno prođemo `Publisher` instancu. Da biste generirali fasadu u stvarnom vremenu, dodajte prefiks prostoru imena (engl. namespace) uvezene klase s `Facades`:

```
<?php

namespace App\Models;

use App\Contracts\Publisher;
use Facades\App\Contracts\Publisher;
use Illuminate\Database\Eloquent\Model;

class Podcast extends Model
{
    /**
     * Objavi podcast.
     */
    public function publish(Publisher $publisher): void
    public function publish(): void
    {
        $this->update(['publishing' => now()]);
    }
}
```

```
        $publisher->publish($this);
        Publisher::publish($this);
    }
}
```

Kada se koristi fasada u stvarnom vremenu, implementacija izdavača bit će riješena izvan kontejnera usluge pomoću dijela naziva interface-a ili klase koji se pojavljuje nakon prefiksa [Facades](#). Kada testiramo možemo koristiti ugrađene Laravel-ove pomoćnike za testiranje fasada kako bismo imitirali (engl. mock) ovaj poziv metode:

```
<?php

namespace Tests\Feature;

use App\Models\Podcast;
use Facades\App\Contracts\Publisher;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Tests\TestCase;

class PodcastTest extends TestCase
{
    use RefreshDatabase;

    /**
     * Primjer testa.
     */
    public function test_podcast_can_be_published(): void
    {
        $podcast = Podcast::factory()->create();

        Publisher::shouldReceive('publish')->once()->with($podcast);

        $podcast->publish();
    }
}
```

#### Referenca klase fasade

Ispod ćete pronaći sve fasade i njihove osnovne klasu. Ovo je koristan alat za brzo kopanje po API dokumentaciji za određeni root fasade. Ključ za uvezivanje usluge kontejnera također je uključen gdje je primjenjivo.

Fasada	Klasa	Uvezani kontejner usluge
App	Illuminate\Foundation\Application	app
Artisan	Illuminate\Contracts\Console\Kernel	artisan
Auth	Illuminate\Auth\AuthManager	auth
Auth (Instance)	Illuminate\Contracts\Auth\Guard	auth.driver
Blade	Illuminate\View\Compilers\BladeCompiler	blade.compiler
Broadcast	Illuminate\Contracts\Broadcasting\Factory	
Broadcast (Instance)	Illuminate\Contracts\Broadcasting\Broadcaster	
Bus	Illuminate\Contracts\Bus\Dispatcher	
Cache	Illuminate\Cache\CacheManager	cache
Cache (Instance)	Illuminate\Cache\Repository	cache.store
Config	Illuminate\Config\Repository	config
Cookie	Illuminate\Cookie\CookieJar	cookie
Crypt	Illuminate\Encryption\Encrypter	encrypter
Date	Illuminate\Support\DateFactory	date
DB	Illuminate\Database\DatabaseManager	db
DB (Instance)	Illuminate\Database\Connection	db.connection
Event	Illuminate\Events\Dispatcher	events
File	Illuminate\Filesystem\Filesystem	files
Gate	Illuminate\Contracts\Auth\Access\Gate	
Hash	Illuminate\Contracts\Hashing\Hasher	hash
Http	Illuminate\Http\Client\Factory	
Lang	Illuminate\Translation\Translator	translator
Log	Illuminate\Log\LogManager	log
Mail	Illuminate\Mail\Mailer	mailer
Notification	Illuminate\Notifications\ChannelManager	
Password	Illuminate\Auth\Passwords>PasswordBrokerManager	auth.password
Password (Instance)	Illuminate\Auth\Passwords>PasswordBroker	auth.password.broker
Pipeline (Instance)	Illuminate\Pipeline\Pipeline	
Process	Illuminate\Process\Factory	
Queue	Illuminate\Queue\QueueManager	queue
Queue (Instance)	Illuminate\Contracts\Queue\Queue	queue.connection
Queue (Base Class)	Illuminate\Queue\Queue	
Redirect	Illuminate\Routing\Redirector	redirect
Redis	Illuminate\Redis\RedisManager	redis
Redis (Instance)	Illuminate\Redis\Connections\Connection	redis.connection
Request	Illuminate\Http\Request	request
Response	Illuminate\Contracts\Routing\ResponseFactory	
Response (Instance)	Illuminate\Http\Response	
Route	Illuminate\Routing\Router	router
Schema	Illuminate\Database\Schema\Builder	
Session	Illuminate\Session\SessionManager	session

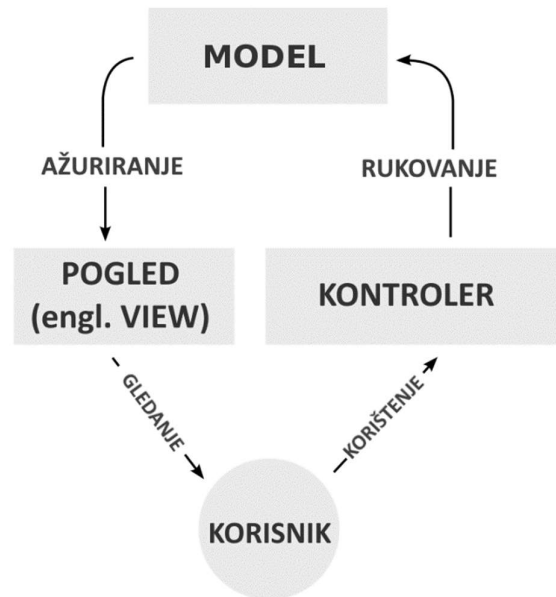
Session (Instance)	Illuminate\Session\Store	session.store
Storage	Illuminate\Filesystem\FilesystemManager	filesystem
Storage (Instance)	Illuminate\Contracts\Filesystem\Filesystem	filesystem.disk
URL	Illuminate\Routing\UrlGenerator	url
Validator	Illuminate\Validation\Factory	validator
Validator (Instance)	Illuminate\Validation\Validator	
View	Illuminate\View\Factory	view
View (Instance)	Illuminate\View\View	
Vite	Illuminate\Foundation\Vite	

## Osnove

### Model-View-Controller (MVC)

Model-View-Controller (MVC) je obrazac dizajna softvera koji se obično koristi za razvoj korisničkih interface-a (veznih sklopova) koja dijeli srodnu programsku logiku na tri međusobno povezana elementa. To se radi da bi se odvojile interne reprezentacije informacija od načina na koji se informacije prezentiraju i prihvaćaju od korisnika. Ova vrsta obrasca koristi se za oblikovanje izgleda stranice.

Tradicionalno korišten za grafička korisničke interface za stolne kompjutere (GUI), ovaj obrazac je postao popularan za dizajniranje web aplikacija. Osim PHP-a i drugi popularni programski jezici kao što su JavaScript, Python, Ruby, PHP, Java, C # i Swift imaju MVC radne okoline koje se koriste za razvoj web ili mobilnih aplikacija direktno iz prozora.



### Komponente

#### Model

Središnja komponenta obrasca. To je dinamička struktura podataka aplikacije, neovisna o korisničkom interfejsu. Direktno upravlja podacima, logikom i pravilima aplikacije. Da bi ste kreirali model, dovoljno je koristiti `php artisan make:model` naredbu. Ova naredba kreira novu model klasu u `app/Models` direktoriju.

```
php artisan make:model User
```

#### Pogled (engl. View)

Svako predstavljanje informacija kao što su grafikoni, dijagrami ili tablice. Moguće je višestruko prikazivanje istih podataka, kao što su trake grafikona za upravljanje i tabelarni prikaza za računovođe.

U Laravel-u, pogled se obično kreira koristeći mehanizam za izradu predložaka kao što je Blade. Blade pruža jednostavnu, ali moćnu sintaksu za stvaranje pogleda.

```
<!DOCTYPE html>
<html>
  <head>
    <title>@yield('title')</title>
  </head>
  <body>
    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

&lt;/html&gt;

### Kontroler

Prihvaća unos i pretvara ga u naredbe za model ili prikaz. Uz podjelu aplikacije na ove komponente, model-pogled-kontroler dizajn definira interkonekcije među njima.

- Model je odgovoran za upravljanje podacima aplikacije. Korisnički unos prima od kontrolera.
- Prikaz znači predstavljanje modela u određenom formatu.
- Kontroler reagira na korisnički unos i obavlja interakcije na objektima modela podataka. Kontroler prima ulaz, opcionalno ga validira i zatim prosljeđuje ulaz modelu.

Kao i kod drugih softverskih obrazaca, MVC izražava „srž rješenja“ problema, dopuštajući mu da se adaptira za svaki sistem.

Za izradu kontrolera u Laravel-u, možete koristiti naredbu `php artisan make:controller`. Ova naredba će generirati novu klasu kontrolera u direktoriju `app/Http/Controllers.php` `artisan make:controller UserController`

### Usluga

Između kontrolera i modela ponekad ide sloj koji se naziva uslugom. Dohvaća podatke iz modela i omogućava kontroleru da koristi preuzete podatke. Ovaj sloj omogućava odvajanje pohrane podataka (model), dohvaćanje podataka (usluga) i rukovanje tj. manipulaciju (kontroler). Budući da ovaj sloj nije dio originalnog MVC koncepta, u većini je slučajeva neobavezan, ali može biti koristan u svrhe upravljanja kodom i ponovne upotrebe u nekim slučajevima.

### Usmjeravanje (engl. Routing)

Najjednostavnije Laravel rute prihvaćaju URI<sup>24</sup> i anonimne funkcije (engl. closure), pružajući vrlo jednostavnu i izražajnu metodu definiranja ruta i ponašanja bez kompliciranih konfiguracijskih datoteka usmjeravanja:

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Jebi se svijete!';
});
```

### Zadane datoteke rute

Sve Laravel rute definirane su u vašim datotekama ruta koje se nalaze u direktoriju ruta. Ove datoteke automatski učitava Laravel pomoću konfiguracije navedene u datoteci `bootstrap/app.php` vaše

---

<sup>24</sup> Uniformni identifikator resursa (engl. Uniform Resource Identifier URI) je jedinstveni niz znakova koji identificira logički ili fizički resurs koji koriste web tehnologije. Glavna razlika između URL-a i URI-ja je u tome što URL navodi lokaciju izvora na internetu, dok se URI može koristiti za identifikaciju bilo koje vrste izvora, ne samo onih na internetu.

aplikacije. Datoteka `routes/web.php` definira rute koje su za vaš web interface. Ovim se rutama dodjeljuje web middleware grupa, koja pruža svojstva kao što su stanje sesije i CSRF zaštita.

Za većinu aplikacija početak ćete definiranjem ruta u datoteci `routes/web.php`. Rutama definiranim u `routes/web.php` može se pristupiti unosom URL-a definirane rute u vaš browser. Na primjer, možete pristupiti sljedećoj ruti odlaskom na `http://example.com/user` u svom browseru:

```
use App\Http\Controllers\UserController;

Route::get('/korisnik', [UserController::class, 'index']);
```

### API rute

Ako će vaša aplikacija također nuditi API bez stanja, možete omogućiti API usmjeravanje pomoću naredbe `install:api` Artisan:

```
php artisan install:api
```

Naredba `install:api` instalira Laravel Sanctum, koji pruža robustan, ali jednostavan zaštitnik provjere autentičnosti API tokena koji se može koristiti za provjeru autentičnosti API potrošača, SPA-ova ili mobilnih aplikacija nezavisnih proizvođača. Osim toga, naredba `install:api` stvara datoteku `routes/api.php`:

```
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:sanctum');
```

Rute u `routes/api.php` su bez statusa i dodijeljene su `api` middleware grupi. Osim toga, `/api` URI prefiks se automatski primjenjuje na ove rute, tako da ga ne morate ručno primijeniti na svaku rutu u datoteci. Prefiks možete promijeniti izmjenom `bootstrap/app.php` datoteke svoje aplikacije:

```
->withRouting(
    api: __DIR__.'/../routes/api.php',
    apiPrefix: 'api/admin',
    // ...
)
```

### Dostupne metode rutera

Router vam omogućava registraciju ruta koje odgovaraju na bilo koji HTTP glagol<sup>25</sup>:

```
Route::get($uri, $callback);
```

---

<sup>25</sup> HTTP definira skup metoda zahtjeva za označavanje željene radnje koju treba izvesti za dani resurs. Iako mogu biti i imenice, ove se metode zahtjeva ponekad nazivaju HTTP glagolima. API programeri obično koriste samo GET, PUT ili POST, ali službeni registar metode HTTP zahtjeva navodi ukupno 39 HTTP glagola, od kojih svaki omogućava metodu za moćne interakcije.

```
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Ponekad ćete možda trebati registrirati rutu koja odgovara na više HTTP glagola. To možete učiniti pomoću `match` metode. Ili, čak možete registrirati rutu koja odgovara na sve HTTP glagole koristeći `any` metodu:

```
Route::match(['get', 'post'], '/', function () {
    // ...
});

Route::any('/', function () {
    // ...
});
```

#### NAPOMENA:



Kada definirate više ruta koje dijele isti URI, rute koje koriste `get`, `post`, `put`, `patch`, `delete` i `options` metode trebaju biti definirane prije ruta koje koriste `any`, `match` i `redirect` metode. Ovo osigurava da se dolazni zahtjev podudara s ispravnom rutom.

#### *Injektiranje zavisnosti*

Možete nagovijestiti tip (engl. type-hint) bilo koje zavisnosti koje zahtijeva vaša ruta u pozivni potpis (engl. callback signature) vaše rute. Deklarirane zavisnosti će se automatski riješiti i injektirati u povratni potpis poziv sa strane Laravel servisnog kontejnera. Na primjer, možete nagovijestiti tip (engl. type-hint) `Illuminate\Http\Request` klase da se trenutni HTTP zahtjev automatski injektira u vaš povratni potpis rute:

```
use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {
    // ...
});
```

#### *CSRF zaštita*

Zapamtite, sve HTML forme koji upućuju na `POST`, `PUT`, `PATCH`, ili `DELETE` rute koje su definirane u `web` datoteci ruta trebaju sadržavati CSRF<sup>26</sup> token polja. U suprotnom, zahtjev će biti odbijen. Više o CSRF zaštiti možete pročitati u CSRF dokumentaciji :

---

<sup>26</sup> Cross-Site Request Forgery (CSRF) je napad koji tjera autentificirane korisnike da pošalju zahtjev web aplikaciji prema kojoj su trenutno autentificirani. CSRF napadi iskorištavaju povjerenje koje web aplikacija ima u autentificiranog korisnika. (Suprotno tome, cross-site scripting (XSS) napadi iskorištavaju povjerenje koje korisnik ima u određenu web aplikaciju). CSRF napad iskorištava ranjivost u web aplikaciji ako ne može razlikovati zahtjev koji je generirao pojedinačni korisnik od zahtjeva koji je generirao korisnik bez njegovog pristanka.



```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

### Redirekcija ruta

Ako definirate rutu koja preusmjerava na drugi URI, možete koristiti `Route::redirect` metodu. Ova metoda pruža prikladnu prečicu tako da ne morate definirati punu rutu ili kontroler za izvođenje jednostavne redirekcije:

```
Route::redirect('/ovdje', '/tamo');
```

Prema podrazumijevanim postavkama `Route::redirect` vraća 302 statusni kod. Statusni kod možete prilagoditi pomoću opsijskog trećeg parametra:

```
Route::redirect('/ovdje', '/tamo', 301);
```

Ili, možete upotrijebiti `Route::permanentRedirect` metodu za vraćanje 301 statusnog koda:

```
Route::permanentRedirect('/ovdje', '/tamo');
```

### UPOZORENJE:



Kada koristite parametre rute u rutama preusmjeravanja, sljedeće parametre rezervira Laravel i ne mogu se koristiti: `destination` i `status`.

### Prikaz ruta

Ako vaša ruta treba vratiti samo view, možete koristiti `Route::view` metodu. Kao i ova `redirect` metoda, ova metoda pruža jednostavnu prečicu tako da ne morate definirati punu rutu ili kontroler. `view` metoda prihvaća URI kao svoj prvi argument i ime pogleda kao svoj drugi argument. Osim toga, možete dati matricu podataka za prosljeđivanje pregledu kao opcionalni treći argument:

```
Route::view('/welcome', 'welcome');
```

```
Route::view('/welcome', 'welcome', ['name' => 'Branko']);
```

### UPOZORENJE:



Kada koristite parametre rute u prikazu ruta, Laravel rezervira sljedeće parametre i oni se ne mogu koristiti: `view`, `data`, `status` i `headers`.

### Izlistajte svoje rute

Naredba `route:list` Artisan može jednostavno dati pregled svih ruta koje definira vaša aplikacija:

```
php artisan route:list
```

Prema podrazumijevanim postavkama, middleware rute koji je dodijeljen svakoj ruti neće biti prikazan u `route:list` izlazu; međutim, možete naložiti Laravelu da prikaže nazive srednjeg softvera rute i grupa middleware dodavanjem opcije `-v` u naredbu:

```
php artisan route:list -v

# Proširi middleware grupe...
php artisan route:list -vv
```

Također možete uputiti Laravel da prikazuje samo rute koje počinju danim URI-jem:

```
php artisan route:list --path=api
```

Osim toga, možete uputiti Laravel da sakrije sve rute koje definiraju paketi drugih proizvođača omogućavanjem opcije `--except-vendor` kada izvršavate `route:list` naredbu:

```
php artisan route:list --except-vendor
```

Isto tako, možete uputiti Laravel da prikazuje samo rute koje definiraju paketi trećih strana pružanjem opcije `--only-vendor` kada izvršavate `route:list` naredbu:

```
php artisan route:list --only-vendor
```

### *Prilagođavanje ruta*

Prema zadanim postavkama, rute vaše aplikacije konfigurirane su i učitane datotekom `bootstrap/app.php`:

```
<?php

use Illuminate\Foundation\Application;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )->create();
```

Međutim, ponekad ćete možda htjeti definirati potpuno novu datoteku koja će sadržavati podskup ruta vaše aplikacije. Da biste to postigli, možete dati anonimne funkcije (engl. closure) `withRouting` metodom. Unutar ove anonimne funkcije (engl. closure) možete registrirati sve dodatne rute koje su potrebne za vašu aplikaciju:

```
use Illuminate\Support\Facades\Route;

->withRouting(
    web: __DIR__.'/../routes/web.php',
    commands: __DIR__.'/../routes/console.php',
    health: 'up',
    then: function () {
        Route::middleware('api')
            ->prefix('webhooks')
            ->name('webhooks.')
            ->group(base_path('routes/webhooks.php'));
    },
)
```

Ili, čak možete preuzeti potpunu kontrolu nad registracijom rute pružanjem `using` anonimne funkcije (engl. closure) metode `withRouting`. Kada se ovaj argument proslijedi, radna okolina neće registrirati HTTP rute i vi ste odgovorni za ručno registriranje svih ruta:

```
use Illuminate\Support\Facades\Route;

->withRouting(
    commands: __DIR__.'/../routes/console.php',
    using: function () {
        Route::middleware('api')
            ->prefix('api')
            ->group(base_path('routes/api.php'));

        Route::middleware('web')
            ->group(base_path('routes/web.php'));
    },
)
```

## Parametri ruta

### *Obavezni parametri*

Ponekad ćete morati uhvatiti segmente URI-ja unutar vaše rute. Na primjer, možda ćete morati uhvatiti ID korisnika iz URL-a. To možete učiniti definiranjem parametara rute:

```
Route::get('/user/{id}', function (string $id) {
    return 'User '.$id;
});
```

Možete definirati koliko god parametara rute zahtijeva vaša ruta:

```
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string $commentId) {  
    // ...  
});
```

Parametri rute uvijek su uokvireni unutar `{}` zagrada i trebaju se sastojati od abecednih znakova. Donje crtice (`_`) također su prihvatljive unutar naziva parametara rute. Parametri rute injektiraju se u povratne rute (engl. route callbacks)/ kontrolere na temelju njihovog redoslijeda - nazivi argumenata povratne rute / kontrolera nisu važni.

#### *Parametri i injektiranje zavisnosti*

Ako vaša ruta ima zavisnosti koje biste željeli da kontejner usluge Laravel automatski injektira u callback (funkcije koje se prosleđuju kao argument drugoj funkciji), trebali biste navesti parametre rute nakon svojih zavisnosti:

```
use Illuminate\Http\Request;  
  
Route::get('/user/{id}', function (Request $request, string $id) {  
    return 'User '.$id;  
});
```

#### *Opcionalni parametri*

Povremeno ćete možda morati navesti parametar rute koji možda neće uvijek biti prisutan u URI-ju. To možete učiniti tako da stavite `?` oznaku iza naziva parametra. Obavezno dajte odgovarajućoj varijabli rute podrazumijevanu (engl. default) vrijednost:

```
Route::get('/user/{name?}', function (?string $name = null) {  
    return $name;  
});  
  
Route::get('/user/{name?}', function (?string $name = 'John') {  
    return $name;  
});
```

#### *Ograničenja regularnog izraza*

Možete ograničiti (engl. constrain) format vaših parametara rute pomoću `where` metode na instanci rute. `where` metoda prihvaća naziv parametra i regularni izraz koji definira kako bi parametar trebao biti ograničen:

```
Route::get('/user/{name}', function (string $name) {  
    // ...  
})->where('name', '[A-Za-z]+');
```

```
Route::get('/user/{id}', function (string $id) {
    // ...
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

Radi praktičnosti, neki često korišteni uzorci uobičajenih izraza imaju pomoćne metode koje vam omogućavaju brzo dodavanje uzorka ograničenja (engl. pattern constraints) vašim rutama:

```
Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function (string $name) {
    // ...
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUuid('id');

Route::get('/user/{id}', function (string $id) {
    // ...
})->whereUlid('id');

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);
```

Ako dolazni zahtjev ne odgovara ograničenjima uzorka rute, vratit će se HTTP odgovor 404.

### *Globalna ograničenja*

Ako želite da parametar rute uvijek bude ograničen (engl. constrained) danim regularnim izrazom, možete upotrijebiti `pattern` metodu. Trebali biste definirati ove uzorke u `boot` metodi svoje `App\Providers\RouteServiceProvider` klase:

```
use Illuminate\Support\Facades\Route;

/**
 * Pokrenite (engl. Bootstrap) bilo koju aplikacijsku uslugu.
```

```
*/  
public function boot(): void  
{  
    Route::pattern('id', '[0-9]+');  
}
```

Nakon što je uzorak definiran, automatski se primjenjuje na sve rute koristeći taj naziv parametra:

```
Route::get('/user/{id}', function (string $id) {  
    // Izvršava se samo ako je {id} numerički...  
});
```

### *Kodiranje kosih crta*

Laravel komponenta rutanja dopušta da svi znakovi osim `/` budu prisutni unutar vrijednosti parametara rute. Morate izričito dopustiti `/` da bude dio vašeg rezerviranog mjesta pomoću `where` regularnog izraza uslova (engl. condition regular expression):

```
Route::get('/search/{search}', function (string $search) {  
    return $search;  
})->where('search', '.*');
```

### **UPOZORENJE:**



Kodirane kose crte podržane su samo unutar posljednjeg segmenta rute.

### Imenovane rute

Imenovane rute omogućuju prikladno generiranje URL-ova ili preusmjeravanja za određene rute. Možete navesti naziv za rutu lančanim povezivanjem `name` metode na definiciju rute:

```
Route::get('/user/profile', function () {  
    // ...  
})->name('profile');
```

Također možete navesti nazive ruta za radnje kontrolera:

```
Route::get(  
    '/user/profile',  
    [UserProfileController::class, 'show']  
)->name('profile');
```

**UPOZORENJE:**

Nazivi ruta uvijek bi trebali biti jedinstveni.

*Generiranje URL-ova za imenovane rute*

Nakon što dodijelite naziv određenoj ruti, možete upotrijebiti naziv rute kada generirate URL-ove ili preusmjeravate pomoću Laravel-ovih `route` i `redirect` pomoćnih funkcija:

```
// Generiranje URL-ova...
$url = route('profile');

// Generiranje redirekcija...
return redirect()->route('profile');

return to_route('profile');
```

Ako imenovana ruta definira parametre, možete proslijediti parametre kao drugi argument u `route` funkciji. Zadani parametri automatski će se umetnuti u generirani URL na svojim ispravnim pozicijama:

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1]);
```

Ako proslijedite dodatne parametre u matrici, ti ključ/vrijednost parovi automatski će se dodati generiranom stringu upita URL-a:

```
Route::get('/user/{id}/profile', function (string $id) {
    // ...
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

**UPOZORENJE:**

Ponekad ćete možda htjeti navesti zadane vrijednosti za nivou cijelog zahtjev za parametre URL-a, kao što je trenutna lokalizacija. Da biste to postigli, možete koristiti `URL::defaults` metodu .

### Pregled trenutne rute

Ako želite utvrditi je li trenutni zahtjev preusmjeren na danu imenovanu rutu, možete koristiti `named` metodu na instanci `Rute`. Na primjer, možete provjeriti trenutni naziv rute iz middleware rute:

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Rukovanje dolaznim zahtjevom.
 *
 * @param  \Closure(\Illuminate\Http\Request):
(\Symfony\Component\HttpFoundation\Response) $next
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

    return $next($request);
}
```

### Grupe ruta

Grupe ruta vam omogućuju dijeljenje atributa rute, kao što je middleware, preko velikog broja ruta bez potrebe za definiranjem tih atributa na svakoj pojedinačnoj ruti.

Ugniježdene grupe pokušavaju inteligentno „spojiti“ attribute sa svojom nadređenom (engl. parent) grupom. Middleware i `where` uslovi se spajaju dok se imena i prefiksi dodaju. Razdjelnici prostora imena (engl. namespace) i kose crte u URI prefiksima automatski se dodaju gdje je to prikladno.

### Middleware

Za dodjelu middleware-a svim rutama unutar grupe, možete koristiti `middleware` metodu prije definiranja grupe. Middleware-i se izvršavaju redoslijedom kojim su navedeni u matrici:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Koristi prvi i drugi middleware...
    });

    Route::get('/user/profile', function () {
        // Koristi prvi i drugi middleware...
    });
});
```



### Kontroleri

Ako grupa ruta koristi isti kontroler, možete koristiti `controller` metodu za definiranje zajedničkog kontrolera za sve rute unutar grupe. Zatim, kada definirate rute, trebate samo osigurati metodu kontrolera koju oni pozivaju:

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

### Rutanje poddomena

Grupe ruta također se mogu koristiti za rukovanje rutanjem poddomena. Poddomenama se mogu dodijeliti parametri rute baš kao i URI rute, što vam omogućava snimanje dijela poddomene za korištenje u vašoj ruti ili kontroleru. Poddomena se može navesti pozivanjem `domain` metode prije definiranja grupe:

```
Route::domain('{account}.primjer.com')->group(function () {
    Route::get('user/{id}', function (string $account, string $id) {
        // ...
    });
});
```

#### UPOZORENJE:



Kako biste bili sigurni da su rute vaše poddomene dostupne, trebali biste registrirati rute poddomene prije registracije ruta početnog direktorija domene. To će spriječiti rute početnog direktorija domene da prebrišu rute poddomena koje imaju istu URI putanju.

### Prefiksi ruta

`prefix` metoda se može koristiti kao prefiks svakoj ruti u grupi sa URI-jem. Na primjer, možda ćete htjeti svim URI-jima ruta unutar grupe dodati prefiks `admin`:

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // Sastavljeno "/admin/users" URL
    });
});
```

### Prefiksi naziva rute

Metoda `name` se može koristiti za postavljanje prefiksa svakom nazivu rute u grupi s danim stringom. Na primjer, možda ćete htjeti nazivima svih ruta u grupi dodati prefiks `admin`. Zadani string ima prefiks

pred nazivom rute točno onako kako je naveden, tako da ćemo biti sigurni da ćemo osigurati praćenje znaka `.` u prefiksu:

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // Ruti pridruženo ime "admin.users"...
    }->name('users');
});
```

### Uvezivanje modela rute

Kada injektirate ID modela u radnju rute ili kontrolera, često ćete postavljati upit bazi podataka kako biste dohvatili model koji odgovara tom ID-u. Laravel povezivanje modela rute pruža prikladan način za automatsko injektiranje instanci modela direktno u vaše rute. Na primjer, umjesto ubacivanja korisničkog ID-a, možete ubaciti cijelu `User` instancu modela koja odgovara danom ID-u.

### *Implicitno uvezivanje*

Laravel automatski rješava Eloquent modele definirane u rutama ili radnjama kontrolera čija nagoviješteni tipovi (engl. type-hint) varijabli odgovaraju nazivu segmenta rute. Na primjer:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

Budući da je `$user` varijabla čiji je tip nagoviješten (engl. type-hint) kao `App\Models\Usermodel` Eloquent model i naziv varijable odgovara `{user}` segmentu URI-ja, Laravel će automatski injektirati instancu modela koja ima ID koji odgovara odgovarajućoj vrijednosti iz URI-ja zahtjeva. Ako odgovarajuća instanca modela nije pronađena u bazi podataka, automatski će se generirati 404 HTTP odgovor.

Naravno, implicitno uvezivanje također je moguće kada se koriste metode kontrolera. Opet, imajte na umu da `{user}` URI segment odgovara `$user` varijabli u kontroleru koji sadrži `App\Models\User` nagoviještaj tipa (engl. type-hint):

```
use App\Http\Controllers\UserController;
use App\Models\User;

// Definicija rute...
Route::get('/users/{user}', [UserController::class, 'show']);

// Definicija metode kontrolera...
public function show(User $user)
{
```

```
return view('user.profile', ['user' => $user]);
}
```

#### *Meko izbrisani modeli*

Tipično, implicitno uvezivanje modela neće dohvatiti modele koji su meko izbrisani. Međutim, možete naložiti implicitnom povezivanju da dohvati ove modele lančanim povezivanjem `withTrashed` metode s definicijom vaše rute:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
})->withTrashed();
```

#### *Prilagodba ključa*

Ponekad ćete možda poželjeti riješiti Eloquent modele koristeći stupac koji nije `id`. Da biste to učinili, možete navesti kolonu u definiciji parametra rute:

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

Ako želite da uvezivanje modela uvijek koristi stupac baze podataka osim `id` kada dohvaća danu klasu modela, možete nadjačati `getRouteKeyName` metodu na Eloquent modelu:

```
/**
 * Nabavi ključ rute za model.
 */
public function getRouteKeyName(): string
{
    return 'slug';
}
```

#### *Prilagođeni ključevi i doseg*

Kada implicitno uvezete više Eloquent modela u jednoj definiciji rute, možda ćete htjeti odrediti doseg (engl. scope) drugog Eloquent modela tako da on mora biti dijete prethodnog Eloquent modela. Na primjer, razmotrite ovu definiciju rute koja dohvaća post na blogu pomoću umetka (engl. slug) za određenog korisnika:

```
use App\Models\Post;
use App\Models\User;
```

```
Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {  
    return $post;  
});
```

Kada koristite implicitno uvezivanje s prilagođenim ključem kao ugniježđenim parametrom rute, Laravel će automatski odrediti doseg upita kako bi dohvatio ugniježđeni model koristeći konvencije da pogodi ime odnosa s roditeljem (engl. relationship name on the parent). U ovom slučaju, pretpostavit će se da `User` model ima odnos s `posts` (množina imena parametra rute) koji se može koristiti za dohvaćanje modela `Post`.

Ako želite, možete naložiti Laravel-u da obrađuje „child“ uvezivanja čak i kada prilagođeni ključ nije dostavljen. Da biste to učinili, možete pozvati metodu `scopeBindings` kada definirate svoju rutu:

```
use App\Models\Post;  
use App\Models\User;  
  
Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {  
    return $post;  
})->scopeBindings();
```

Ili možete uputiti cijelu grupu definicija ruta da koriste uvezivanje dosega (engl. scoped bindings):

```
Route::scopeBindings()->group(function () {  
    Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {  
        return $post;  
    });  
});
```

Slično tome, možete eksplicitno dati instrukcije Laravel-u da ne koristi o uvezivanje dosega (engl. scoped bindings): pozivanjem metode `withoutScopedBindings`:

```
Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {  
    return $post;  
})->withoutScopedBindings();
```

#### *Prilagođavanje ponašanja modela koji nedostaje*

Obično 404 HTTP odgovor će se generirati ako implicitno uvezan model (engl. implicitly bound model) nije pronađen. Međutim, možete prilagoditi ovo ponašanje pozivanjem metode `missing` kada definirate rutu. `missing` metoda prihvaća anonimnu funkciju (engl. closure) koje će se pozvati ako se implicitno uvezan model (engl. implicitly bound model) ne može pronaći:

```
use App\Http\Controllers\LocationsController;  
use Illuminate\Http\Request;
```

```
use Illuminate\Support\Facades\Redirect;

Route::get('/locations/{location:slug}', [LocationsController::class, 'show'])
    ->name('locations.view')
    ->missing(function (Request $request) {
        return Redirect::route('locations.index');
    });
```

#### *Implicitno uvezivanje enuma*

PHP 8.1 je uveo podršku za [Enums](#). Kako bi nadopunio ovo svojstvo, Laravel vam omogućava da na definiciji rute nagovijestite tip (engl. type-hint) string-povratnog Enum-a (engl. backed Enuma) i Laravel će pozvati rutu samo ako taj segment rute odgovara ispravnoj Enum vrijednosti (to mogu biti [int](#) ili [string](#) vrijednosti u istom trenutku). Enum koji sadrži sve povratne slučajeve naziva se „povratni enum“. U suprotnom će automatski biti vraćen 404 HTTP odgovor. Na primjer, s obzirom na sljedeći Enum:

```
<?php

namespace App\Enums;

enum Category: string
{
    case Fruits = 'fruits';
    case People = 'people';
}
```

Možete definirati rutu koja će se pozivati samo ako je [{category}](#) segment rute [fruits](#) ili [people](#). U suprotnom, Laravel će vratiti 404 HTTP odgovor:

```
use App\Enums\Category;
use Illuminate\Support\Facades\Route;

Route::get('/categories/{category}', function (Category $category) {
    return $category->value;
});
```

#### *EksPLICITNO uvezivanje*

Ne morate koristiti Laravelovu implicitnu rezoluciju modela temeljenu na konvenciji (engl. convention based model resolution) kako biste koristili uvezivanje modela. Također možete eksplicitno definirati kako parametri rute odgovaraju modelima. Da biste registrirali eksplicitno uvezivanje, upotrijebite metodu rutera [model](#) za navođenje klase za dati parametar. Trebali biste definirati svoja eksplicitna uvezivanja modela na početku [boot](#) metode svoje [RouteServiceProvider](#) klase:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Pokrenite bilo koju aplikacijsku uslugu..
 */
public function boot(): void
{
    Route::model('user', User::class);

    // ...
}
```

Zatim definirajte rutu koja sadrži `{user}` parametar:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    // ...
});
```

Budući da smo vezali sve `{user}` parametre za `App\Models\User` model, instanca te klase bit će umetnuta u rutu. Tako će, na primjer, zahtjev za `users/1` injektirati `User` instancu iz baze podataka koja ima ID `1`.

Ako odgovarajuća instanca modela nije pronađena u bazi podataka, automatski će se generirati 404 HTTP odgovor.

#### *Prilagođavanje rezolucijske logike*

Ako želite definirati vlastitu logiku rezolucije uvezivanja modela, možete upotrijebiti `Route::bind` metodu. Anonimna funkcija (engl. closure) koju proslijedite `bind` metodi primit će vrijednost URI segmenta i trebalo bi vratiti instancu klase koju treba umetnuti u rutu. Opet, ova prilagodba bi se trebala odvijati u `boot` metodi vaše aplikacije `RouteServiceProvider`:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu..
 */
public function boot(): void
{
    Route::bind('user', function (string $value) {
```

```
        return User::where('name', $value)->firstOrFail();
    });
}
```

Alternativno, možete nadjačati `resolveRouteBinding` metodu na svom Eloquent modelu. Ova metoda će primiti vrijednost URI segmenta i trebala bi vratiti instancu klase koju treba injektirati u rutu:

```
/**
 * Dohvaćanje modela za uvezanu vrijednost.
 *
 * @param mixed $value
 * @param string|null $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveRouteBinding($value, $field = null)
{
    return $this->where('name', $value)->firstOrFail();
}
```

Ako ruta koristi implicitni uvezujući doseg (engl. implicit binding scoping) , `resolveChildRouteBinding` metoda će se koristiti za rješavanje podređenog vezivanja nadređenog (engl. parent) modela:

```
/**
 * Dohvaćanje child modela za uvezanu vrijednost.
 *
 * @param string $childType
 * @param mixed $value
 * @param string|null $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveChildRouteBinding($childType, $value, $field)
{
    return parent::resolveChildRouteBinding($childType, $value, $field);
}
```

### Zamjenske rute

Koristeći `Route::fallback` metodu, možete definirati rutu koja će se izvršiti kada nijedna druga ruta ne odgovara dolaznom zahtjevu. Obično će neobrađeni zahtjevi automatski prikazati stranicu "404" pomoću Upravljačkog sklopa za rukovanje iznimkama (engl. exception handler) vaše aplikacije. Međutim, budući da obično definirate `fallback` rutu unutar vaše `routes/web.php` datoteke, sav

middleware u `web` middleware grupi primijenit će se na rutu. Možete slobodno dodati dodatni middleware ovoj ruti prema potrebi:

```
Route::fallback(function () {
    // ...
});
```

#### UPOZORENJE:



Rezervna ruta bi uvijek trebala biti zadnja ruta koju je registrirala vaša aplikacija.

#### Ograničenje brzine

##### *Definiranje ograničenja brzine*

Laravel uključuje moćne i prilagodljive usluge ograničavanja brzine koje možete koristiti za ograničavanje količine prometa za određenu rutu ili grupu ruta. Da biste započeli, trebali biste definirati konfiguracije limitera brzine koje zadovoljavaju potrebe vaše aplikacije.

Limiteri brzine mogu se definirati unutar `boot` metode vaše aplikacije `App\Providers\AppServiceProvider` klase:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
protected function boot(): void
{
    RateLimiter::for('api', function (Request $request) {
        return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
    });

    // ...
}
```

Limiteri brzine definirani su koristeći `RateLimiter` fasade `for` metode. `for` metoda prihvaća naziv limitera brzine i anonimnu funkciju (engl. closure) koje vraća konfiguraciju ograničenja koja bi se trebala primijeniti na rute koje su dodijeljene limiteru brzine. Konfiguracija ograničenja su instance `Illuminate\Cache\RateLimiting\Limit` klase. Ova klasa sadrži korisne „builder“ metode tako da možete brzo definirati svoje ograničenje. Naziv limitera brzine može biti bilo koji string koji želite:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;
```



```
/**
 * Definirajte svoje modele uvezivanja rute, filtere uzoraka i drugu konfiguraciju
rute.
 */
protected function boot(): void
{
    RateLimiter::for('global', function (Request $request) {
        return Limit::perMinute(1000);
    });

    // ...
}
```

Ako dolazni zahtjev premaši navedeno ograničenje brzine, Laravel će automatski vratiti odgovor s 429 HTTP statusnim kodom. Ako želite definirati vlastiti odgovor koji bi trebao biti vraćen ograničenjem brzine, možete koristiti `response` metodu:

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000)->response(function (Request $request, array
$headers) {
        return response('Custom response...', 429, $headers);
    });
});
```

Budući da callback (funkcija koja se proslijeđuje drugim funkcijama kao argument) limitera brzine primaju instancu dolaznog HTTP zahtjeva, možete izgraditi odgovarajuće ograničenje brzine dinamički na temelju dolaznog zahtjeva ili autentificiranog korisnika:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
        ? Limit::none()
        : Limit::perMinute(100);
});
```

#### *Ograničenja stope segmentiranja*

Ponekad ćete možda željeti segmentirati ograničenja brzine prema nekoj proizvoljnoj vrijednosti. Na primjer, možda želite dopustiti korisnicima pristup određenoj ruti 100 puta u minuti po IP adresi. Da biste to postigli, možete upotrijebiti `by` metodu kada gradite ograničenje brzine:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
        ? Limit::none()
        : Limit::perMinute(100)->by($request->ip());
});
```

```
});
```

Da bismo ilustrirali ovo svojstvo pomoću drugog primjera, možemo ograničiti pristup ruti na 100 puta u minuti po autentificiranom korisničkom ID-ju ili 10 puta u minuti po IP adresi za goste:

```
RateLimiter::for('uploads', function (Request $request) {  
    return $request->user()  
        ? Limit::perMinute(100)->by($request->user()->id)  
        : Limit::perMinute(10)->by($request->ip());  
});
```

### *Višestruka ograničenja brzine*

Ako je potrebno, možete vratiti matricu ograničenja brzine za danu konfiguraciju limitera brzine. Svako ograničenje brzine bit će procijenjeno za rutu na temelju redoslijeda kojim su postavljena unutar matrice:

```
RateLimiter::for('login', function (Request $request) {  
    return [  
        Limit::perMinute(500),  
        Limit::perMinute(3)->by($request->input('email')),  
    ];  
});
```

### *Postavljanje ograničenja brzine na rutama*

Limiteri brzine mogu se pridružiti rutama ili grupama ruta pomoću `throttle` middleware. Regulator (engl. throttle) middleware prihvaća naziv limitera brzine koji želite dodijeliti ruti:

```
Route::middleware(['throttle:uploads'])->group(function () {  
    Route::post('/audio', function () {  
        // ...  
    });  
  
    Route::post('/video', function () {  
        // ...  
    });  
});
```

### *Reguliranje s Redis-om*

Prema zadanim postavkama, prigušivanje middleware-a je mapirano u klasu `Illuminate\Routing\Middleware\ThrottleRequests`. Međutim, ako koristite Redis kao keš driver vaše aplikacije, možda ćete htjeti uputiti Laravel da koristi Redis za upravljanje ograničenjem brzine (engl. rate limiting). Da biste to učinili, trebali biste koristiti metodu `throttleWithRedis` u datoteci

`bootstrap/app.php` svoje aplikacije. Ova metoda mapira `throttle` middleware-a u `Illuminate\Routing\Middleware\ThrottleRequestsWithRedis` u middleware klasu:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->throttleWithRedis();
    // ...
})
```

```
'throttle' => \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
```

#### *Lažiranje metoda forme spoofing metode obrasca*

HTML forme ne podržavaju `PUT`, `PATCH`, ili `DELETE` akcije. Dakle, kada definirate `PUT`, `PATCH`, ili `DELETE` rute koje se pozivaju iz HTML forme, morat ćete dodati skriveno `_method` polje u formu. Vrijednost poslana s `_method` poljem koristit će se kao HTTP metoda zahtjeva:

```
<form action="/primjer" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

Radi praktičnosti, možete koristiti `@method` Blade direktivu<sup>27</sup> za generiranje `_method` polja za unos:

```
<form action="/primjer" method="POST">
    @method('PUT')
    @csrf
</form>
```

#### *Pristup trenutnoj ruti*

Možete koristiti metode `current`, `currentRouteName` i `currentRouteAction` na `Route` fasadi za pristup informacijama o ruti koja obrađuje dolazni zahtjev:

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

Možete pogledati dokumentaciju API-ja za temeljnu klasu fasade rute i instancu rute kako biste pregledali sve metode koje su dostupne na usmjerivaču i klasama rute.

---

<sup>27</sup> PHP direktive su konfiguracijske postavke koje se koriste za kontrolu različitih ponašanja PHP funkcija na vašem site-u. To može uključivati na primjer; ograničenje memorije dodijeljeno PHP-u, maksimalnu količinu vremena prije nego što PHP proces istekne i maksimalnu veličinu datoteke koja se može učitati pomoću PHP skripte. Blade direktive su kodovi prečica za implementaciju osnovne kontrole PHP strukture, kao što su petlje i uslovne izjave.

### *Dijeljenje resursa s različitim izvorima (engl. Cross-Origin Resource Sharing -CORS)*

Laravel može automatski odgovoriti na CORS `OPTIONS` HTTP zahtjeve s vrijednostima koje konfigurirate. Sve CORS postavke mogu se konfigurirati u konfiguracijskoj datoteci `config/cors.php` vaše aplikacije. `OPTIONS` zahtjevima će automatski rukovati middleware `HandleCors` koji je prema podrazumijevanim postavkama uključen u vašu globalnu middleware hrpu.

Ponekad ćete možda trebati prilagoditi konfiguracijske vrijednosti CORS-a za svoju aplikaciju. To možete učiniti objavljivanjem konfiguracijske datoteke `cors` pomoću `config:publish` Artisan naredbe:

```
php artisan config:publish cors
```

Ova naredba smjestit će konfiguracijsku datoteku `cors.php` unutar konfiguracijskog direktorija vaše aplikacije.

#### **NAPOMENA:**



Za više informacija o CORS-u i CORS-ovim zaglavljima, pogledajte [web-dokumentaciju MDN-a o CORS-u](#).

### *Keširanje rute*

Kada implementirate svoju aplikaciju u produkciju, trebali biste iskoristiti prednosti Laravelove keš rute. Korištenje keš rute drastično će smanjiti količinu vremena potrebnog za registraciju svih ruta vaše aplikacije. Za generiranje keš rute, izvršite `route:cache` naredbu Artisan:

```
php artisan route:cache
```

Nakon pokretanja ove naredbe, vaša keširana datoteka ruta bit će učitana na svaki zahtjev. Zapamtite, ako dodate nove rute, morat ćete generirati svježi keš ruta. Zbog toga biste trebali pokrenuti `route:cache` naredbu samo tokom razvoja vašeg projekta.

Možete koristiti `route:clear` naredbu za brisanje keša rute:

```
php artisan route:clear
```

## Middleware

### Uvod

Middleware pruža prikladan mehanizam za pregled i filtriranje HTTP zahtjeva koji ulaze u vašu aplikaciju. Na primjer, Laravel uključuje middleware koji provjerava je li korisnik vaše aplikacije autentificiran. Ako korisnik nije autentificiran, middleware će preusmjeriti korisnika vaše aplikacije na ekran za prijavu. Međutim, ako je korisnik autentificiran, middleware će dopustiti da zahtjev nastavi dalje u aplikaciju.

Dodatni middleware može se napisati za obavljanje raznih zadataka osim provjere autentičnosti. Na primjer, middleware za bilježenje može bilježiti sve dolazne zahtjeve vašoj aplikaciji. Postoji nekoliko middleware-a uključenih u Laravel radnu okolinu, uključujući middleware za autentifikaciju i CSRF zaštitu. Sva ova middleware oprema nalazi se u `app/Http/Middleware` direktoriju.

### Definiranje Middleware-a

Za izradu novog middleware -a upotrijebite `make:middleware` Artisan naredbu:

```
php artisan make:middleware EnsureTokenIsValid
```

Ova naredba će postaviti novu `EnsureTokenIsValid` klasu unutar vašeg `app/Http/Middleware` direktorija. U ovom middleware dopustiti ćemo pristup ruti samo ako dostavljeni `token` unos odgovara određenoj vrijednosti. U suprotnom, preusmjerit ćemo korisnike nazad na `/home` URI:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class EnsureTokenIsValid
{
    /**
     * Rukovanje dolaznim zahtjevom.
     *
     * @param  \Closure(\Illuminate\Http\Request):
    (\Symfony\Component\HttpFoundation\Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }
    }
}
```

```
        return $next($request);
    }
}
```

Kao što možete vidjeti, ako dani `token` ne odgovara našem tajnom tokenu, middleware će vratiti HTTP preusmjerenje klijentu; u suprotnom, zahtjev će biti proslijeđen dalje u aplikaciju. Za proslijeđivanje zahtjeva dublje u aplikaciju (dopuštajući middleware da „prođe“), trebali biste pozvati povratni `$next` poziv s `$request`.

Najbolje je zamisliti middleware kao niz „slojeva“ kroz koje HTTP zahtjevi moraju proći prije nego što dođu do vaše aplikacije. Svaki sloj može ispitati zahtjev i čak ga u potpunosti odbiti.

**NAPOMENA:**

Sav middleware riješen je pomoću kontejnera usluga, tako da možete nagovijestiti tip (engl. type-hint) sve zavisnosti koje su vam potrebne unutar middleware konstruktora.

### *Middleware i odgovori*

Naravno, middleware može obavljati zadatke prije ili nakon proslijeđivanja zahtjeva dublje u aplikaciju. Na primjer, sljedeći middleware izvršio bi neki zadatak prije nego što aplikacija obradi zahtjev:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class BeforeMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        // Izvršavanje akcije

        return $next($request);
    }
}
```

Međutim, ovaj bi middleware izvršio svoju zadatku **nakon** što aplikacija obradi zahtjev:

```
<?php

namespace App\Http\Middleware;
```

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class AfterMiddleware
{
    public function handle(Request $request, Closure $next): Response
    {
        $response = $next($request);

        // Izvršavanje akcije

        return $response;
    }
}
```

#### Registriranje Middleware -a

##### *Globalni Middleware*

Ako želite da se middleware pokreće tokom svakog HTTP zahtjeva vašoj aplikaciji, možete ga dodati globalnoj middleware gomili u datoteci `bootstrap/app.php` vaše aplikacije:

```
use App\Http\Middleware\EnsureTokenIsValid;

->withMiddleware(function (Middleware $middleware) {
    $middleware->append(EnsureTokenIsValid::class);
})
```

`$middleware` objekt koji se daje anonimnu funkciju (engl. closure) `withMiddleware` je instanca `Illuminate\Foundation\Configuration\Middleware` i odgovoran je za upravljanje middleware koji je dodijeljen rutama vaše aplikacije. Metoda dodavanja dodaje middleware na kraj popisa globalne middleware. Ako želite dodati middleware na početak popisa, trebali biste koristiti `prepend` metodu.

##### *Ručno upravljanje Laravelovim zadanim globalnim middleware-om*

Ako želite ručno upravljati Laravelovim globalnom middleware gomilom, metodi korištenja možete dati Laravelovu zadanu globalnu middleware gomilu. Zatim možete prema potrebi prilagoditi zadanu middleware gomilu s `necessary:method`.

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->use([
        // \Illuminate\Http\Middleware\TrustHosts::class,
```

```
\Illuminate\Http\Middleware\TrustProxies::class,  
\Illuminate\Http\Middleware\HandleCors::class,  
  
\Illuminate\Foundation\Http\Middleware\PreventRequestsDuringMaintenance::class,  
\Illuminate\Http\Middleware\ValidatePostSize::class,  
\Illuminate\Foundation\Http\Middleware\TrimStrings::class,  
\Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,  
    ]);  
})
```

### *Pridruživanje Middleware rutama*

Ako želite dodijeliti middleware određenim rutama, možete ga dodati globalnoj middleware hrpi u vašoj aplikacijskom `bootstrap/app.php` datoteci:

```
use App\Http\Middleware\EnsureTokenIsValid;  
  
->withMiddleware(function (Middleware $middleware) {  
    $middleware->append(EnsureTokenIsValid::class);  
})
```

Ruti možete dodijeliti više middleware prosljeđivanjem matrice s middleware nazivima middleware metodi:

```
Route::get('/', function () {  
    // ...  
})->middleware([First::class, Second::class]);
```

### *Isključivanje Middleware*

Kada dodjeljujete middleware grupi ruta, povremeno ćete možda trebati spriječiti primjenu middleware na pojedinačnu rutu unutar grupe. To možete postići pomoću `withoutMiddleware` metode:

```
use App\Http\Middleware\EnsureTokenIsValid;  
  
Route::middleware([EnsureTokenIsValid::class])->group(function () {  
    Route::get('/', function () {  
        // ...  
    });  
  
    Route::get('/profile', function () {  
        // ...  
    })->withoutMiddleware([EnsureTokenIsValid::class]);  
});
```

Također možete isključiti određeni skup middleware iz cijele grupe definicija rute:



```
use App\Http\Middleware\EnsureTokenIsValid;

Route::withoutMiddleware([EnsureTokenIsValid::class])->group(function () {
    Route::get('/profile', function () {
        // ...
    });
});
```

`withoutMiddleware` metoda može ukloniti samo middleware rute i ne primjenjuje se na globalni middleware.

### *Middleware grupe*

Ponekad ćete možda htjeti grupirati nekoliko middleware pod jednim ključem kako biste ih lakše dodijelili rutama. To možete postići koristeći korištenjem metode `appendToGroup` unutar `bootstrap/app.php` datoteke vaše aplikacije.

```
use App\Http\Middleware\First;
use App\Http\Middleware\Second;

->withMiddleware(function (Middleware $middleware) {
    $middleware->appendToGroup('group-name', [
        First::class,
        Second::class,
    ]);

    $middleware->prependToGroup('group-name', [
        First::class,
        Second::class,
    ]);
});
```

Middleware grupe mogu se dodijeliti rutama i radnjama kontrolera koristeći istu sintaksu kao pojedinačni middleware:

```
Route::get('/', function () {
    // ...
})->middleware('web');

Route::middleware(['web'])->group(function () {
    // ...
});
```

### Laravelove zadane middleware grupe

Laravel uključuje unaprijed definirane `web` i `api` middleware grupe koje sadrže zajednički middleware koji biste mogli primijeniti na svoje web i API rute. Zapamtite, Laravel automatski primjenjuje ove middleware grupe na odgovarajuće datoteke `route/web.php` i `routes/api.php`:

<code>web</code> Middleware grupa
<code>Illuminate\Cookie\Middleware\EncryptCookies</code>
<code>Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse</code>
<code>Illuminate\Session\Middleware\StartSession</code>
<code>Illuminate\View\Middleware\ShareErrorsFromSession</code>
<code>Illuminate\Foundation\Http\Middleware\ValidateCsrfToken</code>
<code>Illuminate\Routing\Middleware\SubstituteBindings</code>

<code>api</code> Middleware grupa
<code>Illuminate\Routing\Middleware\SubstituteBindings</code>

Ako biste željeli dodati ili dodati middleware ovim grupama, možete koristiti `web` i `api` metode unutar datoteke `bootstrap/app.php` svoje aplikacije. `web` i `api` metode prikladne su alternative metodi `appendToGroup`:

```
use App\Http\Middleware\EnsureTokenIsValid;
use App\Http\Middleware\EnsureUserIsSubscribed;

->withMiddleware(function (Middleware $middleware) {
    $middleware->web(append: [
        EnsureUserIsSubscribed::class,
    ]);

    $middleware->api(prepend: [
        EnsureTokenIsValid::class,
    ]);
})
```

Možete čak zamijeniti jedan od Laravelovih zadanih unosa grupe middleware vlastitim prilagođenim middleware:

```
use App\Http\Middleware\StartCustomSession;
use Illuminate\Session\Middleware\StartSession;

$middleware->web(replace: [
    StartSession::class => StartCustomSession::class,
]);
```

Ili možete u potpunosti ukloniti middleware:

```
$middleware->web(remove: [
    StartSession::class,
]);
```

### *Ručno upravljanje Laravelovim zadanim middleware grupama*

Ako želite ručno upravljati svim middleware unutar Laravelovih zadanih `web` i `api` middleware grupa, možete u potpunosti redefinirati grupe. Primjer ispod definirat će `web` i `api` middleware grupe s njihovim zadanim middleware-om, omogućujući vam da ih prilagodite prema potrebi:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->group('web', [
        \Illuminate\Cookie\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \Illuminate\Foundation\Http\Middleware\ValidateCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
    ]);

    $middleware->group('api', [
        //
        \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
        // 'throttle:api',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ]);
});
```

### **NAPOMENA:**



Prema zadanim postavkama, grupe `web` i `api` middleware automatski se primjenjuju na odgovarajuće datoteke `route/web.php` i `routes/api.php` vaše aplikacije pomoću datoteke `bootstrap/app.php`.

### *Middleware aliasi*

Možete dodijeliti pseudonime (alias) middleware-u u datoteci `bootstrap/app.php` u svojoj aplikaciji. Pseudonimi middleware-a omogućavaju vam definiranje skraćenog aliasa za danu klasu middleware-a, što može biti posebno korisno za middleware s dugim imenima klasa:

```
use App\Http\Middleware\EnsureUserIsSubscribed;

->withMiddleware(function (Middleware $middleware) {
```

```
$middleware->alias([
    'subscribed' => EnsureUserIsSubscribed::class
]);
})
```

Nakon što je pseudonim middleware-a definiran u datoteci `bootstrap/app.php` vaše aplikacije, možete koristiti alias prilikom dodjele middleware-a rutama:

```
Route::get('/profile', function () {
    // ...
})->middleware('subscribed');
```

Radi praktičnosti, neki od Laravelovih ugrađenih middleware-a prema zadanim su postavkama aliasi. Na primjer, `auth` middleware je alias za `Illuminate\Auth\Middleware\Authenticate middleware`. Ispod je popis zadanih aliasa middleware-a:

Alias	Middleware
<code>auth</code>	<code>Illuminate\Auth\Middleware\Authenticate</code>
<code>auth.basic</code>	<code>Illuminate\Auth\Middleware\AuthenticateWithBasicAuth</code>
<code>auth.session</code>	<code>Illuminate\Session\Middleware\AuthenticateSession</code>
<code>cache.headers</code>	<code>Illuminate\Http\Middleware/SetCacheHeaders</code>
<code>can</code>	<code>Illuminate\Auth\Middleware\Authorize</code>
<code>guest</code>	<code>Illuminate\Auth\Middleware\RedirectIfAuthenticated</code>
<code>password.confirm</code>	<code>Illuminate\Auth\Middleware\RequirePassword</code>
<code>precognitive</code>	<code>Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests</code>
<code>signed</code>	<code>Illuminate\Routing\Middleware\ValidateSignature</code>
<code>subscribed</code>	<code>\Spark\Http\Middleware\VerifyBillableIsSubscribed</code>
<code>throttle</code>	<code>Illuminate\Routing\Middleware\ThrottleRequests</code> ili <code>Illuminate\Routing\Middleware\ThrottleRequestsWithRedis</code>
<code>verified</code>	<code>Illuminate\Auth\Middleware\EnsureEmailIsVerified</code>

### Sortiranje Middleware-a

Rijetko će vam možda trebati vaš middleware da se izvršava određenim redoslijedom, ali da nemate kontrolu nad njihovim redoslijedom kada su dodijeljeni ruti. U ovim situacijama možete navesti svoj prioritet middleware pomoću metode prioriteta u datoteci `bootstrap/app.php` vaše aplikacije:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->priority([
        \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
        \Illuminate\Cookie\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
```

```
\Illuminate\Foundation\Http\Middleware\ValidateCsrfToken::class,  
\Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,  
\Illuminate\Routing\Middleware\ThrottleRequests::class,  
\Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,  
\Illuminate\Routing\Middleware\SubstituteBindings::class,  
\Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,  
\Illuminate\Auth\Middleware\Authorize::class,  
]);  
})
```

### Middleware parametri

Middleware također može primiti dodatne parametre. Na primjer, ako vaša aplikacija treba provjeriti ima li autentificirani korisnik zadanu „ulogu“ prije izvođenja zadane radnje, možete stvoriti `EnsureUserHasRole` middleware koji prima naziv uloge kao dodatni argument.

Dodatni middleware parametri bit će proslijeđeni middleware-u nakon `$next` argumenta:

```
<?php  
  
namespace App\Http\Middleware;  
  
use Closure;  
use Illuminate\Http\Request;  
use Symfony\Component\HttpFoundation\Response;  
  
class EnsureUserHasRole  
{  
    /**  
     * Handle an incoming request.  
     *  
     * @param  \Closure(\Illuminate\Http\Request):  
     * (\Symfony\Component\HttpFoundation\Response)  $next  
     */  
    public function handle(Request $request, Closure $next, string $role): Response  
    {  
        if (! $request->user()->hasRole($role)) {  
            // Redirect...  
        }  
  
        return $next($request);  
    }  
}
```

Middleware parametri mogu se navesti kada definirate rutu odvajanjem naziva middleware i parametara sa `::`:

```
Route::put('/post/{id}', function (string $id) {  
    // ...  
})->middleware('role:editor');
```

Više parametara može biti razdvojeno zarezima:

```
Route::put('/post/{id}', function (string $id) {  
    // ...  
})->middleware('role:editor,publisher');
```

### Ograničeni Middleware

Ponekad middleware možda treba obaviti neki posao nakon što je HTTP odgovor poslan browser-u. Ako definirate `terminate` metodu na vašem middleware-u, a vaš web server koristi FastCGI, `terminate` metoda će se automatski pozvati nakon što se odgovor pošalje browser-u:

```
<?php  
  
namespace Illuminate\Session\Middleware;  
  
use Closure;  
use Illuminate\Http\Request;  
use Symfony\Component\HttpFoundation\Response;  
  
class TerminatingMiddleware  
{  
    /**  
     * Rukovanje dolaznim zahtjevom.  
     *  
     * @param  \Closure(\Illuminate\Http\Request):  
(\Symfony\Component\HttpFoundation\Response)  $next  
     */  
    public function handle(Request $request, Closure $next): Response  
    {  
        return $next($request);  
    }  
  
    /**  
     * Rukovanje zadacima nakon što je odgovor poslan browseru.  
     */  
    public function terminate(Request $request, Response $response): void  
    {
```

```
        // ...  
    }  
}
```

Metoda `terminate` bi trebala primiti i zahtjev i odgovor. Nakon što ste definirali vremenski određen middleware, trebali biste ga dodati na listu ruta ili globalnog middleware u datoteci `bootstrap/app.php` vaše aplikacije.

Kada pozivate `terminate` metodu na vašem middleware-u, Laravel će riješiti novu instancu middleware iz servisnog kontejnera . Ako želite koristiti istu instancu middleware kada se pozivaju metode `handle` i `terminate`, registrirajte middleware sa kontejnerom koristeći `singleton` metodu kontejnera. Obično se to treba učiniti na `register` metodu vašeg `AppServiceProvider`:

```
use App\Http\Middleware\TerminatingMiddleware;  
  
/**  
 * Registrira bilo koju uslugu aplikacije.  
 */  
public function register(): void  
{  
    $this->app->singleton(TerminatingMiddleware::class);  
}
```

## CSRF zaštita

### Uvod

CSRF (eng. Cross-Site Request Forgery) je napad kod kojeg napadač u ime ovlaštenog korisnika pristupa nekom web odredištu. Pritom se iskorištava ranjivost stranice koja ne provjerava izvor HTTP zahtjeva prije nego ga obradi i izvede. Osim greške u web stranici, iskorištava se i web browser korisnika u kojem je zapamćena prethodna prijava korisnika na ranjivu stranicu. CSRF se može izvoditi podmetanjem HTTP zahtjeva u HTML objekte te podmetanjem zloćudnog skriptnog koda. Ovim napadom može se iskorištavati i nepoštivanje pravila o istom izvoru i odredištu zahtjeva u web browseru. U tom slučaju automatski se pokreće slanje HTTP zahtjeva, učitano s jednog web odredišta, drugom web odredištu, bez znanja korisnika. Mogućnosti izvođenja ovog napada su široke, moguće posljedice uspješne zlouporabe još i šire. Svaka aktivnost koja se može pokrenuti preko URL zahtjeva može se pokrenuti i preko CSRF napada. Budući da napad podrazumijeva pristup korisničkom računu legitimnog korisnika, nije tako lako izvesti.

Srećom, Laravel olakšava zaštitu vaše aplikacije od napada krivotvorenjem zahtjeva ukrštenim lokacijama (engl. Cross-Site Request Forgery - CSRF).

### Objašnjenje ranjivosti

U slučaju da niste upoznati s Cross-Site Request Forgery, razgovarajmo o primjeru kako se ova ranjivost može iskoristiti. Zamislite da vaša aplikacija ima `/user/email` rutu koja prihvća `POST` zahtjev za promjenu adrese e-pošte provjerenog korisnika. Najvjerojatnije ova ruta očekuje da `email` polje za unos sadrži adresu e-pošte koju bi korisnik želio početi koristiti.

Bez CSRF zaštite, zlonamjerni website moglo bi kreirati HTML formu koji ukazuje na rutu vaše aplikacije `/user/email` i šalje vlastitu adresu e-pošte zlonamjernog korisnika:

```
<form action="https://your-application.com/user/email" method="POST">
  <input type="email" value="zlonamjerni-email@primjer.com">
</form>

<script>
  document.forms[0].submit();
</script>
```

Ako zlonamjerni website automatski pošalje formu kada se stranica učita, zlonamjerni korisnik samo treba namamiti korisnika vaše aplikacije koji ništa ne sumnja da posjeti njihovu web stranicu i njihova adresa e-pošte bit će promijenjena u vašoj aplikaciji.

Kako bismo spriječili ovu ranjivost, moramo pregledati svaki dolazni `POST`, `PUT`, `PATCH`, ili `DELETE` zahtjev za tajnu vrijednost sesije kojoj zlonamjerna aplikacija ne može pristupiti.

### Spriječavanje CSRF zahtjeva

Laravel automatski generira CSRF „token“ za svaku aktivnu korisničku sesiju kojom upravlja aplikacija. Ovaj se token koristi za provjeru je li autentificirani korisnik osoba koja stvarno šalje zahtjeve aplikaciji. Budući da je ovaj token pohranjen u korisnikovoj sesiji i mijenja se svaki put kada se sesija ponovno generira, zlonamjerna aplikacija ne može mu pristupiti.



CSRF tokenu trenutne sesije može se pristupiti pomoću sesije zahtjeva (engl. request's session) ili pomoću `csrf_token` pomoćne (engl. helper) funkcije:

```
use Illuminate\Http\Request;

Route::get('/token', function (Request $request) {
    $token = $request->session()->token();

    $token = csrf_token();

    // ...
});
```

Bilo kada kada definirate HTML „POST“, „PUT“, „PATCH“ ili „DELETE“ forme u svojoj aplikaciji, trebali biste uključiti skriveno CSRF `_token` polje u formu kako bi CSRF zaštitni middleware mogao potvrditi zahtjev. Zbog praktičnosti, možete koristiti `@csrf` Blade direktivu za generiranje skrivenog tokena polja za unos:

```
<form method="POST" action="/profile">
    @csrf

    <!--Ekvivalentno sa... -->
    <input type="hidden" name="_token" value="{{ csrf_token() }}" />
</form>
```

`App\Http\Middleware\VerifyCsrfToken` middleware, koji je prema podrazumijevanim postavkama uključen u middleware grupu, će automatski provjeriti odgovara li token u zahtjevu za unos tokenu pohranjenom u sesiji. Kada se ta dva tokena podudaraju, znamo da je autentificirani korisnik onaj koji je pokrenuo zahtjev.

### *CSRF tokeni i SPA*

Ako izgradite SPA (Radna okolina aplikacije s jednom stranicom - engl. Single-Page Application Frameworks) koji koristi Laravel kao API backend, trebali biste konzultirati dokumentaciju Laravel Sanctum za informacije o autentifikaciji s vašim API-jem i zaštiti od CSRF ranjivosti.

### *Isključivanje URI-ja iz CSRF zaštite*

Ponekad ćete možda poželjeti isključiti grupu URI-ja iz CSRF zaštite. Na primjer, ako koristite Stripe za obradu plaćanja i koristite njihov webhook sistem, morat ćete isključiti svoju rutu Stripe webhook rukovatelja iz CSRF zaštite jer Stripe neće znati koji CSRF token poslati vašim rutama.

Obično biste ove vrste ruta trebali postaviti van `web` middleware koju Laravel primjenjuje na sve rute u datoteci `routes/web.php`. Međutim, također možete isključiti određene rute tako da navedete njihove URI-je metodi `validateCsrfTokens` u datoteci vaše aplikacije `bootstrap/app.php`:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->validateCsrfTokens(except: [
        'stripe/*',
        'http://example.com/foo/bar',
        'http://example.com/foo/*',
    ]);
});
})
```

**NAPOMENA:**

Radi praktičnosti, CSRF middleware automatski je isključen za sve rute kada pokrećete testiranje.

## X-CSRF-token

Uz provjeru CSRF tokena kao POST parametra, `App\Http\Middleware\VerifyCsrfToken` middleware će također provjeriti `X-CSRF-TOKEN` zaglavlje zahtjeva. Možete, na primjer, pohraniti token u HTML `meta` tag:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Zatim možete uputiti biblioteku kao što je jQuery da automatski doda token svim zaglavljima zahtjeva. Ovo pruža jednostavnu, praktičnu CSRF zaštitu za vaše aplikacije temeljene na AJAX-u koje koriste naslijeđenu JavaScript tehnologiju:

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});
```

## X-XSRF-token

Laravel pohranjuje trenutni CSRF token u šifrirani `XSRF-TOKEN` kolačić koji je uključen uz svaki odgovor koji generira radna okolina. Možete koristiti vrijednost kolačića za postavljanje `X-XSRF-TOKEN` zaglavlja zahtjeva.

Ovaj se kolačić primarno šalje kao pogodnost za razvojne programere jer neke JavaScript radne okoline i biblioteke, kao što su Angular i Axios, automatski postavljaju svoju vrijednost u `X-XSRF-TOKEN` zaglavlje zahtjeva s istim porijeklom.

**NAPOMENA:**

Prema podrazumijevanim postavkama, `resources/js/bootstrap.js` datoteka uključuje Axios HTTP biblioteku koja će automatski poslati `X-XSRF-TOKEN` zaglavlje umjesto vas.

## Kontroleri

### Uvod

Umjesto definiranja cijele vaše logike rukovanja zahtjevima kao anonimnih funkcija (engl. closures) u datotekama rute, možda biste željeli organizirati ovo ponašanje pomoću klasa „kontrolera“. Kontroleri mogu grupirati povezanu logiku rukovanja zahtjevima u jednu klasu. Na primjer, `UserController` klasa može rukovati svim dolaznim zahtjevima koji se odnose na korisnike, uključujući prikazivanje, kreiranje, ažuriranje i brisanje korisnika. Prema podrazumijevanim postavkama, kontroleri su pohranjeni u `app/Http/Controllers` direktoriju.

### Kontroleri zapisivanja

#### Osnovni kontroleri

Za brzo generiranje novog kontrolera, možete pokrenuti `make:controller` Artisan naredbu. Prema podrazumijevanim postavkama, svi kontroleri za vašu aplikaciju pohranjeni su u `app/Http/Controllers` direktoriju:

```
php artisan make:controller UserController
```

Pogledajmo primjer osnovnog kontrolera. Kontroler može imati bilo koji broj javnih metoda koje će odgovoriti na dolazne HTTP zahtjeve:

```
<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Prikažite profil određenog korisnika.
     */
    public function show(string $id): View
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

Nakon što ste napisali klasu i metodu kontrolera, možete definirati rutu do metode kontrolera ovako:

```
use App\Http\Controllers\UserController;
```

```
Route::get('/user/{id}', [UserController::class, 'show']);
```

Kada dolazni zahtjev odgovara navedenom URI-ju rute, `show` metoda u `App\Http\Controllers\UserController` klasi će biti pozvana i parametri rute će biti proslijeđeni metodi.

#### NAPOMENA:



Kontroleri **ne moraju** proširiti (engl. extend) osnovnu klasu. Međutim, nećete imati pristup praktičnim karakteristikama kao što su `middleware` i `authorize` metode koje bi se trebale dijeliti na svim vašim kontrolerima.

#### Kontroleri s jednostrukim djelovanjem

Ako je radnja kontrolera posebno složena, moglo bi vam biti zgodno posvetiti cijelu klasu kontrolera toj jednoj radnji. Da biste to postigli, možete definirati jednu `__invoke` metodu unutar kontrolera:

```
<?php

namespace App\Http\Controllers;

class ProvisionServer extends Controller
{
    /**
     * Osiguravanje novog web servera.
     */
    public function __invoke()
    {
        // ...
    }
}
```

Kada registrirate rute za kontrolere s jednom akcijom, ne morate navesti metodu kontrolera. Umjesto toga, možete jednostavno proslijediti naziv kontrolera ruteru:

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

Možete generirati kontroler koji se može pozvati korištenjem `--invokable` opcije `make:controller` Artisan naredbe:

```
php artisan make:controller ProvisionServer --invokable
```

**NAPOMENA:**

Zamjene (engl. Stubs) kontrolera mogu se prilagoditi pomoću zamjena (engl. stubs) objavljivanja.

## Middleware kontroleri

Middleware se može dodijeliti rutama kontrolera u vašim datotekama rute:

```
Route::get('profile', [UserController::class, 'show'])->middleware('auth');
```

Ili, možda će vam biti zgodno navesti middleware unutar konstruktora vašeg kontrolera. Da biste to učinili, vaš kontroler bi trebao implementirati HasMiddleware interface, koje nalaže da kontroler treba imati statičku middleware metodu. Iz ove metode možete vratiti matricu middleware-a koji bi se trebao primijeniti na radnje kontrolera:

```
class UserController extends Controller
{
    /**
     * Instancirajte novu instancu kontrolera.
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

Kontroleri također omogućavaju registraciju middleware-a pomoću anonimne funkcije (engl. closure). Ovo omogućava prikladan način za definiranje ugrađenog middleware-a za jedan kontroler bez definiranja cijele middleware klase:

```
use Closure;
use Illuminate\Http\Request;

/**
 * Dohvati middleware koji bi trebao biti dodijeljen kontroleru.
 */
public static function middleware(): array
{
    return [
        function (Request $request, Closure $next) {
            return $next($request);
        },
    ],
}
```

```
];
}
```

### Kontroleri resursa

Ako o svakom Eloquent modelu u svojoj aplikaciji razmišljate kao o „resursu“, uobičajeno je izvršiti iste skupove radnji u odnosu na svaki resurs u vašoj aplikaciji. Na primjer, zamislite da vaša aplikacija sadrži `Photo` model i `Movie` model. Vjerojatno je da korisnici mogu kreirati, čitati, ažurirati ili brisati te resurse.

Zbog ovog uobičajenog slučaja upotrebe, Laravel rutanje resursa dodjeljuje uobičajne rute kreiranja, čitanja, ažuriranja i brisanja (engl. create, read, update, and delete – „CRUD“) ka kontroleru s jednom linijom koda. Za početak, možemo upotrijebiti opciju `make:controller` Artisan naredbe `--resource` za brzo kreiranje kontrolera za upravljanje ovim radnjama:

```
php artisan make:controller PhotoController --resource
```

Ova naredba će generirati kontroler na `app/Http/Controllers/PhotoController.php`. Kontroler će sadržavati metodu za svaku od dostupnih operacija resursa. Zatim možete registrirati rutu resursa koja upućuje na kontroler:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

Ova jedna deklaracija rute stvara više ruta za rukovanje različitim radnjama na resursu. Generirani kontroler već će imati zaglavljene metode za svaku od ovih radnji. Zapamtite, uvijek možete dobiti brzi pregled ruta svoje aplikacije pokretanjem `route:list` Artisan naredbe.

Možete čak registrirati mnogo kontrolera resursa odjednom prosljeđivanjem matrice `resources` metodi:

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

### Radnje kojima upravljaju kontroleri resursa

Glagol	URI	Akcija	Naziv rute
GET	<code>/photos</code>	indeks	<code>photos.index</code>
GET	<code>/photos/create</code>	kreiranje	<code>photos.create</code>
POST	<code>/photos</code>	pohranjivanje	<code>photos.store</code>
GET	<code>/photos/{photo}</code>	prikaz	<code>photos.show</code>
GET	<code>/photos/{photo}/edit</code>	editovanje	<code>photos.edit</code>
PUT/PATCH	<code>/photos/{photo}</code>	ažuriranje	<code>photos.update</code>
DELETE	<code>/photos/{photo}</code>	uništavanje	<code>photos.destroy</code>

### Prilagođavanje ponašanja modela koji nedostaje

Obično će se generirati HTTP odgovor 404 ako implicitno uvezani model resursa nije pronađen. Međutim, ovo ponašanje možete prilagoditi pozivanjem `missing` metode kada definirate vaš resurs rute. `missing` metoda prihvata anonimnu funkciju (engl. closure) koje će se pozvati ako se implicitno vezan model ne može pronaći ni za jednu od ruta resursa:

```
use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
    ->missing(function (Request $request) {
        return Redirect::route('photos.index');
    });
```

### Meko izbrisani modeli

Tipično, implicitno uvezivanje modela neće dohvatiti modele koji su meko izbrisani, već će umjesto toga vratiti 404 HTTP odgovor. Međutim, možete uputiti radnu okolinu da dopusti meko izbrisane modele pozivanjem metode `withTrashed` kada definirate vaše resurse rute:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->withTrashed();
```

Pozivanje `withTrashed` bez argumenata omogućit će meko izbrisane modele za `show`, `edit`, i `update` resursa ruta. Možete navesti podskup ovih ruta prosljeđivanjem matrice `withTrashed` metodi:

```
Route::resource('photos', PhotoController::class)->withTrashed(['show']);
```

### Određivanje modela resursa

Ako koristite uvezivanje modela rute (engl. route model binding) i želite da metode kontrolera resursa nagovijestiti tip (engl. type-hint) instance modela, možete upotrijebiti opciju `--model` kada generirate kontroler:

```
php artisan make:controller PhotoController --model=Photo --resource
```

### Generiranje zahtjeva za formu

Možete dati `--requests` opciju kada generirate resurs kontrolera da uputite Artisan da generira klase zahtjeva forme za metode pohranjivanja i ažuriranja kontrolera:

```
php artisan make:controller PhotoController --model=Photo --resource --requests
```

### Djelomične rute resursa

Kada deklarirate rutu resursa, možete navesti podskup radnji koje bi kontroler trebao obrađivati umjesto punog seta zadanih radnji:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

### Rute API resursa

Kada deklarirate rute resursa koje će koristiti API-ji, obično želite izuzeti rute koje predstavljaju HTML predloške (engl. template) kao što su `create` i `edit`. Radi praktičnosti, možete koristiti `apiResource` metodu za automatsko isključivanje ove dvije rute:

```
use App\Http\Controllers\PhotoController;

Route::apiResource('photos', PhotoController::class);
```

Možete registrirati više kontrolera resursa API-ja odjednom prosljeđivanjem matrice `apiResources` metodi:

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\PostController;

Route::apiResources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

Za brzo generiranje kontrolera resursa API-ja koji ne uključuje `create` ili `edit` metode, koristite `--api` prekidač kada izvršavate `make:controller` naredbu:

```
php artisan make:controller PhotoController --api
```

### Ugniježđeni resursi

Ponekad ćete možda trebati definirati rute do ugniježđenog resursa. Na primjer, resurs fotografije može imati više komentara koji mogu biti priloženi fotografiji. Za ugniježđivanje kontrolera resursa, možete koristiti notaciju „točka“ u svojoj deklaraciji rute:



```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class);
```

Ova će ruta registrirati ugniježđeni resurs kojem se može pristupiti s URI-jima kao što je sljedeći:

```
/photos/{photo}/comments/{comment}
```

#### *Određivanje dosega ugniježđenih resursa*

Laravel-ovo svojstvo implicitnog uvezivanja modela može automatski obuhvatiti ugniježđena povezivanja tako da se potvrdi da riješeni podređeni model pripada nadređenom modelu. Koristeći `scoped` metodu kada definirate vaš ugniježđeni resurs, možete omogućiti automatsko određivanje dosega, kao i uputiti Laravel iz kojeg polja treba dohvatiti podređeni resurs. Za više informacija o tome kako to postići, pogledajte dokumentaciju o [određivanju dosega ruta resursa](#).

#### *Plitko gniježđenje (engl. shallow nesting)*

Često nije u potpunosti potrebno imati i nadređeni i podređeni ID unutar URI-ja budući da je podređeni ID već jedinstveni identifikator. Kada koristite jedinstvene identifikatore kao što je automatski inkrementirajući primarni ključ za identifikaciju vaših modela u URI segmentima, možete odabrati korištenje „plitkog gniježđenja“ (engl. shallow nesting):

```
use App\Http\Controllers\CommentController;

Route::resource('photos.comments', CommentController::class)->shallow();
```

Ova definicija rute će definirati sljedeće rute:

Glagol	URI	Akcija	Naziv rute
GET	/photos/{photo}/comments	indeks	photos.comments.index
GET	/photos/{photo}/comments/create	kreiranje	photos.comments.create
POST	/photos/{photo}/comments	Pohranjivanje	photos.comments.store
GET	/comments/{comment}	prikaz	comments.show
GET	/comments/{comment}/edit	editovanje	comments.edit
PUT/PATCH	/comments/{comment}	ažuriranje	comments.update
DELETE	/comments/{comment}	uništavanje	comments.destroy

#### *Imenovanje ruta resursa*

Prema podrazumijevanim postavkama, sve radnje kontrolera resursa imaju naziv rute; međutim, možete nadjačati ova imena prosljeđivanjem matrice `names` sa željenim nazivima ruta:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->names([
    'create' => 'photos.build'
]);
```

### Imenovanje parametara rute resursa

Prema podrazumijevanim postavkama, `Route::resource` kreirat će parametre rute za vaše rute resursa na temelju „singularizirane“ verzije naziva resursa. To možete jednostavno nadjačati na temelju resursa pomoću `parameters` metode. Matrica proslijeđena u `parameters` metodu trebala bi biti asocijativna matrica naziva resursa i naziva parametara:

```
use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([
    'users' => 'admin_user'
]);
```

Primjer iznad generira sljedeći URI za rutu resursa `show`:

```
/users/{admin_user}
```

### Određivanje dosega ruta resursa

Laravelovo svojstvo dosega opsega implicitnog uvezivanja modela može automatski obuhvatiti ugniježđena uvezivanja tako da se potvrdi da riješeni podređeni (engl. child) model pripada nadređenom (engl. parent) modelu. Koristeći `scoped` metodu kada definirate vaš ugniježđeni resurs, možete omogućiti automatsko određivanje dosega, kao i uputiti Laravel koje polje treba dohvatiti podređeni (engl. child) resurs:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class)->scoped([
    'comment' => 'slug',
]);
```

Ova će ruta registrirati ograničeni ugniježđeni resurs kojemu se može pristupiti s URI-jem kao što je ovaj:

```
/photos/{photo}/comments/{comment:slug}
```

Kada koristite implicitno uvezivanje s prilagođenim ključem kao ugniježđeni parametar rute, Laravel će automatski odrediti opseg upita kako bi dohvatio ugniježđeni model od njegovog nadređenog (engl. parent) koristeći konvencije za pogađanje imena odnosa na nadređenom (engl. parent). U ovom slučaju, pretpostavit će se da `Photo` model ima odnos pod imenom `comments` (množina parametra rute name) koji se može koristiti za dohvaćanje `Comment` modela.

### Lokaliziranje URI-ja izvora

Prema podrazumijevanim postavkama, `Route::resource` stvorit će URI resursa koristeći engleske glagole i pravila za množinu. Ako trebate lokalizirati glagole radnje `create` i `edit`, možete upotrijebiti

ovu `Route::resourceVerbs` metodu. To se može učiniti na početku metode `boot` unutar vaše aplikacije `App\Providers\RouteServiceProvider`:

```
/**
 * Definirajte svoje rute modela uvezivanja (route model bindings), filtre uzoraka
 (pattern filters) itd.
 */
public function boot(): void
{
    Route::resourceVerbs([
        'create' => 'crea',
        'edit' => 'edita',
    ]);

    // ...
}
```

Laravelov pluralizator podržava nekoliko različitih jezika koje možete konfigurirati prema svojim potrebama. Nakon što su glagoli i jezik pluralizacije prilagođeni, registracija rute resursa kao što je `Route::resource('publicacion', PublicacionController::class)` proizvest će sljedeće URI-je:

```
/publicacion/crea
/publicacion/{publicaciones}/edita
```

### *Dopunjavanje kontrolera resursa*

Ako trebate dodati dodatne rute kontroleru resursa izvan zadanog skupa ruta resursa, trebali biste definirati te rute prije poziva metode `Route::resource`; inače, rute definirane metodom `resource` mogu nenamjerno imati prednost nad vašim dodatnim rutama:

```
use App\Http\Controller\PhotoController;

Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

### **NAPOMENA:**



Ne zaboravite držati svoje kontrolere fokusirane. Ako ustanovite da vam rutinski trebaju metode van tipičnog skupa radnji resursa, razmislite o dijeljenju kontrolera na dva manja kontrolera.

### *Singleton kontroleri resursa*

Ponekad će vaša aplikacija imati resurse koji mogu imati samo jednu instancu. Na primjer, korisnikov „profil“ može se uređivati ili ažurirati, ali korisnik ne smije imati više od jednog „profila“. Isto tako, slika

može imati jednu „sličicu“. Ti se resursi nazivaju „singleton resursi“, što znači da može postojati jedna i samo jedna instanca resursa. U ovim scenarijima možete registrirati „singleton“ kontroler resursa:

```
use App\Http\Controllers\ProfileController;
use Illuminate\Support\Facades\Route;

Route::singleton('profile', ProfileController::class);
```

Definicija singleton resursa iznad registrirat će sljedeće rute. Kao što vidite, „kreacijske“ rute nisu registrirane za singleton resurse, a registrirane rute ne prihvaćaju identifikator jer može postojati samo jedna instanca resursa:

Glagol	URI	Akcija	Naziv rute
GET	/profile	show	profile.show
GET	/profile/edit	edit	profile.edit
PUT/PATCH	/profile	update	profile.update

Singleton resursi također mogu biti ugniježđeni unutar standardnog resursa:

```
Route::singleton('photos.thumbnail', ThumbnailController::class);
```

U ovom primjeru, `photos` resurs bi primio sve standardne rute resursa ; međutim, `thumbnail` resurs bi bio singleton resurs sa sljedećim rutama:

Glagol	URI	Akcija	Naziv rute
GET	/photos/{photo}/thumbnail	show	photos.thumbnail.show
GET	/photos/{photo}/thumbnail/edit	edit	photos.thumbnail.edit
PUT/PATCH	/photos/{photo}/thumbnail	update	photos.thumbnail.update

### *Singleton resursi koji se mogu kreirati*

Povremeno ćete možda htjeti definirati rute kreiranja i pohranjivanja za pojedinačni resurs. Da biste to postigli, možete pozvati `creatable` metodu kada registrirate singleton rutu resursa:

```
Route::singleton('photos.thumbnail', ThumbnailController::class)->creatable();
```

U ovom primjeru bit će registrirane sljedeće rute. Kao što vidite, `DELETE` ruta će također biti registrirana za kreiranje singleton resurse:

Glagol	URI	Akcija	Naziv rute
GET	/photos/{photo}/thumbnail/create	create	photos.thumbnail.create
POST	/photos/{photo}/thumbnail	store	photos.thumbnail.store
GET	/photos/{photo}/thumbnail	show	photos.thumbnail.show
GET	/photos/{photo}/thumbnail/edit	edit	photos.thumbnail.edit
PUT/PATCH	/photos/{photo}/thumbnail	update	photos.thumbnail.update
DELETE	/photos/{photo}/thumbnail	destroy	photos.thumbnail.destroy

Ako želite da Laravel registira `DELETE` rutu za singleton resurs, ali ne registira rute stvaranja ili pohranjivanja, možete upotrijebiti `destroyable` metodu:

```
Route::singleton(...)->destroyable();
```

#### *API Singleton resursi*

Metoda `apiSingleton` se može koristiti za registraciju singleton resursa kojim će se manipulirati pomoću API-ja, čime `create` i `edit` rute postaju nepotrebne:

```
Route::apiSingleton('profile', ProfileController::class);
```

Naravno, API singleton resursi također mogu biti `creatable`, koji će se registrirati `store` i `destroy` rute za resurs:

```
Route::apiSingleton('photos.thumbnail', ProfileController::class)->creatable();
```

## Uvođenje zavisnosti i kontroleri

### *Injektiranje konstruktora*

Laravel servisni kontejner koristi se za rješavanje svih Laravel kontrolera. Kao rezultat toga, možete nagovijestiti tip (engl. type-hint) bilo koje zavisnosti koje bi vaš kontroler mogao trebati u svom konstruktoru. Deklarirane zavisnosti automatski će se razriješiti i injektirati u instancu kontrolera:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * Kreiranje nove instance kontrolera.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}
}
```

### *Metoda injektiranja*

U dodatku injektiranja konstruktora, možete također nagovijestiti tip (engl. type-hint) zavisnosti u metodama vašeg kontrolera. Uobičajeni slučaj upotrebe za ubacivanje metode je injektiranje `Illuminate\Http\Request` instance u vaše metode kontrolera:

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Pohrani novog korisnika.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->name;

        // Pohrani korisnika...

        return redirect('/users');
    }
}
```

Ako vaša metoda kontrolera također očekuje unos od parametra rute, navedite svoje argumente rute nakon ostalih zavisnosti. Na primjer, ako je vaša ruta definirana ovako:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

I dalje možete nagovijestiti tip (engl. type-hint) `Illuminate\Http\Request` i pristupati svom `id` parametru definiranjem metode kontrolera na sljedeći način:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Ažuriranje danog korisnika.
     */
```

```
public function update(Request $request, string $id): RedirectResponse
{
    // Ažuriranje korisnika...

    return redirect('/users');
}
}
```

## HTTP zahtjevi

### Uvod

Laravelova `Illuminate\Http\Request` klasa pruža objektno orijentiran način interakcije s trenutnim HTTP zahtjevom koji obrađuje vaša aplikacija, kao i dohvaćanje unosa, kolačića i datoteka koje su poslone uz zahtjev.

### Interakcija sa zahtjevom

#### *Pristupanje zahtjevu*

Da biste dobili instancu trenutnog HTTP zahtjeva pomoću ubrizgavanja zavisnosti, trebali biste nagovijestiti tip (engl. type-hint) `Illuminate\Http\Request` klase na svojoj ruti anonimne funkcije (engl. closure) ili metodi kontrolera. Instancu dolaznog zahtjeva automatski će injektirati Laravel servisni kontejner:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Pohrani novog korisnika.
     */
    public function store(Request $request): RedirectResponse
    {
        $name = $request->input('name');

        // Pohrani novog korisnika...

        return redirect('/users');
    }
}
```

Kao što je spomenuto, možete također nagovijestiti tip (engl. type-hint) `Illuminate\Http\Request` klase o anonimnoj funkciji (engl. closure) rute. Kontejner usluge automatski će injektirati dolazni zahtjev u anonimne funkcije (engl. closure) kada se izvrši:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```



### *Injektiranje zavisnosti i parametri rute*

Ako vaša metoda kontrolera također očekuje unos od parametra rute, trebali biste navesti svoje parametre rute nakon ostalih zavisnosti. Na primjer, ako je vaša ruta definirana ovako:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

I dalje možete nagovijestiti tip (engl. type-hint) `Illuminate\Http\Request` i pristupati svom `id` parametru rute definiranjem svoje metode kontrolera na sljedeći način:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Ažuriraj navedenog korisnika.
     */
    public function update(Request $request, string $id): RedirectResponse
    {
        // Ažuriraj navedenog korisnika...

        return redirect('/users');
    }
}
```

### *Putanja zahtjeva, host i metoda*

`Illuminate\Http\Request` instanca pruža niz metoda za ispitivanje dolaznog HTTP zahtjeva i proširuje `Symfony\Component\HttpFoundation\Request` klasu. U nastavku ćemo razmotriti nekoliko najvažnijih metoda.

### *Dohvaćanje putanje zahtjeva*

`path` metoda vraća informacije o putanji zahtjeva. Dakle, ako je dolazni zahtjev usmjeren na `http://primjer.com/foo/bar`, `path` metoda će vratiti `foo/bar`:

```
$uri = $request->path();
```

### Provjera putanje/rute zahtjeva

Metoda `is` vam omogućava da provjerite odgovara li dolazna putanja zahtjeva danom uzorku. Možete koristiti `*` znak kao zamjenski znak (engl. wildcard) kada koristite ovu metodu:

```
if ($request->is('admin/*')) {  
    // ...  
}
```

Koristeći `routeIs` metodu, možete utvrditi odgovara li dolazni zahtjev [imenovanoj ruti](#):

```
if ($request->routeIs('admin.*')) {  
    // ...  
}
```

### Dohvaćanje URL-a zahtjeva

Za dohvaćanje punog URL-a za dolazni zahtjev možete koristiti `url` ili `fullUrl` metode. `url` metoda će vratiti URL bez stringa upita, dok `fullUrl` metoda uključuje string upita:

```
$url = $request->url();  
  
$urlWithQueryString = $request->fullUrl();
```

Ako želite dodati podatke stringa upita trenutnom URL-u, možete pozvati `fullUrlWithQuery` metodu. Ova metoda spaja dane varijable stringa upita s trenutnim stringom upita:

```
$request->fullUrlWithQuery(['type' => 'phone']);
```

Ako želite dobiti trenutni URL bez zadanog parametra stringa upita, možete upotrijebiti `fullUrlWithoutQuery` metodu:

```
$request->fullUrlWithoutQuery(['type']);
```

### Dohvaćanje host-a zahtjeva

Možete dohvatiti „host”<sup>28</sup> dolaznog zahtjeva pomoću `host`, `httpHost` i `schemeAndHttpHost` metoda:

```
$request->host();  
$request->httpHost();  
$request->schemeAndHttpHost();
```

---

<sup>28</sup> Host je kompjuter ili drugi uređaj spojen na kompjutersku mrežu.

### Dohvaćanje metode zahtjeva

`method` metoda će vratiti HTTP glagol za zahtjev. Možete upotrijebiti `isMethod` metodu da provjerite odgovara li HTTP glagol danom stringu:

```
$method = $request->method();

if ($request->isMethod('post')) {
    // ...
}
```

### Zaglavlja zahtjeva

Možete dohvatiti zaglavlje zahtjeva iz `Illuminate\Http\Request` instance pomoću `header` metode. Ako zaglavlje nije prisutno na zahtjevu, bit će vraćeno `null`. Međutim, `header` metoda prihvaća opcionalni drugi argument koji će biti vraćen ako zaglavlje nije prisutno na zahtjevu:

```
$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');
```

Metoda `hasHeader` se može koristiti za utvrđivanje sadrži li zahtjev određeno zaglavlje:

```
if ($request->hasHeader('X-Header-Name')) {
    // ...
}
```

Radi praktičnosti, `bearerToken` metoda se može koristiti za dohvaćanje tokena nositelja (engl. bearer) iz `Authorization` zaglavlja. Ako takvo zaglavlje ne postoji, vratit će se prazan string:

```
$token = $request->bearerToken();
```

### IP adresa zahtjeva

`ip` metoda se može koristiti za dohvaćanje IP adrese klijenta koji je uputio zahtjev vašoj aplikaciji:

```
$ipAddress = $request->ip();
```

Ako želite dohvatiti matricu IP adresa, uključujući sve IP adrese klijenata koje su prosljedili proxy serveri<sup>29</sup>, možete koristiti ovu `ips` metodu. „Originalna“ IP adresa klijenta bit će na kraju matrice:

```
$ipAddresses = $request->ips();
```

---

<sup>29</sup> Proxy server je sistem ili router koji pruža povezivanje (engl. gateway) među korisnika i interneta. Zbog toga pomaže spriječiti kibernetičke napadače da uđu u privatnu mrežu. To je server koji se naziva „posrednik“ (engl. intermediary) jer ide između krajnjih korisnika i web stranica koje posjećuju na mreži.

Općenito, IP adrese treba smatrati nepouzdanim, korisnički kontroliranim unosom i koristiti ih samo u informativne svrhe.

### *Pregovaranje o sadržaju*

Laravel nudi nekoliko metoda za provjeru traženog tipa sadržaja dolaznog zahtjeva pomoću `Accept` zaglavlja. Prvo, `getAcceptableContentTypes` metoda će vratiti matricu koji sadrži sve tipove sadržaja koje prihvaća zahtjev:

```
$contentTypes = $request->getAcceptableContentTypes();
```

`accepts` metoda prihvaća matricu vrsta sadržaja i vraća `true` ako zahtjev prihvaća bilo koju vrstu sadržaja. U suprotnom bit će vraćen `false`:

```
if ($request->accepts(['text/html', 'application/json'])) {  
    // ...  
}
```

Možete koristiti `prefers` metodu da odredite koja vrsta sadržaja iz dane matrice vrsta sadržaja je najpoželjnija prema zahtjevu. Ako zahtjev ne prihvati niti jedan od ponuđenih vrsta sadržaja, vratit će se `null`:

```
$preferred = $request->prefers(['text/html', 'application/json']);
```

Budući da mnoge aplikacije poslužuju samo HTML ili JSON, možete koristiti `expectsJson` metodu za brzo određivanje očekuje li dolazni zahtjev JSON odgovor:

```
if ($request->expectsJson()) {  
    // ...  
}
```

### *PSR-7 zahtjevi*

PSR-7 standard specificira interfejs za HTTP poruke, uključujući zahtjeve i odgovore. Ako želite dobiti instancu PSR-7 zahtjeva umjesto Laravel zahtjeva, prvo morate instalirati nekoliko biblioteka. Laravel koristi komponentu *Symfony HTTP Message Bridge* za pretvaranje karakterističnih Laravel zahtjeva i odgovora u PSR-7 kompatibilne implementacije:

```
composer require symfony/psr-http-message-bridge  
composer require nyholm/psr7
```

Nakon što ste instalirali ove biblioteke, možete dobiti PSR-7 zahtjev nagovijestiti tip (engl. type-hint) interfejsa zahtjeva na vašoj anonimnoj funkciji (engl. closure) rute ili metodi kontrolera:

```
use Psr\Http\Message\ServerRequestInterface;
```

```
Route::get('/', function (ServerRequestInterface $request) {  
    // ...  
});
```

**NAPOMENA:**

Ako vratite instancu PSR-7 odgovora iz rute ili kontrolera, ona će se automatski pretvoriti natrag u instancu Laravel odgovora i prikazati u radnoj okolini.

## Unos

### *Dohvaćanje unosa*

### *Dohvaćanje svih ulaznih podataka*

Možete dohvatiti sve ulazne podatke dolaznog zahtjeva `array` koristeći `all` metodu. Ova se metoda može koristiti bez obzira je li dolazni zahtjev iz HTML forme ili je XHR zahtjev<sup>30</sup>:

```
$input = $request->all();
```

Koristeći `collect` metodu, možete dohvatiti sve ulazne podatke dolaznog zahtjeva kao kolekciju:

```
$input = $request->collect();
```

Metoda `collect` vam također omogućava dohvaćanje podskupa unosa dolaznog zahtjeva kao zbirke:

```
$request->collect('users')->each(function (string $user) {  
    // ...  
});
```

### *Dohvaćanje ulazne vrijednosti*

Koristeći nekoliko jednostavnih metoda, možete pristupiti svim korisničkim unosima iz vaše `Illuminate\Http\Request` instance bez brige o tome koji je HTTP glagol korišten za zahtjev. Bez obzira na HTTP glagol, `input` metoda se može koristiti za dohvaćanje korisničkog unosa:

```
$name = $request->input('name');
```

Možete proslijediti podrazumijevanu vrijednost kao drugi argument `input` metode. Ova vrijednost će biti vraćena ako tražena ulazna vrijednost nije prisutna na zahtjevu:

---

<sup>30</sup> XMLHttpRequest ili XMR je JavaScript klasa koja sadrži metode za asinkroni prijenos HTTP zahtjeva iz web browsera na web server. Metode omogućuju aplikaciji temeljenoj na browseru da napravi precizni poziv serveru i pohrani rezultate u XMLHttpRequest atribut `responseText`.

```
$name = $request->input('name', 'Branko');
```

Kada radite s formama koji sadrže ulaze matrice, koristite notaciju „točka“ za pristup matricama:

```
$name = $request->input('products.0.name');  
  
$names = $request->input('products.*.name');
```

Možete pozvati `input` metodu bez ikakvih argumenata kako biste dohvatili sve ulazne vrijednosti kao asocijativnu matricu:

```
$input = $request->input();
```

#### *Dohvaćanje unosa iz stringa upita*

Dok `input` metoda dohvaća vrijednosti iz cjelokupnog sadržaja zahtjeva (uključujući string upita), `query` metoda će dohvaćati vrijednosti samo iz stringa upita:

```
$name = $request->query('name');
```

Ako traženi podaci o vrijednosti stringa upita nisu prisutni, drugi argument ove metode bit će vraćen:

```
$name = $request->query('name', 'Mladen');
```

Možete pozvati `query` metodu bez ikakvih argumenata kako biste dohvatili sve vrijednosti stringa upita kao asocijativna matrica:

```
$query = $request->query();
```

#### *Dohvaćanje JSON ulaznih vrijednosti*

Kada šaljete JSON zahtjeve vašoj aplikaciji, možete pristupiti JSON podacima pomoću `input` metode sve dok je `Content-Type` zaglavlje zahtjeva ispravno postavljeno na `application/json`. Možete čak koristiti sintaksu „točka“ za dohvaćanje vrijednosti koje su ugniježdene unutar JSON matrica/objekata:

```
$name = $request->input('user.name');
```

#### *Dohvaćanje stringable ulaznih vrijednosti*

Stringable interface označava da klasa ima metodu `__toString()`. Za razliku od većine interface-a, Stringable je implicitno prisutan u svakoj klasi koja ima definiranu čarobnu metodu `__toString()`, iako se može i treba eksplicitno deklarirati.

Njegova primarna vrijednost je dopustiti funkcijama provjeru tipa u odnosu na union tip<sup>31</sup> `string|Stringable` za prihvatanje ili jednostavnog stringa ili objekta koji se može pretvoriti tip (engl. cast)<sup>32</sup> u string.

Umjesto dohvaćanja ulaznih podataka zahtjeva kao jednostavan `string`, možete koristiti `string` metodu za dohvaćanje podataka zahtjeva kao instance `Illuminate\Support\Stringable`:

```
$name = $request->string('name')->trim();
```

#### *Dohvaćanje Boolean-ovih ulaznih vrijednosti*

Kada radite s HTML elementima kao što su checkbox, vaša aplikacija može primiti „istine“ vrijednosti koje su zapravo stringovi. Na primjer, „true“ ili „on“. Radi praktičnosti, možete koristiti `boolean` metodu za dohvaćanje ovih vrijednosti kao Boolean. `boolean` metoda vraća `true` za 1, „1“, true, „true“, „on“ i „yes“. Sve ostale vrijednosti će se vratiti `false`:

```
$archived = $request->boolean('archived');
```

#### *Dohvaćanje datumskih ulaznih vrijednosti*

Radi praktičnosti, ulazne vrijednosti koje sadrže datume/vremena mogu se dohvatiti kao Carbon instance pomoću `date` metode. Ako zahtjev ne sadrži ulaznu vrijednost s danim imenom, vratit će se `null`:

```
$birthday = $request->date('birthday');
```

Drugi i treći argument koje prihvaća `date` metoda mogu se koristiti za određivanje formata datuma odnosno vremenske zone:

```
$elapsed = $request->date('elapsed', '!H:i', 'Europe/Zagreb');
```

Ako je ulazna vrijednost postoji, ali ima nevažeći format, `InvalidArgumentException` bit će izbačeno (engl. thrown); zato se preporučuje da potvrdite unos prije pozivanja `date` metode.

#### *Dohvaćanje Enum ulaznih vrijednosti*

Ulazne vrijednosti koje odgovaraju [PHP enum-ima](#)<sup>33</sup> također se mogu dohvatiti iz zahtjeva. Ako zahtjev ne sadrži ulaznu vrijednost s danim imenom ili enum nema pozadinsku vrijednost koja odgovara

<sup>31</sup> Od PHP 8.0 sada možete deklarirati bilo koji broj proizvoljnih tipova za svojstva, argumente i povratne tipove.

<sup>32</sup> PHP ne zahtijeva eksplicitnu definiciju tipa u deklaraciji varijabli. U ovom slučaju, tip varijable određen je vrijednošću koju pohranjuje. Drugim riječima, ako je niz dodijeljen varijabli `$var`, onda je `$var` tipa string. Ako se nakon toga cjelobrojna vrijednost dodijeli `$var`, on će biti tipa int. Cast se vadi tako da se ispred vrijednosti koju pretvaramo u zagradi napiše u čega pretvaramo (int), (bool), (float), (string), (array), (object), (unset).

<sup>33</sup> Enumeracije ili „Enums“ omogućavaju da definirate prilagođeni tip na moguće vrijednosti. U PHP-u to su posebna vrsta objekata. Enum je po sebi klasa a njegovi mogući slučajevi su svi objekti jedne instance te klase. Najpopularniji primjer enumeracije je ugrađeni Boolean tip

ulaznoj vrijednosti, nullbit će vraćen. Metoda `enum` prihvaća naziv ulazne vrijednosti i enum klasu kao prvi i drugi argument:

```
use App\Enums\Status;

$status = $request->enum('status', Status::class);
```

#### *Dohvaćanje unosa pomoću dinamičkih svojstava*

Također možete pristupiti korisničkom unosu pomoću dinamičkih svojstava na instanci `Illuminate\Http\Request`. Na primjer, ako jedan od formi vaše aplikacije sadrži polje `name`, možete pristupiti vrijednosti polja ovako:

```
$name = $request->name;
```

Kada koristite dinamička svojstva, Laravel će prvo potražiti vrijednost parametra u sadržaju zahtjeva. Ako nije prisutna, Laravel će tražiti polje u odgovarajućim parametrima rute.

#### *Dohvaćanje dijela ulaznih podataka*

Ako trebate dohvatiti podskup ulaznih podataka, možete koristiti `only` i `except` metode. Obje ove metode prihvaćaju jedan `array` ili dinamičku listu argumenata:

```
$input = $request->only(['username', 'password']);

$input = $request->only('username', 'password');

$input = $request->except(['credit_card']);

$input = $request->except('credit_card');
```

#### **UPOZORENJE:**



`only` metoda vraća sve parove ključ/vrijednost koje tražite; međutim, neće vratiti parove ključ/vrijednost koji nisu prisutni u zahtjevu.

#### *Prisutnost unosa*

Možete koristiti `has` metodu da odredite je li vrijednost prisutna na zahtjevu. `has` metoda vraća `true` ako je vrijednost prisutna na zahtjevu:

```
if ($request->has('name')) {
    // ...
}
```

Kada se dobije matrica, `has` metoda će odrediti jesu li prisutne sve navedene vrijednosti:



```
if ($request->has(['name', 'email'])) {  
    // ...  
}
```

`hasAny` metoda vraća `true` ako je prisutna bilo koja od navedenih vrijednosti:

```
if ($request->has(['name', 'email'])) {  
    // ...  
}
```

`whenHas` metoda će izvršiti zadanu anonimnu funkciju (engl. closure) ako je vrijednost prisutna na zahtjevu:

```
$request->whenHas('name', function (string $input) {  
    // ...  
});
```

Druga anonimna funkcija (engl. closure) može se proslijediti `whenHas` metodi koja će se izvršiti ako navedena vrijednost nije prisutna na zahtjevu:

```
$request->whenHas('name', function (string $input) {  
    // "name" vrijednost je prisutna...  
}, function () {  
    // "name" vrijednost nije prisutna..  
});
```

Ako želite utvrditi je li vrijednost prisutna na zahtjevu i nije prazan string, možete koristiti `filled` metodu:

```
if ($request->filled('name')) {  
    // ...  
}
```

`anyFilled` metoda vraća `true` ako bilo koja od navedenih vrijednosti nije prazan string:

```
if ($request->anyFilled(['name', 'email'])) {  
    // ...  
}
```

`whenFilled` metoda će izvršiti zadanu anonimnu funkciju (engl. closure) ako je vrijednost prisutna na zahtjevu i nije prazan string:

```
$request->whenFilled('name', function (string $input) {
```

```
// ...  
});
```

Druga anonimna funkcija (engl. closure) može se proslijediti `whenFilled` metodi koja će se izvršiti ako navedena vrijednost nije „popunjena“:

```
$request->whenFilled('name', function (string $input) {  
    // "name" vrijednost je popunjena...  
}, function () {  
    // "name" vrijednost nije popunjena...  
});
```

Da biste utvrdili nedostaje li određeni ključ u zahtjevu, možete koristiti `missing` i `whenMissing` metode:

```
if ($request->missing('name')) {  
    // ...  
}  
  
$request->whenMissing('name', function (array $input) {  
    // "name" vrijednost nedostaje...  
}, function () {  
    // "name" vrijednost je prisutna...  
});
```

### *Spajanje dodatnog unosa*

Ponekad ćete možda trebati ručno spojiti dodatni unos u postojeće ulazne podatke zahtjeva. Da biste to postigli, možete koristiti `merge` metodu. Ako određeni ključ za unos već postoji na zahtjevu, bit će prebrisan podacima koji su dostavljeni `merge` metodi:

```
$request->merge(['votes' => 0]);
```

`mergeIfMissing` metoda se može koristiti za spajanje (kombiniranje) unosa u zahtjev ako odgovarajući ključevi već ne postoje unutar ulaznih podataka zahtjeva:

```
$request->mergeIfMissing(['votes' => 0]);
```

### *Stari unos*

Laravel vam omogućava da zadržite unos iz jednog zahtjeva tokom sljedećeg zahtjeva. Ova je karakteristika posebno korisna za ponovno popunjavanje formi nakon otkrivanja grešaka u provjere ispravnosti. Međutim, ako koristite Laravel-ova uključena svojstva provjere ispravnosti (validnosti),

moguće je da nećete morati ručno koristiti ove blic (engl. flash) metode unosa sesije direktno, budući da će ih neki od Laravel-ovih ugrađenih svojstava provjere ispravnosti pozvati automatski.

#### *Blic unos u sesiju*

`flash` metoda na `Illuminate\Http\Request` klasi će blicnuti ono što trenutno unosite u sesiju tako da bude dostupno tokom sljedećeg zahtjeva korisnika prema aplikaciji:

```
$request->flash();
```

Također možete koristiti metode `flashOnly` i `flashExcept` za blicanje podskupa podataka zahtjeva za sesiju. Ove su metode korisne za držanje osjetljivih informacija kao što su šifre izvan sesije:

```
$request->flashOnly(['username', 'email']);
```

```
$request->flashExcept('password');
```

#### *Blicani unos pa preusmjeravanje*

Budući da ćete često htjeti blicnuti unos u sesiju i zatim ga preusmjeravati na prethodnu stranicu, možete lako ulančati blicanje unosa na preusmjeravanje pomoću `withInput` metode:

```
return redirect('/form')->withInput();

return redirect()->route('user.create')->withInput();

return redirect('/form')->withInput(
    $request->except('password')
);
```

#### *Dohvaćanje starog unosa*

Da biste dohvatili blicani unos iz prethodnog zahtjeva, pozovite `old` metodu na instanci `Illuminate\Http\Request`. `old` metoda će povući prethodno blicane ulazne podatke iz sesije:

```
$username = $request->old('username');
```

Laravel također nudi globalnog `old` pomoćnika (engl. helper). Ako prikazujete stari unos unutar Blade predloška (engl. template), praktičnije je koristiti `old` pomoćnik da bi ste ponovo popunili formu. Ako za dano polje ne postoji stari unos, vratit će se `null`:

```
<input type="text" name="username" value="{{ old('username') }}">
```

## Kolačići

### Dohvaćanje kolačića iz zahtjeva

Svi kolačići (engl. cookies) koje kreira Laravel radna okolina šifrirani su i potpisani kodom za provjeru autentičnosti, što znači da će se smatrati nevažećima ako ih je promijenio klijent. Da biste dohvatili vrijednost kolačića iz zahtjeva, upotrijebite `cookie` metodu na `Illuminate\Http\Request` instanci:

```
$value = $request->cookie('name');
```

### Obrezivanje i normalizacija unosa

Prema podrazumijevanim postavkama, Laravel uključuje `App\Http\Middleware\TrimStrings` i `Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull` middleware u globalni middleware hrpu vaše aplikacije. Ovaj middleware je naveden u globalnoj hrpi middleware `App\Http\Kernel` klase. Ovaj middleware će automatski obrezati sva dolazna string polja na zahtjev, kao i pretvoriti sva string prazna polja u `null`. To vam omogućava da ne brinete o problemima normalizacije u vašim rutama i kontrolerima.

### Onemogućivanje normalizacije unosa

Ako želite onemogućiti ovo ponašanje za sve zahtjeve, možete ukloniti dva middleware-a sa middleware stack-a vaše aplikacije pozivanjem metode `$middleware->remove` u datoteci `bootstrap/app.php` vaše aplikacije:

```
use Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull;
use Illuminate\Foundation\Http\Middleware\TrimStrings;

->withMiddleware(function (Middleware $middleware) {
    $middleware->remove([
        ConvertEmptyStringsToNull::class,
        TrimStrings::class,
    ]);
});
```

Ako želite onemogućiti obrezivanje stringa i konverziju praznog stringa za podskup zahtjeva vašoj aplikaciji, možete upotrijebiti `trimStrings` i `convertEmptyStringsToNull` metode srednjeg softvera unutar datoteke `bootstrap/app.php` vaše aplikacije. Obje metode prihvataju matricu anonimnih funkcija (closures), koji bi trebali vratiti `true` ili `false` da naznače treba li se normalizacija unosa preskočiti:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->convertEmptyStringsToNull(except: [
        fn (Request $request) => $request->is('admin/*'),
    ]);

    $middleware->trimStrings(except: [
        fn (Request $request) => $request->is('admin/*'),
    ]);
});
```

```
});  
})
```

## Datoteke

### *Dohvaćanje snimljenih datoteka*

Možete dohvatiti snimljene datoteke iz `Illuminate\Http\Request` instance pomoću `file` metode ili koristeći dinamička svojstva. `file` metoda vraća instancu klase `Illuminate\Http\UploadedFile`, koja proširuje PHP `SplFileInfo` klasu i pruža niz metoda za interakciju s datotekom:

```
$file = $request->file('photo');  
  
$file = $request->photo;
```

Možete utvrditi je li datoteka prisutna u zahtjevu koristeći `hasFile` metodu:

```
if ($request->hasFile('photo')) {  
    // ...  
}
```

### *Provjera uspješnih uploada*

Osim provjere je li datoteka postoji, možete provjeriti da nije bilo problema sa uploadom datoteke pomoću `isValid` metode:

```
if ($request->file('photo')->isValid()) {  
    // ...  
}
```

### *Putanje i ekstenzije datoteka*

`UploadedFile` klasa također sadrži metode za pristup potpuno kvalificiranoj putanji datoteke i njenoj ekstenziji<sup>34</sup>. `extension` metoda će pokušati pogoditi ekstenziju datoteke na temelju njenog sadržaja. Ova ekstenzija može se razlikovati od ekstenzije koju je dao klijent:

```
$path = $request->photo->path();  
  
$extension = $request->photo->extension();
```

---

<sup>34</sup> Potpuno kvalificirani naziv domene (engl. fully qualified domain name FQDN) je puna adresa internet hosta ili kompjutera. Omogućava točnu lokaciju unutar sistema imena domene (engl. domain name system - DNS) određivanjem naziva hosta, naziva domene i domene najvišeg nivoa (engl. top-level domain - TLD).

### *Ostale metode datoteka*

Postoji niz drugih metoda dostupnih na `UploadedFile` instancama. Više informacija o ovim metodama potražite u online [API dokumentaciji za ovu klasu](#).

### *Pohranjivanje datoteka za snimanje*

Za pohranjivanje datoteka za snimanje obično ćete koristiti jedan od svojih konfiguriranih datotečnih sistema. `UploadedFile` klasa ima `store` metodu koja će premjestiti snimljenu datoteku na jedan od vaših diskova, što može biti lokacija u vašem lokalnom datotečnom sistemu ili lokacija za pohranu u oblaku (engl. cloud) kao što je pCloud, Sync.com, Icedrive, MEGA, Google drive, OneDrive, Koofr, Dropbox, MediaFire, Degoo.

Metoda `store` prihvaća putanju gdje bi datoteka trebala biti pohranjena u odnosu na konfigurirani početni (engl. root) direktorij datotečnog sistema. Ova putanja ne bi trebala sadržavati naziv datoteke jer će se automatski generirati jedinstveni ID koji će služiti kao naziv datoteke.

`store` metoda također prihvaća opcionalni drugi argument za naziv diska koji bi se trebao koristiti za pohranu datoteke. Metoda će vratiti putanju datoteke u odnosu na root diska:

```
$path = $request->photo->store('images');  
  
$path = $request->photo->store('images', 's3');
```

Ako ne želite da se naziv datoteke automatski generira, možete koristiti `storeAs` metodu koja prihvaća putanju, naziv datoteke i naziv diska kao svoje argumente:

```
$path = $request->photo->storeAs('images', 'filename.jpg');  
  
$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

### **NAPOMENA:**



Za više informacija o pohranjivanju datoteka u Laravel-u pogledajte kompletnu dokumentaciju o pohranjivanju datoteka.

### *Konfiguriranje proxy-ja kojim vjerujete*

Kada pokrećete svoje aplikacije iza djelatila opterećenja (engl. load balancer) koji prekida TLS/SSL certifikate, možete primijetiti da vaša aplikacija ponekad ne generira HTTPS veze kada koristite `url` pomoćnik (engl. helper). To je obično zato što se vašoj aplikaciji proslijeđuje promet s vašeg djelatila opterećenja na port 80 i on ne zna da bi trebao generirati sigurnu vezu.

Da biste to riješili, možete upotrijebiti `App\Http\Middleware\TrustProxies` middleware koji je uključen u vašu Laravel aplikaciju, a koji vam omogućava da brzo prilagodite djelatila opterećenja (engl. load balancers) ili proxy-je kojima vaša aplikacija treba vjerovati. Vaši pouzdani proxy-ji trebali bi biti navedeni korištenjem `trustProxies` middleware metode u `bootstrap/app.php` datoteci vaše aplikacije:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustProxies(at: [
        '192.168.1.1',
        '10.0.0.0/8',
    ]);
})
```

Uz konfiguriranje pouzdanih proxyja, također možete konfigurirati proxy zaglavlja koja bi trebala biti pouzdana:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustProxies(headers: Request::HEADER_X_FORWARDED_FOR |
        Request::HEADER_X_FORWARDED_HOST |
        Request::HEADER_X_FORWARDED_PORT |
        Request::HEADER_X_FORWARDED_PROTO |
        Request::HEADER_X_FORWARDED_AWS_ELB
    );
})
```

#### NAPOMENA:



Ako koristite AWS Elastic Load Balancing, vaša `$headers` vrijednost bi trebala biti `Request::HEADER_X_FORWARDED_AWS_ELB`. Ako vaš djelatelj opterećenja (engl. load balancer) koristi standardno Proslijeđeno zaglavlje iz RFC 7239, vrijednost zaglavlja bi trebala biti `Request::HEADER_FORWARDED`. Za više informacija o konstantama koje se mogu koristiti u `$headers` svojstvu, pogledajte Symfony-jevu dokumentaciju o pouzdanim proxy-jima.

#### *Povjerenje u sve proxy-je*

Ako koristite Amazon AWS ili drugog pružatelja djelatelja opterećenja u „oblaku“, možda ne znate IP adrese svojih stvarnih djelatelja. U ovom slučaju možete upotrijebiti `*` za povjerenje svim proxy-jima:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->trustProxies(at: '*');
})
```

#### *Konfiguriranje pouzdanih host-ova*

Prema podrazumijevanim postavkama, Laravel će odgovoriti na sve zahtjeve koje primi bez obzira na sadržaj HTTP zahtjeva `Host` zaglavlja. Osim toga, `Host` vrijednost zaglavlja koristit će se kada generirate apsolutni URL vaše aplikacije tokom web zahtjeva.

Da biste omogućili TrustHosts middleware, trebali biste pozvati metodu middleware-a `trustHosts` u datoteci `bootstrap/app.php` svoje aplikacije. Koristeći argument `at` ove metode, možete navesti nazive hostova na koje vaša aplikacija treba odgovoriti. Dolazni zahtjevi s drugim Host zaglavljima bit će odbijeni:

```
->withMiddleware(function (Middleware $middleware) {  
    $middleware->trustHosts(at: ['laravel.test']);  
})
```

Prema zadanim postavkama, zahtjevi koji dolaze iz poddomena URL-a aplikacije također se automatski smatraju pouzdanim. Ako želite onemogućiti ovo ponašanje, možete koristiti `subdomains` argument:

```
->withMiddleware(function (Middleware $middleware) {  
    $middleware->trustHosts(at: ['laravel.test'], subdomains: false);  
})
```

Ako trebate pristupiti konfiguracijskim datotekama ili bazi podataka svoje aplikacije kako biste odredili svoje pouzdane hostove, možete osigurati anonimnu funkciju (engl. closure) argumenta `at`:

```
->withMiddleware(function (Middleware $middleware) {  
    $middleware->trustHosts(at: fn () => config('app.trusted_hosts'));  
})
```



## HTTP odgovori

### Kreiranje odgovora

#### *Stringovi i matrice*

Sve rute i kontroleri trebaju vratiti odgovor koji se šalje natrag korisničkom browser-u. Laravel nudi nekoliko različitih načina vraćanja odgovora. Najosnovniji odgovor je vraćanje niza iz rute ili kontrolera. Radna okolina će automatski pretvoriti string u puni HTTP odgovor:

```
Route::get('/', function () {  
    return 'Zdravo svijete!';  
});
```

Osim vraćanja stringova iz vaših ruta i kontrolera, možete vraćati i matrice. Radna okolina će automatski pretvoriti matricu u JSON odgovor:

```
Route::get('/', function () {  
    return [1, 2, 3];  
});
```

#### **NAPOMENA:**



Jeste li znali da također možete vratiti Eloquent kolekcije iz svojih ruta ili kontrolera? Automatski će se pretvoriti u JSON. Pokušajte!

#### *Objekti odgovora*

Obično nećete samo vraćati jednostavne stringove ili matrice iz svojih radnji rute. Umjesto toga, vraćat ćete pune `Illuminate\Http\Response` instance ili pogleda.

Vraćanje pune `Response` instance omogućava vam prilagodbu HTTP statusnog koda i zaglavlja odgovora. `Response` instanca nasljeđuje `Symfony\Component\HttpFoundation\Response` klasu koja omogućava različite metode za izgradnju HTTP odgovora:

```
Route::get('/home', function () {  
    return response('Hello World', 200)  
        ->header('Content-Type', 'text/plain');  
});
```

#### *Eloquent modeli i kolekcije*

Također možete vratiti Eloquent ORM modele i kolekcije direktno sa svojih ruta i kontrolera. Kada to učinite, Laravel će automatski pretvoriti modele i kolekcije u JSON odgovore poštujući skrivene attribute modela:

```
use App\Models\User;  
  
Route::get('/user/{user}', function (User $user) {
```

```
    return $user;
});
```

### *Pridruživanje zaglavlja odgovorima*

Imajte na umu da se većina metoda odgovora može ulančati, što omogućava živahnu konstrukciju instanci odgovora. Na primjer, možete upotrijebiti `header` metodu za dodavanje niza zaglavlja u odgovor prije nego što ga pošaljete natrag korisniku:

```
return response($content)
    ->header('Content-Type', $type)
    ->header('X-Header-One', 'Header Value')
    ->header('X-Header-Two', 'Header Value');
```

Ili možete koristiti `withHeaders` metodu za navođenje matrice zaglavlja koja će se dodati odgovoru:

```
return response($content)
    ->withHeaders([
        'Content-Type' => $type,
        'X-Header-One' => 'Header Value',
        'X-Header-Two' => 'Header Value',
    ]);
```

### *Kontrola Middleware keša*

Laravel uključuje `cache.headers` middleware koji se može koristiti za brzo postavljanje `Cache-Control` zaglavlja za grupu ruta. Direktive bi se trebale dati korištenjem „snake case”<sup>35</sup> ekvivalenta odgovarajuće direktive za kontrolu keša i trebale bi biti odvojene točka-zarezom. Ako je `etag` naveden na popisu direktiva, MD5<sup>36</sup> hash<sup>37</sup> sadržaja odgovora automatski će se postaviti kao `etag` identifikator:

```
Route::middleware('cache.headers:public;max_age=2628000;etag')->group(function () {
    Route::get('/privacy', function () {
        // ...
    });
});
```

---

<sup>35</sup> snake\_case je konvencija imenovanja u kojoj se svaki razmak zamjenjuje znakom podvlačenje (\_), a riječi se pišu malim slovima. SCREAMING\_SNAKE\_CASE je varijacija gdje su riječi napisane velikim slovima. Koristi se za konstante u PHP-u, C/C++, Python-u, Javi i shell skriptiranju. CamelCase je praksa pisanja bez razmaka i interpunkcijskih znakova s velikim slovima. Neki stilovi programiranja preferiraju velika mala slova s velikim početnim slovom, drugi ne.

<sup>36</sup> MD5 (algoritam za sažimanje poruka, engl. message-digest algorithm) raspršivanje (engl. hash) je algoritam je jednosmjerna kriptografska funkcija koja prihvata poruku bilo koje dužine kao ulaz i vraća sažetu vrijednost fiksne dužine koja se koristi za provjeru autentičnosti originalne poruke. Nekad se MD5 koristio kao algoritam za provjeru autentičnosti digitalnih potpisa. Kako je zastario, koristi se kao kontrolni zbroj za provjeru integriteta podataka i otkrivanje nenamjernog oštećenja podataka.

<sup>37</sup> Hash ili raspršivanje je proces transformacije bilo kojeg ključa ili stringa znakova u drugu vrijednost. To se obično predstavlja kraćom vrijednošću ili ključem fiksne dužine koji predstavlja i olakšava pronalaženje ili korištenje originalnog stringa. Najpopularnija upotreba hashiranja je implementacija hash tablica.

```
Route::get('/terms', function () {  
    // ...  
});  
});
```

#### *Pridruživanje kolačića uz odgovore*

Možete priložiti kolačić odlaznoj `Illuminate\Http\Response` instanci pomoću `cookie` metode. Ovoj metodi trebate proslijediti naziv, vrijednost i broj minuta koliko bi se kolačić trebao smatrati važećim:

```
return response('Hello World')->cookie(  
    'name', 'value', $minutes  
);
```

`cookie` metoda također prihvaća još nekoliko argumenata koji se rjeđe koriste. Općenito, ovi argumenti imaju istu svrhu i značenje kao argumenti koji bi se dali PHP-ovoj izvornoj `setcookie` metodi:

```
return response('Hello World')->cookie(  
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly  
);
```

Ako želite osigurati slanje kolačića s odlaznim odgovorom, ali još nemate instancu tog odgovora, možete upotrijebiti fasadu `Cookie` za kolačić „u redu čekanja“ (engl. queue) za prilog odgovoru kada se šalje. `queue` metoda prihvaća argumente potrebne za stvaranje instance kolačića. Ovi će kolačići biti priloženi odlaznom odgovoru prije nego što se pošalje browser-u:

```
use Illuminate\Support\Facades\Cookie;  
  
Cookie::queue('name', 'value', $minutes);
```

#### *Generiranje instanci kolačića*

Ako želite generirati `Symfony\Component\HttpFoundation\Cookie` instancu koja se kasnije može pridružiti instanci odgovora, možete koristiti globalni `cookie` pomoćnik. Ovaj kolačić neće biti poslan natrag klijentu osim ako nije pridružen instanci odgovora:

```
$cookie = cookie('name', 'value', $minutes);  
  
return response('Hello World')->cookie($cookie);
```

#### *Kolačići koji istječu ranije*

Kolačić možete ukloniti kada istekne pomoću `withoutCookie` metode odlaznog odgovora:

```
return response('Hello World')->withoutCookie('name');
```

Ako još nemate instancu odlaznog odgovora, možete upotrijebiti `Cookie` fasadu `expire` metodu da istekne kolačić:

```
Cookie::expire('name');
```

### Kolačići i šifriranje

Prema podrazumijevanim postavkama, svi kolačići koje generira Laravel šifrirani su i potpisani tako da ih klijent ne može mijenjati niti čitati. Ako želite onemogućiti šifriranje (engl. encryption) za podskup kolačića koje generira vaša aplikacija, možete koristiti `$except` svojstvo middleware-a `App\Http\Middleware\EncryptCookies` koje se nalazi u `app/Http/Middleware` direktoriju:

```
/**
 * Imena kolačića koji ne bi trebala biti šifrirana.
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

### Preusmjeravanja (redirekcije)

Odgovori za preusmjeravanje su instance klase `Illuminate\Http\RedirectResponse` i sadrže odgovarajuća zaglavlja potrebna za preusmjeravanje korisnika na drugi URL. Postoji nekoliko načina za generiranje `RedirectResponse` instance. Najjednostavnija metoda je korištenje globalnog `redirect` helper-a:

```
Route::get('/dashboard', function () {
    return redirect('home/dashboard');
});
```

Ponekad ćete možda htjeti preusmjeriti korisnika na njegovu prethodnu lokaciju, na primjer kada je poslana forma nevažeća. To možete učiniti korištenjem globalne `back` funkcije pomoćnika (engl. helper). Budući da ovo svojstvo koristi sesiju, provjerite li ruta koja poziva `back` funkciju koristi `web` middleware grupu:

```
Route::post('/user/profile', function () {
    // Potvrdit zahtjev...

    return back()->withInput();
});
```

### *Preusmjeravanje na imenovane rute*

Kada pozovete `redirect` pomoćnika bez parametara, vraća se `Illuminate\Routing\Redirector` instanca, što vam omogućava da pozovete bilo koju metodu u `Redirector` instanci. Na primjer, za generiranje `RedirectResponse` do imenovane rute, možete koristiti `route` metodu:

```
return redirect()->route('login');
```

Ako vaša ruta ima parametre, možete ih proslijediti kao drugi argument `route` metode:

```
// Za rutu sa sljedećim URI-jem: /profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

### *Popunjavanje parametara pomoću Eloquent modela*

Ako preusmjeravate na rutu s parametrom „ID“ koji se popunjava iz modela Eloquent, možete proslijediti sam model. ID će se automatski izdvojiti:

```
// Za rutu sa sljedećim URI-jem: /profile/{id}

return redirect()->route('profile', [$user]);
```

Ako želite prilagoditi vrijednost koja se nalazi u parametru rute, možete navesti stupac u definiciji parametra rute (`/profile/{id:slug}`) ili možete nadjačati `getRouteKey` metodu na svom Eloquent modelu:

```
/**
 * Dohvati vrijednost ključa rute modela.
 */
public function getRouteKey(): mixed
{
    return $this->slug;
}
```

### *Preusmjeravanje na Radnje kontrolera*

Također možete generirati preusmjeravanja na [radnje kontrolera](#). Da biste to učinili, proslijedite kontroler i naziv radnje `action` metodi:

```
use App\Http\Controllers\UserController;

return redirect()->action([UserController::class, 'index']);
```

Ako vaša ruta kontrolera zahtijeva parametre, možete ih proslijediti kao drugi argument `action` metode:

```
return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

#### *Preusmjeravanje na vanjske domene*

Ponekad ćete možda morati napraviti redirekciju na domenu van vaše aplikacije. To možete učiniti pozivanjem `away` metode koja stvara `RedirectResponse` bez ikakvog dodatnog kodiranja URL-a, validacije ili provjere:

```
return redirect()->away('https://www.google.com');
```

#### *Preusmjeravanje s blicanim podacima o sesiji*

Preusmjeravanje na novi URL i blicanje podataka u sesiju obično se obavljaju u isto vrijeme. Obično se to radi nakon uspješnog izvođenja radnje kada blicate poruku o uspjehu sesije. Radi praktičnosti, možete kreirati `RedirectResponse` instancu i blic podatke u sesiji u jednom, tekućem lancu metode:

```
Route::post('/user/profile', function () {
    // ...

    return redirect('dashboard')->with('status', 'Profil ažuriran!');
});
```

Nakon što je korisnik preusmjeren, možete prikazati blic poruku iz sesije . Na primjer, koristeći Blade sintaksu:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

#### *Preusmjeravanje s unosom*

Možete upotrijebiti `withInput` metodu koju omogućava `RedirectResponse` instanca za blicanje ulaznih podataka trenutnog zahtjeva u sesiju prije preusmjeravanja korisnika na novu lokaciju. To se obično radi ako je korisnik naišao na grešku provjere valjanosti. Nakon što je unos blican u sesiji, možete ga lako [dohvatiti](#) tokom sljedećeg zahtjeva za ponovno popunjavanje forme:

```
return back()->withInput();
```

### Druge vrste odgovora

`response` pomoćnik može koristiti za generiranje drugih vrsta odgovora. Kada se `response` pomoćnik pozove bez argumenata, implementacija `Illuminate\Contracts\Routing\ResponseFactory` ugovora (engl. contract) se vraća. Ovaj ugovor nudi nekoliko korisnih metoda za generiranje odgovora.

#### *View odgovori*

Ako vam je potrebna kontrola nad statusom i zaglavljima odgovora, ali također trebate vratiti pogled (engl. view) kao sadržaj odgovora, trebali biste koristiti `view` metodu:

```
return response()
    ->view('hello', $data, 200)
    ->header('Content-Type', $type);
```

Naravno, ako ne trebate proslijediti prilagođeni HTTP kod statusa ili prilagođena zaglavlja, možete koristiti globalnu `view` funkciju pomoćnika (engl. helper).

#### *JSON odgovori*

`json` metoda će automatski postaviti `Content-Type` zaglavlje na `application/json`, kao i pretvoriti danu matricu u JSON pomoću `json_encode` PHP funkcije:

```
return response()->json([
    'name' => 'Ana',
    'state' => 'BA',
]);
```

Ako želite kreirati JSONP odgovor, možete koristiti `json` metodu u kombinaciji s `withCallback` metodom:

```
return response()
    ->json(['name' => 'Ana', 'state' => 'BA'])
    ->withCallback($request->input('callback'));
```

#### *Preuzimanja datoteka*

`download` metoda se može koristiti za generiranje odgovora koji tjera korisnikov browser da preuzme datoteku na danoj putanji. `download` metoda prihvaća naziv datoteke kao drugi argument metode, koji će odrediti naziv datoteke koji vidi korisnik koji preuzima datoteku. Konačno, možete proslijediti matricu HTTP zaglavlja kao treći argument metode:

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);
```

**UPOZORENJE:**

Symfony HttpFoundation, koji upravlja učitavanjima datoteka, zahtijeva da datoteka koja se preuzima ima ASCII naziv datoteke.

*Datoteka Response*

Metoda datoteke može se koristiti za prikaz datoteke, kao što je slika ili PDF, direktno u korisničkom browser-u umjesto pokretanja preuzimanja. Ova metoda prihvaća apsolutnu putanju do datoteke kao svoj prvi argument i matricu zaglavlja kao svoj drugi argument:

```
return response()->file($pathToFile);

return response()->file($pathToFile, $headers);
```

*Streaming Response*

Strujanjem podataka klijentu kako se generiraju, možete značajno smanjiti korištenje memorije i poboljšati performanse, posebno za vrlo velike odgovore. Stream-ani odgovori omogućavaju klijentu da započne obradu podataka prije nego što server završi s slanjem:

```
function streamedContent(): Generator {
    yield 'Zdravo, ';
    yield 'svijete!';
}

Route::get('/stream', function () {
    return response()->stream(function (): void {
        foreach (streamedContent() as $chunk) {
            echo $chunk;
            ob_flush();
            flush();
            sleep(2); // Simuliraj kašnjenje između dijelova...
        }
    }, 200, ['X-Accel-Buffering' => 'no']);
});
```

**NAPOMENA:**

Interno, Laravel koristi PHP-ovu funkcionalnost baferovanja (engl. buffering)<sup>38</sup> izlaza. Kao što možete vidjeti u gornjem primjeru, trebali biste koristiti funkcije `ob_flush` i `flush` za buffer-iranje sadržaja ka klijentu.

---

<sup>38</sup> Prethodno učitana medijska datoteka ili njen dio čuva se u buffer-u ili privremenoj memoriji. Ovaj proces omogućava da se video, audio, igre ili druge medijske datoteke glatko reproduciraju. Ovime se izbjegava kašnjenje striming prijenosa. Ako mreža ili Internet veza mogu isporučiti streaming medij onoliko brzo koliko je potrebno, buffering nije potreban.



### Stream-ani JSON odgovori

Ako trebate inkrementalno prenositi JSON podatke, možete upotrijebiti metodu `streamJson`. Ova je metoda posebno korisna za velike skupove podataka koje je potrebno postupno slati u browser u formatu koji se lako može analizirati JavaScriptom:

```
use App\Models\User;

Route::get('/users.json', function () {
    return response()->streamJson([
        'users' => User::cursor(),
    ]);
});
```

### Stream-anje download-a

Ponekad ćete možda poželjeti pretvoriti string odgovor dane operacije u odgovor koji se može preuzeti bez potrebe za pisanjem sadržaja operacije na disk. U ovom scenariju možete koristiti metodu `streamDownload`. Ova metoda prihvaća funkcije koje se prosleđuju kao argument drugoj funkciji (engl. callback), naziv datoteke i opcionalnu matricu zaglavlja kao svoje argumente:

```
use App\Services\GitHub;

return response()->streamDownload(function () {
    echo GitHub::api('repo')
        ->contents()
        ->readme('laravel', 'laravel')['contents'];
}, 'laravel-readme.md');
```

### Makronaredbe odgovora

Ako želite definirati prilagođeni odgovor koji možete ponovno koristiti u različitim rutama i kontrolerima, možete koristiti `macro` metodu na `Response` fasadi. Obično biste trebali pozvati ovu metodu iz `boot` metode jednog od pružatelja usluga vaše aplikacije, kao što je `App\Providers\AppServiceProvider` pružatelj usluge:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap (Pokrenite) bilo koju aplikacijsku uslugu.
     */
}
```

```
*/  
public function boot(): void  
{  
    Response::macro('caps', function (string $value) {  
        return Response::make(strtoupper($value));  
    });  
}
```

**macro** funkcija prihvaća ime kao svoj prvi argument i anonimnu funkciju (engl. closure) kao svoj drugi argument. Closure makroi izvršit će se kada pozivate imena makroa iz **ResponseFactory** implementacije ili **response** pomoćnika:

```
return response()->caps('foo');
```

## Pogledi

### Uvod

Naravno, nije praktično vraćati cijele stringove HTML dokumenata direktno iz vaših ruta i kontrolera. Srećom, pogledi (engl. view) pružaju prikladan način za smještanje našeg HTML-a u odvojene datoteke.

Pogledi odvajaju logiku vašeg kontrolera/aplikacije od logike vaše prezentacije i pohranjuju se u `resources/views` direktoriju. Kada koristite Laravel, predlošci pogleda obično se pišu pomoću Blade jezika za predloške. Jednostavan pogled može izgledati otprilike ovako:

```
<!-- Pogled pohranjen u resources/views/greeting.blade.php -->
<html>
  <body>
    <h1>Zdravo, {{ $name }}</h1>
  </body>
</html>
```

Budući da je ovaj pogled pohranjen u `resources/views/greeting.blade.php`, možemo ga vratiti koristeći globalni `view` pomoćnik (engl. helper) ovako:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'Branko']);
});
```

### BILJEŠKA:



Tražite više informacija o tome kako napisati Blade predloške? Za početak pogledajte potpunu Blade dokumentaciju .

### *Pisanje pogleda u React / Vue*

Umjesto da svoje frontend predloške pišu u PHP-u pomoću Blade-a, mnogi programeri počeli su radije pisati svoje predloške koristeći React ili Vue. Laravel to čini bezbolnim zahvaljujući [Inertia](#) , biblioteci koja olakšava povezivanje vašeg React/Vue frontenda s vašim Laravel backend-om bez tipične kompleksnosti izgradnje SPA-a.

Naši [početni paketi](#) Breeze i Jetstream daju vam sjajnu početnu točku za vašu sljedeću Laravel aplikaciju koju pokreće Inertia. Osim toga, [Laravel Bootcamp](#) pruža potpunu demonstraciju izrade Laravel aplikacije koju pokreće Inertia, uključujući primjere u Vue i React.

### Kreiranje i renderiranje pogleda

Pogled (engl. view) možete stvoriti postavljanjem datoteke s `.blade.php` ekstenzijom vaše aplikacije u direktorij `resources/views` ili korištenjem `make:view` Artisan naredbe:

```
php artisan make:view greeting
```

`.blade.php` ekstenzija obavještava radnu okolinu da datoteka sadrži Blade predložak (engl. template). Blade predlošci sadrže HTML, kao i Blade direktive koje vam omogućuju jednostavno ponavljanje vrijednosti, stvaranje „if“ izjava, ponavljanje podataka i još mnogo toga.

Nakon što ste kreirali pogled, možete ga vratiti iz jedne od ruta ili kontrolera vaše aplikacije koristeći globalni `view` pomoćnik:

```
Route::get('/', function () {  
    return view('greeting', ['name' => 'Branko']);  
});
```

Pogledi se također mogu vratiti pomoću `View` fasade:

```
use Illuminate\Support\Facades\View;  
  
return View::make('greeting', ['name' => 'Branko']);
```

Kao što možete vidjeti, prvi argument proslijeđen `view` pomoćniku odgovara nazivu datoteke pogleda u `resources/views` direktoriju. Drugi argument je matrica s podacima koji bi trebali biti dostupni pogledu. U ovom slučaju, mi proslijeđujemo `name` varijablu, koja je prikazana u pogledu koristeći Blade sintaksu.

### *Ugniježđeni View direktoriji*

Pogledi također mogu biti ugniježđeni unutar poddirektorija `resources/views` direktorija. Notacija „točka“ može se koristiti za referencu ugniježđenih prikaza. Na primjer, ako je vaš pogled pohranjen u `resources/views/admin/profile.blade.php`, možete ga vratiti s jedne od ruta/kontrolora svoje aplikacije ovako:

```
return view('admin.profile', $data);
```

### **UPOZORENJE:**



Nazivi direktorija pogleda ne smiju sadržavati `.` znak.

### *Kreiranje prvog dostupnog pogleda*

Koristeći `first` metodu `View` fasade, možete kreirati prvi pogled koji postoji u danom matrici pogleda. Ovo može biti korisno ako vaša aplikacija ili paket dopušta prilagođavanje ili prepisivanje preko postojećih pogleda:

```
use Illuminate\Support\Facades\View;  
  
return View::first(['custom.admin', 'admin'], $data);
```

### Utvrđivanje postoji li pogled

Ako trebate utvrditi postoji li pogled, možete koristiti fasadu `View`. Metoda `exists` će se vratiti `true` ako pogled postoji:

```
use Illuminate\Support\Facades\View;

if (View::exists('admin.profile')) {
    // ...
}
```

### Prijenos podataka ka pogledima

Kao što ste vidjeli u prethodnim primjerima, možete proslijediti matricu podataka pogledima kako bi ti podaci bili dostupni pogledu:

```
return view('greetings', ['name' => 'Mladen']);
```

Kada prosljeđujete informacije na ovaj način, podaci bi trebali biti matrica s parovima ključ/vrijednost. Nakon što unesete podatke u pogled, možete pristupiti svakoj vrijednosti unutar svog pogleda koristeći ključeve podataka, kao što je `<?php echo $name; ?>`.

Kao alternativu prosljeđivanju kompletne matrice podataka `view` funkciji pomoćnika, možete koristiti `with` metodu za dodavanje pojedinačnih dijelova podataka u pogled. Metoda `with` vraća instancu objekta pogleda tako da možete nastaviti s ulančavanjem metoda prije vraćanja pogleda:

```
return view('greeting')
    ->with('name', 'Mladen')
    ->with('occupation', 'Astronaut');
```

### Dijeljenje podataka sa svim pogledima

Povremeno ćete možda trebati dijeliti podatke sa svim pogledima koje prikazuje vaša aplikacija. To možete učiniti `share` metodom `View` fasade. Obično biste trebali uputiti pozive `share` metodi unutar davatelja usluge `boot` metode. Možete ih slobodno dodati u `App\Providers\AppServiceProvider` klasu ili generirati odvojenog pružatelja usluge za njihov smještaj:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Registrirajte bilo koju aplikacijsku uslugu.
     */
    public function register(): void
```

```

{
    // ...
}

/**
 * Pokrenite (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    View::share('key', 'value');
}
}

```

### Pogled Composer-a

Pogledi Composer-a su funkcije koje se prosljeđuju kao argument drugoj funkciji (engl. callback) ili metode klase koje se pozivaju kada se pogled obavlja. Ako imate podatke koje želite vezati uz pogled svaki put kada se taj pogled obavlja, kompozitor pogleda (engl. view composer) može vam pomoći organizirati tu logiku na jednom mjestu. Kompozitori pogleda (engl. view composer) mogu se pokazati posebno korisni ako isti pogled vraća više ruta ili kontrolera unutar vaše aplikacije i uvijek treba određeni dio podataka.



U pravilu, kompozitori pogleda bit će registrirani unutar jednog od pružatelja usluga vaše aplikacije. U ovom primjeru pretpostavit ćemo da smo stvorili novi `App\Providers\ViewServiceProvider` za ovu logiku.

Koristit ćemo `composer` metodu `View` fasade za registraciju kompozitora pogleda. Laravel ne uključuje zadani direktorij za kompozitora pogleda koji se temelje na klasama, tako da ih možete organizirati kako god želite. Na primjer, možete stvoriti `app/View/Composers` direktorij za smještaj svih kompozitora pogleda vaše aplikacije:

```

<?php

namespace App\Providers;

use App\View\Composers\ProfileComposer;
use Illuminate\Support\Facades;
use Illuminate\Support\ServiceProvider;
use Illuminate\View\View;

class ViewServiceProvider extends ServiceProvider
{
    /**
     * Registriraj bilo koju aplikacijsku uslugu.
     */
    public function register(): void
    {
    }
}

```

```
{
    // ...
}

/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    // Koristi kompozitora temeljenog na klasi...
    Facades\View::composer('profile', ProfileComposer::class);

    // Koristi kompozitora temeljnog na closure...
    Facades\View::composer('welcome', function (View $view) {
        // ...
    });

    Facades\View::composer('dashboard', function (View $view) {
        // ...
    });
}
}
```

Sada kada smo registrirali kompozitora, `compose` metoda `App\View\Composers\ProfileComposer` klase će se izvršiti svaki puta kada se renderira `profile` pogled. Pogledajmo primjer klase kompozitora:

```
<?php

namespace App\View\Composers;

use App\Repositories\UserRepository;
use Illuminate\View\View;

class ProfileComposer
{
    /**
     * Kreiraj novi profil kompozitora.
     */
    public function __construct(
        protected UserRepository $users,
    ) {}

    /**
     * Poveži podatke s pogledom.
     */
}
```

```
*/  
public function compose(View $view): void  
{  
    $view->with('count', $this->users->count());  
}  
}
```

Kao što možete vidjeti, svi kompozitori pogleda (engl. view composers) rješavaju se pomoću kontejnera usluga, tako da možete nagovijestiti tip (engl. type-hint) svih zavisnosti koje su vam potrebne unutar konstruktora kompozitora.

#### *Pridruživanje konstruktora na više pogleda*

Možete priložiti kompozitor pogleda (engl. view composer) na više pogleda odjednom prosljeđivanjem matrice pogleda kao prvog argumenta `composer` metode:

```
use App\Views\Composers\MultiComposer;  
use Illuminate\Support\Facades\View;  
  
View::composer(  
    ['profile', 'dashboard'],  
    MultiComposer::class  
);
```

`composer` metoda također prihvaća `*` znak kao zamjenski znak, što vam omogućava da pridružite kompozitora svim pogledima:

```
use App\View\Creators\ProfileCreator;  
use Illuminate\Support\Facades\View;  
  
View::creator('profile', ProfileCreator::class);
```

#### *Kreatori pogleda*

View „kreatori“ su vrlo slični kompozitor pogleda (engl. view composer); međutim, oni se izvršavaju odmah nakon instanciranja pogleda umjesto da čekate da se pogled prikaže. Za registraciju kreatora pogleda upotrijebite `creator` metodu:

```
use App\View\Creators\ProfileCreator;  
use Illuminate\Support\Facades\View;  
  
View::creator('profile', ProfileCreator::class);
```



### Optimiziranje pogleda

Prema podrazumijevanim postavkama, pogledi Blade predložaka sastavljaju se na zahtjev. Kada se izvrši zahtjev koji izvodi pogled, Laravel će utvrditi postoji li kompajlirana verzija pogleda. Ako datoteka postoji, Laravel će tada utvrditi da li je nekompajlirani pogled mijenjan poslije prevedenog pogleda. Ako kompajlirani pogled ne postoji ili je nekompajlirani pogled mijenjan, Laravel će ponovno kompajlirati pogled.

Kompajliranje pogleda tokom zahtjeva može imati mali negativan utjecaj na izvedbu, zato Laravel omogućava `view:cache` Artisan naredbu za pred-kompajliranje svih pogleda koje koristi vaša aplikacija. Za povećanje performansi, možda ćete htjeti pokrenuti ovu naredbu kao dio procesa implementacije:

```
php artisan view:cache
```

Možete koristiti `view:clear` naredbu za brisanje keša prikaza:

```
php artisan view:clear
```

## Blade predlošci

### Uvod

Blade je jednostavan, ali moćan mehanizam za izradu predložaka koji je uključen u Laravel. Za razliku od nekih PHP strojeva za izradu predložaka, Blade vas ne ograničava u korištenju običnog PHP koda u vašim predlošcima. Zapravo, svi Blade predlošci kompajlirani su u običan PHP kod i pohranjeni u keš dok se ne izmijene, što znači da Blade u biti ne dodaje dodatne troškove (eng. overhead)<sup>39</sup> vašoj aplikaciji. Datoteke predložaka Blade koriste `.blade.php` ekstenziju datoteke i obično su pohranjene u `resources/views` direktoriju.

Blade pogledi mogu se vratiti iz ruta ili kontrolera pomoću globalnog `view` pomoćnika. Naravno, kao što je spomenuto u dokumentaciji o views tj. pogledima, podaci se mogu proslijediti Blade pogledu pomoću drugog argumenta `view` pomoćnika:

```
Route::get('/', function () {  
    return view('greeting', ['name' => 'Finn']);  
});
```

### Brži i snažniji Blade s Livewire

Želite podići svoje Blade predloške na viši nivo i s lakoćom izgraditi dinamičke interface? Provjerite Laravel Livewire. Livewire vam omogućava pisanje Blade komponenti koje su proširene dinamičkom funkcionalnošću koja bi obično bila moguća samo preko frontend-a interface-a kao što su React ili Vue, pružajući sjajan pristup izgradnji modernih, reaktivnih frontend-a bez kompleksnosti, prikazivanjem na strani klijenta ili koraka izgradnje nekih od JavaScript radnih okolina.

### Prikaz podataka

Možete prikazati podatke koji se prosljeđuju vašim Blade pogledima tako da varijablu stavite u vitičaste zagrade. Na primjer, s obzirom na sljedeću rutu:

```
Route::get('/', function () {  
    return view('welcome', ['name' => 'Mladen']);  
});
```

Možete prikazati sadržaj `name` varijable ovako:

```
Hello, {{ $name }}.
```

### NAPOMENA:



Bladeove `{{ }}` echo izjave automatski se šalju kroz PHP `htmlspecialchars` funkciju kako bi se spriječili XSS<sup>40</sup> napadi.

---

<sup>39</sup> Kombinacija više potrošenog vremena ili indirektnog vremena potrebnog za izračun, memorije, propusnosti (eng. bandwidth) ili drugih resursa koji su potrebni za izvođenje određenog zadatka.

<sup>40</sup> Cross-site scripting (XSS) je napad u kojem napadač injektira zlonamjerne izvršne skripte u kôd pouzdane aplikacije ili web stranice sa strane korisnika. Napadači često pokreću XSS napad slanjem zlonamjerne poveznice korisniku i navođenjem korisnika da je klikne. Zlonamjerna skripta može pristupiti svim kolačićima, tokenima

Niste ograničeni na prikazivanje sadržaja varijabli proslijeđenih pogledu. Također možete ponoviti rezultate bilo koje PHP funkcije. U stvari, možete staviti bilo koji PHP kod koji želite unutar Blade echo naredbe:

```
Trenutačna UNIX vremenska oznaka (engl. timestamp) je {{ time() }}.
```

#### Kodiranje HTML entiteta

Prema podrazumijevanim postavkama, Blade (i Laravel `e` funkcija) dvostruko će kodirati HTML entitete. Ako želite onemogućiti dvostruko kodiranje, pozovite `Blade::withoutDoubleEncoding` metodu iz `boot` metode vašeg `AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Pokrenite (engl. Bootstrap) bilo koju aplikacijsku uslugu.
     */
    public function boot(): void
    {
        Blade::withoutDoubleEncoding();
    }
}
```

#### Prikaz unescaping podataka

Prema podrazumijevanim postavkama, Blade `{{ }}` izjave se automatski šalju kroz PHP `htmlspecialchars` funkciju kako bi se spriječili XSS napadi. Ako ne želite da vaši podaci budu escaped, možete koristiti sljedeću sintaksu:

```
Hello, {!! $name !!}.
```

#### UPOZORENJE:



Budite vrlo oprezni kada ponavljate sadržaj koji dostavljaju korisnici vaše aplikacije. Obično biste trebali koristiti sintaksu dvostrukih vitičastih zagrada za izbjegavanje prekida, kako biste spriječili XSS napade kod prikazivanja korisničkih podataka.

---

sesije ili drugim osjetljivim informacijama koje preglednik zadržava i koristi s tim mjestom. Ove skripte mogu čak i prepisati sadržaj HTML stranice

### Blade i JavaScript radne okoline

Budući da mnoge JavaScript radne okoline također koriste „vitičaste“ zagrade za označavanje da određeni izraz treba prikazati u browser-u, možete upotrijebiti simbol @ kako bi obavijestili Blade mehanizam za prikazivanje da izraz treba ostati netaknut. Na primjer:

```
<h1>Laravel</h1>

Zdravo, @{{ name }}.
```

U ovom primjeru, @ simbol će ukloniti Blade; međutim, {{ name }} izraz će ostati netaknut od strane Blade motora, dopuštajući da ga prikaže vaša JavaScript radna okolina.

@ simbol se također može koristiti za izbjegavanje Blade direktiva:

```
{{!-- Blade predložak --}}
@@if()

<!-- HTML izlaz -->
@if()
```

### Izvođenje JSON

Ponekad možete proslijediti matricu svom pogledu s namjerom da ga prikazete kao JSON kako biste inicijalizirali JavaScript varijablu. Na primjer:

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

Međutim, umjesto ručnog pozivanja json\_encode, možete koristiti Illuminate\Support\Js::from direktivu metode. from metoda prihvata iste argumente kao PHP json\_encode funkcija; međutim, osigurat će da je dobiveni JSON pravilno izmaknut za uključivanje unutar HTML navodnika. from metoda će vratiti string JSON.parse JavaScript naredbu koja će pretvoriti dani objekt ili matricu u važeći JavaScript objekt:

```
<script>
    var app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

Najnovije verzije Laravel skeleta aplikacije uključuju Js fasadu koja pruža praktičan pristup ovoj funkciji unutar vaših Blade predložaka:

```
<script>
    var app = {{ Js::from($array) }};
```

&lt;/script&gt;

**UPOZORENJE:**

Trebali biste koristiti samo `Js::from` metodu za prikazivanje postojećih varijabli kao JSON. Blade predložak temelji se na regularnim izrazima i pokušaji prosljeđivanja složenog izraza u direktivu mogu uzrokovati neočekivane greške.

*@verbatim direktiva*

Ako prikazujete JavaScript varijable u velikom dijelu svog predloška, možete omotati HTML u `@verbatim` direktivu tako da ne morate svakoj Blade echo naredbi dodavati `@simbol`:

```
@verbatim
    <div class="container">
        Hello, {{ name }}.
    </div>
@endverbatim
```

## Blade direktive

Uz nasljeđivanje predložaka i prikaz podataka, Blade također nudi praktične prečice za uobičajene PHP kontrolne strukture, kao što su uslovne izjave i petlje. Ovi prečice pružaju vrlo jasan, jezgrovit način rada s PHP kontrolnim strukturama, a istovremeno ostaju poznati svojim PHP pandanima.

*If izjave*

Možete konstruirati `if` izjave koristeći `@if`, `@elseif`, `@else` i `@endif` direktive. Ove direktive funkcioniraju identično svojim PHP pandanima:

```
@if (count($records) === 1)
    Imam jedan slog!
}elseif (count($records) > 1)
    Imam više slogova!
@else
    Nemam ni jedan slog!
@endif
```

Radi praktičnosti, Blade također nudi `@unless` direktivu:

```
@unless (Auth::check())
    Niste prijavljeni.
@endunless
```

Uz već spomenute uslovne direktive, `@isset` i `@empty` direktive se mogu koristiti kao prikladni prečice i za odgovarajuće PHP funkcije:

```
@isset($records)
    // $records je definiran i nije null...
@endisset

@empty($records)
    // $records je "empty"...
@endempty
```

#### *Direktive za autentifikaciju*

`@auth` i `@guest` direktive se mogu koristiti za brzo određivanje da li je trenutni korisnik autentificiran ili je gost:

```
@auth
    // Korisnik je autentificiran...
@endauth

@guest
    // Korisnik nije autentificiran...
@endguest
```

Ako je potrebno, možete odrediti zaštitnika provjere autentičnosti (engl. authentication guard) koji bi trebao biti provjeren kada koristite `@auth` i `@guest` direktive:

```
@auth('admin')
    // Korisnik je autentificiran...
@endauth

@guest('admin')
    // Korisnik nije autentificiran...
@endguest
```

#### *Direktive okoline*

Možete provjeriti radi li aplikacija u produkcijskoj okolini pomoću `@production` direktive:

```
@production
    // Sadržaj specifičan za produkciju...
@endproduction
```

Ili, možete odrediti radi li aplikacija u određenoj okolini pomoću `@env` direktive:

```
@env('staging')
    // Aplikacija je pokrenuta u "staging"...
@endenv
```

```
@env(['staging', 'production'])
    // Aplikacija je pokrenuta u "staging" ili "production"...
@endenv
```

### Direktive blokova

Možete odrediti ima li blok nasljeđivanja predloška sadržaj pomoću `@hasSection` direktive:

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

Možete upotrijebiti direktivu `sectionMissing` da odredite nema li blok sadržaja:

```
@sectionMissing('navigation')
    <div class="pull-right">
        @include('default-navigation')
    </div>
@endif
```

### Sesijske direktive

`@session` direktiva se može koristiti za određivanje postoji li vrijednost sesije . Ako vrijednost sesije postoji, procijenit će se sadržaj predloška unutar `@session` i `@endsession` direktiva. Unutar `@session` sadržaja direktive, možete ponoviti `$value` varijablu za prikaz vrijednosti sesije:

```
@session('status')
    <div class="p-4 bg-green-100">
        {{ $value }}
    </div>
@endsession
```

### Switch Izjave

Naredbe Switch mogu se konstruirati korištenjem `@switch`, `@case`, `@break`, `@default` i `@endswitch` direktiva:

```
@switch($i)
    @case(1)
        Prvi case...
    @break
```

```

    @case(2)
        Drugi case...
    @break

    @default
        Default case...
@endswitch

```

### Petlje

Uz uslovne naredbe, Blade nudi jednostavne direktive za rad s PHP strukturama petlje. Opet, svaka od ovih direktiva funkcionira identično svojim PHP pandanima:

```

@for ($i = 0; $i < 10; $i++)
    Trenutna vrijednost je {{ $i }}
@endfor

@foreach ($users as $user)
    <p>Ovo je korisnik {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>Nema korisnika </p>
@endforelse

@while (true)
    <p>U petlji sam zauvijek.</p>
@endwhile

```

#### NAPOMENA:



Tokom ponavljanja kroz `foreach` petlju, možete koristiti [loop varijablu](#) za dobivanje vrijednih informacija o petlji, kao što je jeste li u prvoj ili zadnjoj iteraciji kroz petlju.

Kada koristite petlje, također možete preskočiti trenutnu iteraciju ili završiti petlju koristeći `@continue` i `@break` direktive:

```

@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>
@endforeach

```



```
@if ($user->number == 5)
    @break
@endif
@endforeach
```

Također možete uključiti uslov nastavka (engl. continuation) ili prekida (engl. break) unutar deklaracije direktive:

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

#### *loop varijabla*

Tokom ponavljanja kroz `foreach` petlju, `$loop` varijabla će biti dostupna unutar vaše petlje. Ova varijabla omogućava pristup nekim korisnim informacijama kao što je trenutni indeks petlje i da li ovo prva ili posljednja iteracija kroz petlju:

```
@foreach ($users as $user)
    @if ($loop->first)
        Ovo je prva iteracija.
    @endif

    @if ($loop->last)
        Ovo je zadnja iteracija.
    @endif

    <p>Ovo je korisnik {{ $user->id }}</p>
@endforeach
```

Ako ste u unutrašnjoj petlji, možete pristupiti `$loop` varijabli roditeljske petlje pomoću `parent` svojstva:

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            Ovo je prva iteracija roditeljske petlje.
        @endif
    @endforeach
@endforeach
```

Varijabla `$loop` također sadrži niz drugih korisnih svojstava:

Svojstvo	Opis
<code>\$loop-&gt;index</code>	Indeks trenutne iteracije petlje (počinje od 0).
<code>\$loop-&gt;iteration</code>	Trenutna iteracija petlje (počinje od 1).
<code>\$loop-&gt;remaining</code>	Preostale iteracije u petlji.
<code>\$loop-&gt;count</code>	Ukupan broj stavki u matrici koja se iterira.
<code>\$loop-&gt;first</code>	Da li je ovo prva iteracija kroz petlju.
<code>\$loop-&gt;last</code>	Da li je ovo zadnja iteracija kroz petlju.
<code>\$loop-&gt;even</code>	Da li je ovo neparna iteracija kroz petlju.
<code>\$loop-&gt;odd</code>	Da li je ovo parna iteracija kroz petlju.
<code>\$loop-&gt;depth</code>	Unutrašnji nivo tekuće petlje.
<code>\$loop-&gt;parent</code>	Kada je u unutrašnjoj petlji, roditeljska varijabla petlje

### Uslovne klase i stilovi

`@class` direktiva uslovno kompajlira string CSS klase. Direktiva prihvaća matricu klasa gdje ključ matrica sadrži klasu ili klase koje želite dodati, dok je vrijednost Boolean izraz. Ako element matrice ima numerički ključ, uvijek će biti uključen u prikazanu listu klasa:

```
@php
    $isActive = false;
    $hasError = true;
@endphp

<span @class([
    'p-4',
    'font-bold' => $isActive,
    'text-gray-500' => ! $isActive,
    'bg-red' => $hasError,
])></span>

<span class="p-4 text-gray-500 bg-red"></span>
```

Isto tako, `@style` direktiva se može koristiti za uslovno dodavanje ugrađenih CSS stilova u HTML element:

```
@php
    $isActive = true;
@endphp

<span @style([
    'background-color: red',
    'font-weight: bold' => $isActive,
])></span>

<span style="background-color: red; font-weight: bold;"></span>
```

### Dodatni atributi

Radi praktičnosti, možete upotrijebiti `@checked` direktivu za jednostavno označavanje je li dani unos HTML checkbox „označen“. Ova će se direktiva javiti `checked` ako se navedeni uslov procijeni kao true:

```
<input type="checkbox"
      name="active"
      value="active"
      @checked(old('active', $user->active)) />
```

Nadalje, `@disabled` direktiva se može koristiti za označavanje treba li dati element biti „onemogućen“:

```
<select name="version">
  @foreach ($product->versions as $version)
    <option value="{{ $version }}" @selected(old('version') == $version)>
      {{ $version }}
    </option>
  @endforeach
</select>
```

Osim toga, `@disabled` direktiva se može koristiti za označavanje treba li dati element biti „onemogućen“:

```
<button type="submit" @disabled($errors->isEmpty())>Submit</button>
```

Osim toga, `@readonly` direktiva se može koristiti za označavanje treba li dati element biti „samo za čitanje“:

```
<input type="email"
      name="email"
      value="email@laravel.com"
      @readonly($user->isAdmin()) />
```

Osim toga, `@required` direktiva se može koristiti za označavanje treba li dati element biti „nužan“:

```
<input type="text"
      name="title"
      value="title"
      @required($user->isAdmin()) />
```

*Uključivanje podpregleda***NAPOMENA:**

Iako možete slobodno koristiti `@include` direktivu, Blade [komponente](#) pružaju sličnu funkcionalnost i nude nekoliko prednosti u odnosu na `@include` direktivu kao što su vezanje podataka i atributa.

Bladeova `@include` direktiva omogućava vam da uključite Blade pogled iz drugog pogleda. Sve varijable koje su dostupne nadređenom prikazu bit će dostupne uključenom pogledu:

```
<div>
    @include('shared.errors')

    <form>
        <!--Sadržaj forme -->
    </form>
</div>
```

Iako će uključeni pogled naslijediti sve podatke dostupne u nadređenom (roditeljskom) pogledu, također možete proslijediti matricu dodatnih podataka koji bi trebali biti dostupni uključenom pogledu:

```
@include('view.name', ['status' => 'complete'])
```

Ako pokušate pristupiti `@include` pogledu koji ne postoji, Laravel će izbaciti grešku. Ako želite uključiti pogled koji može ili ne mora biti prisutan, trebali biste upotrijebiti `@includeIf` direktivu:

```
@includeIf('view.name', ['status' => 'complete'])
```

Ako želite `@include` vidjeti da li dati Booleov izraz daje `true` ili `false`, možete koristiti `@includeWhen` i `@includeUnless` direktive:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])
```

```
@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

Da biste uključili prvi pogled koji postoji iz dane matrice pogleda, možete koristiti `includeFirst` direktivu:

```
@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])
```

**UPOZORENJE:**

Trebali biste izbjegavati korištenje konstanti `__DIR__` i `__FILE__` u svojim Blade pogledima jer će se odnositi na lokaciju keširanog, kompajliranog pogleda.

*Renderiranje pogleda za kolekcije*

Možete kombinirati petlje i uključiti u jedan red s Bladeovom `@each` direktivom:

```
@each('view.name', $jobs, 'job')
```

Prvi argument `@each` direktive je pogled koji se prikazuje za svaki element u matrici ili kolekciji. Drugi argument je matrica ili kolekcija preko koje želite iterirati, dok je treći argument naziv varijable koja će biti dodijeljena trenutnoj iteraciji unutar pogleda. Tako, na primjer, ako ponavljate preko matrice `jobs`, obično ćete htjeti pristupiti svakom poslu kao `job` varijabli unutar pogleda. Matrica ključa za trenutnu iteraciju bit će dostupan kao `key` varijabla unutar pogleda.

Također možete proslijediti četvrti argument `@each` direktivi. Ovaj argument određuje pogled koji će se prikazati ako je dana matrica prazna.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

**UPOZORENJE:**

Pogledi generirani pomoću `@each` ne nasljeđuju varijable nadređenog pogleda. Ako podređeni pogled zahtijeva ove varijable, trebali biste umjesto njih koristiti `@foreach` i `@include` direktive.

*@once direktiva*

`@once` direktiva vam omogućava da definirate dio predloška koji će se procijeniti samo jednom po ciklusu iscrtaavanja. Ovo može biti korisno za guranje određenog dijela JavaScripta u zaglavlje stranice pomoću [stekova](#). Na primjer, ako prikazujete određenu komponentu unutar petlje, možda ćete htjeti gurnuti JavaScript samo u zaglavlje prvi put kada se komponenta prikazuje:

```
@once
    @push('scripts')
        <script>
            // Vaš prilagođeni JavaScript...
        </script>
    @endpush
@endonce
```

Budući da se `@once` direktiva često koristi zajedno s `@push` ili `@prepend` direktivama, `@pushOnce` i `@prepend` direktive sudostupne radi vaše udobnosti:

```
@pushOnce('scripts')
    <script>
```

```
// Vaš prilagođeni JavaScript...  
</script>  
@endPushOnce
```

### Sirovi PHP

U nekim je situacijama korisno ugraditi PHP kod u vaše poglede. Možete koristiti Blade `@php` direktivu za izvođenje bloka običnog PHP-a unutar vašeg predloška:

```
@php  
    $counter = 1;  
@endphp
```

Ili ako samo trebate koristiti PHP za uvoz klase, možete koristiti `@use` direktivu:

```
@use('App\Models\Flight')
```

### Komentari

Blade vam također omogućava definiranje komentara u vašim pogledima. Međutim, za razliku od HTML komentara, Blade komentari nisu uključeni u HTML koji vraća vaša aplikacija:

```
{{-- Ovaj komentar neće biti prisutan u prikazanom HTML-u --}}
```

### Komponente

Komponente i slotovi pružaju slične prednosti blokovima, planovima pozicija (engl. layouts) i sadržajima; međutim, nekima bi mentalni model komponenti i slotova mogao biti lakši za razumijevanje. Postoje dva pristupa pisanju komponenti: komponente temeljene na klasi i anonimne komponente.

Za izradu komponente koja se temelji na klasi, možete koristiti `make:component` Artisan naredbu. Da bismo ilustrirali kako koristiti komponente, kreirat ćemo jednostavnu `Alert` komponentu. Naredba `make:component` će smjestiti komponentu u `app/View/Components` direktorij:

```
php artisan make:component Alert
```

`make:component` naredba će također stvoriti predložak pogleda za komponentu. Pogled će biti smješten u `resources/views/components` direktorij. Kada pišete komponente za vlastitu aplikaciju, komponente se automatski otkrivaju unutar `app/View/Components` direktorija i `resources/views/components` direktorija, tako da daljnja registracija komponente obično nije potrebna.

Također možete kreirati komponente unutar poddirektorija:

```
php artisan make:component Forms/Input
```

Naredba iznad kreirat će `Input` komponentu u `app/View/Components/Forms` direktoriju i pogled će biti smješten u `resources/views/components/forms` direktorij.

Ako želite stvoriti anonimnu komponentu (komponentu samo s Blade predloškom i bez klase), možete upotrijebiti `--view` zastavicu kod pozivanja `make:component` naredbe:

```
php artisan make:component forms.input --view
```

Naredba iznad stvorit će Blade datoteku `resources/views/components/forms/input.blade.php` koja se može prikazati kao komponenta pomoću `<x-forms.input />`.

### *Ručno registriranje komponenti paketa*

Kada pišete komponente za vlastitu aplikaciju, komponente se automatski otkrivaju unutar `app/View/Components` imenika i `resources/views/components` direktorija.

Međutim, ako gradite paket koji koristi Blade komponente, morat ćete ručno registrirati svoju klasu komponente i njen HTML tag alias. Obično biste trebali registrirati svoje komponente na `boot` metodi vašeg davatelja usluga (engl. service provider) vašeg paketa:

```
use Illuminate\Support\Facades\Blade;

/**
 * Pokrenite (engl. Bootstrap) usluge vašeg paketa.
 */
public function boot(): void
{
    Blade::component('package-alert', Alert::class);
}
```

Nakon što je vaša komponenta registrirana, može se prikazati pomoću svog tag alias-a:

```
<x-package-alert/>
```

Alternativno, možete koristiti `componentNamespace` metodu za automatsko učitavanje komponenti klasa prema konvenciji. Na primjer, `Nightshade` paket može imati `Calendar` i `ColorPicker` komponente koje se nalaze unutar `Package\Views\Components` prostora imena (engl. namespace):

```
use Illuminate\Support\Facades\Blade;

/**
 * Pokrenite (engl. Bootstrap) usluge vašeg paketa.
 */
public function boot(): void
{
```

```
Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

Ovo će omogućiti korištenje komponenti paketa sa strane njihovog dobavljača prostora imena koristeći `paket-ime::` sintaksu:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade će automatski detektirati klasu koja je povezana s ovom komponentom pascal-casting<sup>41</sup> imenovanjem komponenti. Poddirektoriji su također podržani korištenjem „točka“ notacije.

### *Komponente prikazivanja*

Za prikaz komponente, možete upotrijebiti tag Blade komponente unutar jednog od vaših Blade predložaka. Blade komponente tagove počinju stringom `x-` iz kojeg slijedi naziv kebab-case<sup>42</sup> ime komponente klase:

```
<x-alert/>

<x-user-profile/>
```

Ako je klasa komponente ugniježđena dublje unutar `app/View/Components` direktorija, možete koristiti `.` znak za označavanje ugniježđenosti direktorija. Na primjer, ako pretpostavimo da se komponenta nalazi na `app/View/Components/Inputs/Button.php`, možemo je prikazati ovako:

```
<x-inputs.button/>
```

Ako želite uslovno prikazati svoju komponentu, možete definirati `shouldRender` metodu u svojoj klasi komponente. Ako `shouldRender` metoda vrati `false` komponenta se neće prikazati:

```
use Illuminate\Support\Str;

/**
 * Treba li komponenta biti prikazana
 */
public function shouldRender(): bool
{
```

---

<sup>41</sup> Pascal-casting je pretvaranje varijable iz jednog tipa podataka u drugi po pravilima Paskala. Može biti implicitni i eksplicitni type-casting. Implicitno pretvaranje tipa podataka je prevođenje tipa u kojem je raspon vrijednosti izvornog tipa podataka manji od raspona vrijednosti odredišnog tipa podataka. Eksplicitno pretvaranje tipa podataka je prikazivanje tipa u kojem je raspon vrijednosti izvornog tipa podataka veći od raspona vrijednosti odredišnog tipa podataka.

<sup>42</sup> Kebab-case je konvencija imenovanja programskih varijabli u kojoj programer razmake između riječi zamjenjuje crticom.



```
return Str::length($this->message) > 0;
}
```

### *Prosljeđivanje podataka komponentama*

Možete proslijediti podatke Blade komponentama koristeći HTML attribute. Tvrdokodirane, jednostavne vrijednosti mogu se proslijediti komponenti pomoću jednostavne stringove HTML atributa. PHP izrazi i varijable trebali bi proslijediti komponenti preko atributa koji koriste `:` znak kao prefiks:

```
<x-alert type="error" :message="$message"/>
```

Trebali biste definirati sve attribute podataka komponente u njenom konstruktoru klase. Sva javna svojstva komponente automatski će biti dostupna pogledu komponente. Nije potrebno proslijediti podatke u pogled iz `render` metode komponente:

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;
use Illuminate\View\View;

class Alert extends Component
{
    /**
     * Kreiraj instancu komponente.
     */
    public function __construct(
        public string $type,
        public string $message,
    ) {}

    /**
     * Dobijte pogled / sadržaj koji predstavlja komponentu.
     */
    public function render(): View
    {
        return view('components.alert');
    }
}
```

Kada se vaša komponenta prikazuje, možete prikazati sadržaj javnih varijabli vaše komponente ispisom na ekranu varijabli po imenu:

```
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

### *Konvencije imenovanja*

Argumenti konstruktora komponente trebaju biti specificirani pomoću `camelCase`, dok bi `kebab-case` trebali koristiti kada pozivate nazive argumenata u vašim HTML atributima. Na primjer, dat je sljedeći konstruktor komponente:

```
/**
 * Kreiraj instancu komponente
 */
public function __construct(
    public string $alertType,
) {}
```

`$alertType` argument se može dati komponenti ovako:

```
<x-alert alert-type="danger" />
```

### *Skraćena sintaksa atributa*

Kada prosljeđujete atribut komponentama, također možete koristiti sintaksu „skraćenog atributa“. Ovo je često zgodno jer se imena atributa često podudaraju s imenima varijabli kojima odgovaraju:

```
{{-- Skraćena sintaksa atributa... --}}
<x-profile :$userId :$name />

{{-- Je ekvivalent za... --}}
<x-profile :user-id="$userId" :name="$name" />
```

### *Prikazivanja atributa Escaped sekvenci<sup>43</sup>*

Budući da neke JavaScript radne okoline kao što je [Alpine.js](#) također koriste attribute s prefiksom dvotočke, možete koristiti dvije dvotočke ( `::` ) kao prefiks da obavijestite Blade da atribut nije PHP izraz. Na primjer, dana je sljedeća komponenta:

```
<x-button ::class="{ danger: isDeleting }">
    Submit
</x-button>
```

---

<sup>43</sup> Escaped znakovi su nekada služili za upravljanje perifernim uređajima. To je kombinacija znakova koje imaju različito značenje od doslovnih znakova u njemu. Obično počinju kosom crtom \ (u ovom slučaju ne). To su znakovi ili niz znakova koji se mogu koristiti unutar literala stringa. Svrha izlazne sekvence je predstavljanje znakova koji se ne mogu normalno koristiti pomoću tastature.

Blade će prikazati sljedeći HTML:

```
<button :class="{ danger: isDeleting }">
    Submit
</button>
```

### *Metode komponenti*

Osim javnih varijabli koje su dostupne vašem predlošku komponente, mogu se pozvati sve javne metode na komponenti. Na primjer, zamislite komponentu koja ima `isSelected` metodu:

```
/**
 * Odredite je li dana opcija trenutno odabrana opcija.
 */
public function isSelected(string $option): bool
{
    return $option === $this->selected;
}
```

Ovu metodu možete izvršiti iz svog predloška komponente pozivanjem varijable koja odgovara nazivu metode:

```
<option {{ $isSelected($value) ? 'selected' : '' }} value="{{ $value }}">
    {{ $label }}
</option>
```

### *Pristup atributima i slotovima unutar klasa komponenti*

Blade komponente također vam omogućuju pristup nazivu komponente, atributima i slotu unutar metode klase koja se izvodi. Međutim, kako biste pristupili tim podacima, trebali biste vratiti anonimnu funkciju (engl. closure) iz `render` metode vaše komponente. Anonimna funkcija (engl. closure) će primiti `$data` matricu kao jedini argument. Ova matrica će sadržavati nekoliko elemenata koji pružaju informacije o komponenti:

```
use Closure;

/**
 * Dobij pogled/sadržaj koji predstavlja komponentu.
 */
public function render(): Closure
{
    return function (array $data) {
        // $data['componentName'];
        // $data['attributes'];
        // $data['slot'];
    };
}
```

```

        return '<div>Components content</div>';
    };
}

```

Anonimna funkcija (engl. Closure) koje vraća metoda renderiranja vaše komponente također može primiti polje `$data` kao jedini argument. Ovaj niz će sadržavati nekoliko elemenata koji pružaju informacije o komponenti:

```

return function (array $data) {
    // $data['componentName'];
    // $data['attributes'];
    // $data['slot'];

    return '<div {{ $attributes }}>Sadržaj komponenti </div>';
}

```

#### UPOZORENJE:



Elementi u `$data` matrici nikada ne bi trebali biti direktno ugrađeni u Blade string koji vraća vaša metoda renderiranja jer bi to moglo omogućiti daljinsko izvršavanje koda putem zlonamjernog sadržaja atributa.

`componentName` je jednak imenu korištenom u HTML tagu iza x-prefiksa. Tako će `<x-alert />` i `componentName` biti `alert`. `attributes` element će sadržavati sve atribute koji su bili prisutni u HTML tagu. `slot` element je `Illuminate\Support\HtmlString` instanca sa sadržajem slot-a komponente.

Anonimna funkcija (engl. closure) bi trebala vratiti string. Ako vraćeni string odgovara postojećem pogledu, taj će se pogled prikazati; u suprotnom, vraćeni string bit će procijenjen kao umetnuti (engl. inline) Blade pogled.

#### Dodatne zavisnosti

Ako vaša komponenta zahtijeva zavisnosti iz Laravelovog [kontejnera usluga](#), možete ih navesti prije bilo kojeg atributa podataka komponenti i automatski će ih injektirati kontejner:

```

use App\Services\AlertCreator;

/**
 * Kreiraj instancu komponente.
 */
public function __construct(
    public AlertCreator $creator,
    public string $type,
    public string $message,
) {}

```

### *Skrivanje atributa / metoda*

Ako želite spriječiti da neke javne metode ili svojstva budu izloženi kao varijable vašem predlošku komponente, možete ih dodati `$except` matrici svojstva na svojoj komponenti:

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component
{
    /**
     * Svojstva / metode koje ne bi trebale biti izložene predlošku komponente.
     *
     * @var array
     */
    protected $except = ['type'];

    /**
     * Kreirajte instancu komponente.
     */
    public function __construct(
        public string $type,
    ) {}
}
```

### *Atributi komponente*

Već smo ispitali kako proslijediti attribute podataka komponenti; međutim, ponekad ćete možda morati navesti dodatne HTML attribute, kao što su `class`, koji nisu dio podataka potrebnih za funkcioniranje komponente. Obično ove dodatne attribute želite prenijeti u root element predloška komponente. Na primjer, zamislite da želimo prikazati `alert` komponentu ovako:

```
<x-alert type="error" :message="$message" class="mt-4"/>
```

Svi atributi koji nisu dio konstruktora komponente automatski će se dodati u „vreću atributa“ komponente. Ova vrećica atributa automatski se stavlja na raspolaganje komponenti pomoću `$attributes` varijable. Svi atributi mogu se prikazati unutar komponente navođenjem ove varijable:

```
<div {{ $attributes }}>
    <!-- Component content -->
</div>
```

**UPOZORENJE:**

Korištenje direktiva kao što je `@env` unutar oznaka komponenti trenutno nije podržano. Na primjer, `<x-alert :live="@env('production')"/>` neće se kompajlirati.

*Defalut / Merged atributi*

Ponekad ćete možda morati navesti zadane vrijednosti za atribute ili spojiti dodatne vrijednosti u neke od atributa komponente. Da biste to postigli, možete koristiti `merge` metodu vreće s atributima. Ova je metoda osobito korisna za definiranje skupa zadanih CSS klasa koje bi uvijek trebale biti primijenjene na komponentu:

```
<div {{ $attributes->merge(['class' => 'alert alert-'. $type]) }}>
    {{ $message }}
</div>
```

Ako pretpostavimo da se ova komponenta koristi na sljedeći način:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

Konačni, prikazani HTML komponente izgledat će ovako:

```
<div class="alert alert-error mb-4">
    <!-- Sadržaj varijable $message -->
</div>
```

*Uslovno spajanje klasa*

Ponekad ćete možda poželjeti spojiti klase ako je dani uslov `true`. To možete postići pomoću `class` metode koja prihvaća matricu klasa gdje ključ matrice sadrži klasu ili klase koje želite dodati, dok je vrijednost Boolean izraz. Ako element matrice ima numerički ključ, uvijek će biti uključen u prikazanu listu klasa:

```
<div {{ $attributes->class(['p-4', 'bg-red' => $hasError]) }}>
    {{ $message }}
</div>
```

Ako trebate spojiti druge atribute na svoju komponentu, možete ulančati `merge` metodu na `class` metodu:

```
<button {{ $attributes->class(['p-4'])->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

**NAPOMENA:**

Ako trebate uslovno kompajlirati klase na drugim HTML elementima koji ne bi trebali primati spojene atribute, možete koristiti `@class` direktivu.

### Spajanje atributa bez klase

Kada spajate attribute koji nisu `class` atributi, vrijednosti dane `merge` metodi smatrat će se „podrazumijevanim“ vrijednostima atributa. Međutim, za razliku od `class` atributa, ti atributi neće biti spojeni s injektiranim vrijednostima atributa. Umjesto toga, oni će biti prepisani preko postojećih. Na primjer, `button` implementacija komponente može izgledati ovako:

```
<button {{ $attributes->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

Za prikazivanje komponente dugmeta s prilagođenim `type`, može se navesti kada upotrebljavate komponente. Ako tip nije naveden, koristit će se `button` tip:

```
<x-button type="submit">
    Submit
</x-button>
```

Prikazani HTML `button` komponente u ovom primjeru bio bi:

```
<button type="submit">
    Submit
</button>
```

Ako želite da atribut `class` ima svoju podrazumijevanu vrijednost i injektirane vrijednosti spojene zajedno, možete upotrijebiti `prepends` metodu. U ovom primjeru, `data-controller` atribut će uvijek počinjati s `profile-controller`, a sve dodatne injektirane `data-controller` vrijednosti bit će postavljene nakon ove podrazumijevane vrijednosti:

```
<div {{ $attributes->merge(['data-controller' => $attributes->prepends('profile-
controller')]) }}>
    {{ $slot }}
</div>
```

### Dohvaćanje i filtriranje atributa

Možete filtrirati attribute koristeći `filter` metodu. Ova metoda prihvaća anonimna funkcija (engl. closure) koje bi se trebalo vratiti `true` ako želite zadržati atribut u vreći atributa:

```
{{ $attributes->filter(fn (string $value, string $key) => $key == 'foo') }}
```

Radi praktičnosti, možete koristiti `whereStartsWith` metodu za dohvaćanje svih atributa čiji ključevi počinju danim stringom:

```
{{ $attributes->whereStartsWith('wire:model') }}
```

Suprotno tome, `whereDoesntStartWith` metoda se može koristiti za isključivanje svih atributa čiji ključevi počinju danim stringom:

```
{{ $attributes->whereDoesntStartWith('wire:model') }}
```

Koristeći `first` metodu, možete prikazati prvi atribut u zadanoj vreći atributa:

```
{{ $attributes->whereStartsWith('wire:model')->first() }}
```

Ako želite provjeriti je li atribut prisutan na komponenti, možete koristiti `has` metodu. Ova metoda prihvaća naziv atributa kao svoj jedini argument i vraća Boolean vrijednost koja pokazuje je li atribut prisutan ili ne:

```
@if ($attributes->has('class'))
    <div> Atribut klase je prisutan</div>
@endif
```

Ako se matrici proslijedi `has` metoda, metoda će odrediti jesu li svi zadani atributi prisutni na komponenti:

```
@if ($attributes->has(['name', 'class']))
    <div>Prisutni su svi atributi </div>
@endif
```

`hasAny` metoda se može koristiti za određivanje je li neki od danih atributa prisutan na komponenti:

```
@if ($attributes->has(['name', 'class']))
    <div>Prisutni su svi atributi </div>
@endif
```

Možete dohvatiti vrijednost određenog atributa pomoću `get` metode:

```
{{ $attributes->get('class') }}
```

### Rezervirane ključne riječi

Prema podrazumijevanim postavkama, neke su ključne riječi rezervirane za internu upotrebu Blade-a kako bi se prikazale komponente. Sljedeće ključne riječi ne mogu se definirati kao javna svojstva ili nazivi metoda unutar vaših komponenti:

- `data`
- `render`
- `resolveView`
- `shouldRender`
- `view`
- `withAttributes`
- `withName`



### Slotovi

Često ćete morati proslijediti dodatni sadržaj vašoj komponenti pomoću „slota“. Slotovi komponenti prikazuju se ponavljanjem `$slot` varijable. Kako bismo istražili ovaj koncept, zamislimo da alert komponenta ima sljedeće oznake:

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

Sadržaj možemo proslijediti ubacivanjem `slot` sadržaja u komponentu:

```
<x-alert>
    <strong>Ups!</strong> Nešto je pošlo po zlu!
</x-alert>
```

Ponekad će komponenta možda trebati prikazati više različitih slotova na različitim mjestima unutar komponente. Modificirajmo našu komponentu upozorenja kako bismo omogućili umetanje „naslova“ slota:

```
<!-- /resources/views/components/alert.blade.php -->

<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

Možete definirati sadržaj imenovanog slota pomoću `x-slot` taga. Svaki sadržaj koji nije unutar eksplicitnog `x-slot` taga bit će proslijeđen komponenti u `$slot` varijabli:

```
<x-alert>
    <x-slot:title>
        Server greška
    </x-slot>

    <strong>Ups!</strong> Nešto je pošlo po zlu!
</x-alert>
```

### Doseg slotova

Ako ste koristili JavaScript radnu okolinu kao što je Vue, možda ste upoznati s „dosegom slotova“, koji vam omogućuju pristup podacima ili metodama iz komponente unutar vašeg slot. Možete postići

slično ponašanje u Laravelu definiranjem javnih metoda ili svojstava na svojoj komponenti i pristupom komponenti unutar vašeg slot-a kroz `$component` varijablu. U ovom primjeru pretpostavit ćemo da `x-alert` komponenta ima javnu `formatAlert` metodu definiranu na svojoj klasi komponente:

```
<x-alert>
  <x-slot:title>
    {{ $component->formatAlert('Server Error') }}
  </x-slot>

  <strong>Ups!</strong> Nešto je pošlo po zlu!
</x-alert>
```

Možete pozvati slot `isEmpty` metodu da odredite da li slot sadrži sadržaj:

```
<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
  @if ($slot->isEmpty())
    Ovo je zadani sadržaj ako je slot prazan.
  @else
    {{ $slot }}
  @endif
</div>
```

Nadalje, metoda `hasActualContent` može se koristiti za utvrđivanje sadrži li utor neki „stvarni“ sadržaj koji nije HTML komentar:

```
@if ($slot->hasActualContent())
  Opseg ima sadržaj bez komentara.
@endif
```

### Opseg slotova

Ako ste koristili JavaScript radnu okolinu kao što je Vue, možda ste upoznati s „scoped slots“, koji vam omogućuju pristup podacima ili metodama iz komponente unutar vašeg slot-a. Možete postići slično ponašanje u Laravelu definiranjem javnih metoda ili svojstava na svojoj komponenti i pristupom komponenti unutar vašeg slot-a pomoću varijable `$component`. U ovom primjeru pretpostavit ćemo da komponenta `x-alert` ima javnu metodu `formatAlert` definiranu na svojoj klasi komponente:

```
<x-alert>
  <x-slot:title>
    {{ $component->formatAlert('Server Error') }}
  </x-slot>

  <strong>Ups!</strong> Nešto je pošlo po zlu!
</x-alert>
```

### Atributi slotova

Kao i Blade komponente, možete dodijeliti dodatne [atribute](#) utorima kao što su nazivi CSS klasa:

```
<x-card class="shadow-sm">
  <x-slot:heading class="font-bold">
    Heading
  </x-slot>

  Content

  <x-slot:footer class="text-sm">
    Footer
  </x-slot>
</x-card>
```

Za interakciju s atributima slotova, možete pristupiti `attributes` svojstvu slot varijable. Za više informacija o tome kako komunicirati s atributima, pogledajte dokumentaciju o [atributima komponente](#):

```
@props([
  'heading',
  'footer',
])

<div {{ $attributes->class(['border']) }}>
  <h1 {{ $heading->attributes->class(['text-lg']) }}>
    {{ $heading }}
  </h1>

  {{ $slot }}

  <footer {{ $footer->attributes->class(['text-gray-700']) }}>
    {{ $footer }}
  </footer>
</div>
```

### Umetanje komponenti pogleda

Za vrlo male komponente može se činiti nezgrapnim upravljati i klasom komponente i predložkom pogleda komponente. Iz tog razloga možete vratiti oznaku komponente direktno iz `render` metode:

```
/**
 * Dohvati pogled/sadržaj koji predstavlja komponentu.
 */
public function render(): string
```

```
{
    return <<<'blade'
        <div class="alert alert-danger">
            {{ $slot }}
        </div>
    blade;
}
```

### Generiranje umetnutih komponenti pogleda

Za izradu komponente koja prikazuje umetnuti pogled, možete upotrijebiti `inline` opciju kada izvodite `make:component` naredbu:

```
php artisan make:component Alert --inline
```

### Dinamičke komponente

Ponekad ćete možda morati prikazati komponentu, ali ne znate koju komponentu treba prikazati do vremena izvođenja. U ovoj situaciji možete koristiti ugrađenu `dynamic-component` komponentu Laravel-a za renderiranje komponente na temelju vrijednosti za vrijeme izvođenja (engl. runtime value) ili varijable:

```
// $componentName = "secondary-button";

<x-dynamic-component :component="$componentName" class="mt-4" />
```

### Ručno registriranje komponenti

#### UPOZORENJE:



Sljedeća dokumentacija o ručnom registriranju komponenti primarno je primjenjiva na one koji pišu Laravel pakete koji uključuju komponente pogleda. Ako ne pišete paket, ovaj dio dokumentacije komponente možda neće biti relevantan za vas.

Kada pišete komponente za vlastitu aplikaciju, komponente se automatski otkrivaju unutar `app/View/Components` direktorija i `resources/views/components` direktorija.

Međutim, ako gradite paket koji koristi Blade komponente ili postavljate komponente u nestandardne direktorije, morat ćete ručno registrirati svoju klasu komponente i njen HTML tag alias kako bi Laravel znao gdje pronaći komponentu. Obično biste trebali registrirati svoje komponente na boot metodi vašeg davatelja usluga:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * Pokrenite (engl. Bootstrap) usluge vašeg paketa.
 */
```

```
public function boot(): void
{
    Blade::component('package-alert', AlertComponent::class);
}
```

Nakon što je vaša komponenta registrirana, može prikazivati koristeći svoj tag alias:

```
<x-package-alert/>
```

#### *Automatsko učitavanje komponenti paketa*

Alternativno, možete koristiti `componentNamespace` metodu za automatsko učitavanje komponenti klasa prema konvenciji. Na primjer, `Nightshade` paket može imati `Calendar` i `ColorPicker` komponente koje se nalaze unutar `Package\Views\Components` prostora imena:

```
use Illuminate\Support\Facades\Blade;

/**
 * Pokrenite (engl. Bootstrap) usluge vašeg paketa.
 */
public function boot(): void
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

Ovo će omogućiti korištenje komponenti paketa njihovog isporučitelja prostora imena koristeći `package-name::` sintaksu:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade će automatski detektirati klasu koja je povezana s ovom komponentom s pascal-casting imenom komponente. Poddirektoriji su također podržani korištenjem „točka“ notacije.

#### *Anonimne komponente*

Slično ugrađenim komponentama, anonimne komponente pružaju mehanizam za upravljanje komponentom pomoću jedne datoteke. Međutim, anonimne komponente koriste jednu datoteku prikaza i nemaju pridruženu klasu. Da biste definirali anonimnu komponentu, trebate samo postaviti Blade predložak u svoj `resources/views/components` direktorij. Na primjer, pod pretpostavkom da ste definirali komponentu na `resources/views/components/alert.blade.php`, možete je jednostavno prikazati ovako:

```
<x-alert/>
```

Možete koristiti `.` znak da označite je li komponenta ugniježđena dublje unutar `components` direktorija. Na primjer, pod pretpostavkom da je komponenta definirana na `resources/views/components/inputs/button.blade.php`, možete je prikazati ovako:

```
<x-inputs.button/>
```

### Anonimne komponente indeksa

Ponekad, kada je komponenta sastavljena od više Blade predložaka, možda ćete htjeti grupirati dane predložke komponente unutar jednog direktorija. Na primjer, zamislite komponentu „harmonika<sup>44</sup>“ sa sljedećom strukturom direktorija:

```
/resources/views/components/accordion.blade.php
/resources/views/components/accordion/item.blade.php
```

Ova struktura direktorija omogućava vam da prikazete komponentu harmonike i njezinu stavku renderirate ovako:

```
<x-accordion>
  <x-accordion.item>
    ...
  </x-accordion.item>
</x-accordion>
```



Međutim, kako bismo prikazali komponentu harmonike pomoću `x-accordion`, bili smo prisiljeni smjestiti „indeks“ predložak komponente harmonike u direktorij `resources/views/components` umjesto da ga ugniježdimo unutar `accordion` direktorija s drugim predlošcima koji se odnose na harmoniku.

Srećom, Blade vam omogućava da postavite `index.blade.php` datoteku unutar direktorija predložaka komponente. Kada postoji `index.blade.php` predložak za komponentu, on će se prikazati kao „root“ čvor komponente. Dakle, možemo nastaviti koristiti istu Blade sintaksu danu u gornjem primjeru; međutim, prilagodit ćemo našu strukturu imenika ovako:

```
/resources/views/components/accordion/index.blade.php
/resources/views/components/accordion/item.blade.php
```

### Svojstva podataka/atributi

Budući da anonimne komponente nemaju nikakvu pridruženu klasu, možete se zapitati kako možete razlikovati koji podaci trebaju biti proslijeđeni komponenti kao varijable i koji atributi trebaju biti smješteni u [vreću s atributima](#) komponente.

Možete odrediti koje attribute treba smatrati podatkovnim varijablama pomoću `@props` direktive na vrhu Blade predloška vaše komponente. Svi ostali atributi na komponenti bit će dostupni kroz vreću

<sup>44</sup> Harmonika je grafički kontrolni element koji se sastoji od okomito naslaganog popisa stavki, kao što su oznake ili sličice. Svaka stavka može se „proširiti“ ili „skupiti“ kako bi se otkrio sadržaj povezan s tom stavkom. Moguće je da nijedna stavka ne bude proširena ili jedna ili više stavki mogu biti proširene odjednom. Ovaj meni dobio je ime po harmonici kod koje se dijelovi mijeha mogu proširiti povlačenjem prema van.

atributa komponente. Ako varijabli podataka želite dati zadanu vrijednost, možete navesti naziv varijable kao ključ polja i zadanu vrijednost kao vrijednost polja:

```
<!-- /resources/views/components/alert.blade.php -->

@props(['type' => 'info', 'message'])

<div {{ $attributes->merge(['class' => 'alert alert-'. $type]) }}>
    {{ $message }}
</div>
```

S obzirom na gornju definiciju komponente, komponentu možemo prikazati ovako:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

### *Pristup roditeljskim (nadređenim) podacima*

Ponekad ćete možda htjeti pristupiti podacima iz nadređene komponente unutar podređene komponente. U tim slučajevima možete koristiti `@aware` direktivu. Na primjer, zamislite da gradimo složenu komponentu menija koja se sastoji od roditelja `<x-menu>` i djeteta `<x-menu.item>`:

```
<x-menu color="purple">
    <x-menu.item>...</x-menu.item>
    <x-menu.item>...</x-menu.item>
</x-menu>
```

`<x-menu>` komponenta može imati sljedeću implementaciju:

```
<!-- /resources/views/components/menu/index.blade.php -->

@props(['color' => 'gray'])

<ul {{ $attributes->merge(['class' => 'bg-'. $color. '-200']) }}>
    {{ $slot }}
</ul>
```

Budući da `color` je prop proslijeđen samo nadređenom (`<x-menu>`), neće biti dostupan unutar `<x-menu.item>`. Međutim, ako koristimo `@aware` direktivu, možemo je učiniti dostupnom `<x-menu.item>` i unutar:

```
<!-- /resources/views/components/menu/item.blade.php -->

@aware(['color' => 'gray'])
```

```
<li {{ $attributes->merge(['class' => 'text-'. $color.'-800']) }}>
    {{ $slot }}
</li>
```

**UPOZORENJE:**

`@aware` direktiva ne može pristupiti nadređenim podacima koji nisu izričito proslijeđeni nadređenoj komponenti pomoću HTML atributa. Zadane `@props` vrijednosti koje nisu direktno proslijeđene nadređenoj komponenti ne mogu pristupiti `@aware` direktivi.

*Anonimne putanje komponenti*

Kao što je prethodno objašnjeno, anonimne komponente obično se definiraju postavljanjem Blade predloška u vaš `resources/views/components` direktorij. Međutim, možda ćete povremeno htjeti registrirati druge anonimne putanje komponenti s Laravelom uz zadanu putanju.

`anonymousComponentPath` metoda prihvaća „putanju“ do lokacije anonimne komponente kao svoj prvi argument i opcionalni „prostor imena“ pod koji komponente trebaju biti smještene kao drugi argument. Obično se ova metoda treba pozvati iz `boot` metode jednog od [pružatelja usluga](#) vaše aplikacije:

```
/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Blade::anonymousComponentPath(__DIR__.'/../components');
}
```

Kada su staze komponenti registrirane bez navedenog prefiksa kao u gornjem primjeru, mogu se prikazati u vašim Blade komponentama i bez odgovarajućeg prefiksa. Na primjer, ako `panel.blade.php` komponenta postoji na gore registriranoj putanji, može se prikazati ovako:

```
<x-panel />
```

Prefiks „prostora imena“ može se navesti kao drugi argument `anonymousComponentPath` metode:

```
Blade::anonymousComponentPath(__DIR__.'/../components', 'dashboard');
```

Kada je naveden prefiks, komponente unutar tog „prostora imena“ mogu se prikazati dodavanjem prefiksa u prostor imena komponente nazivu komponente kada se komponenta prikazuje:

```
<x-dashboard::panel />
```



## Izgradnja plana pozicija

### *Plan pozicija koji koriste komponente*

Većina web aplikacija održava isti opći plan pozicija (engl. layout) na različitim stranicama. Bilo bi nevjerovatno glomazno i teško održavati našu aplikaciju ako bismo morali ponavljati cijeli HTML raspored u svakom pogledu koji stvorimo. Srećom, zgodno je definirati ovaj plan pozicija kao jednu Blade komponentu i zatim ga koristiti u cijeloj našoj aplikaciji.

### *Definiranje komponenti plana pozicija*

Na primjer, zamislite da gradimo aplikaciju s listom zadataka („todo“ lista). Mogli bismo definirati `layout` komponentu koja izgleda ovako:

```
<!-- resources/views/components/layout.blade.php -->

<html>
  <head>
    <title>{{ $title ?? 'Todo Manager' }}</title>
  </head>
  <body>
    <h1>Todos</h1>
    <hr/>
    {{ $slot }}
  </body>
</html>
```

### *Primjena plana pozicija komponenti*

Nakon što je `layout` komponenta definirana, možemo stvoriti Blade pogled koji koristi komponentu. U ovom primjeru definirat ćemo jednostavan pogled koji prikazuje naša lista zadataka:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
  @foreach ($tasks as $task)
    {{ $task }}
  @endforeach
</x-layout>
```

Zapamtite, sadržaj koji je injektiran u komponentu bit će isporučen zadanoj `$slot` varijabli unutar naše `layout` komponente. Kao što ste možda primijetili, naš `layout` također poštuje `$title` slot ako je osiguran; inače se prikazuje zadani naslov. Možemo ubaciti prilagođeni naslov iz prikaza liste zadataka koristeći standardnu sintaksu slot-a o kojoj se govori u dokumentaciji komponente:

```
<!-- resources/views/tasks.blade.php -->
```

```
<x-layout>
  <x-slot:title>
    Custom Title
  </x-slot>

  @foreach ($tasks as $task)
    {{ $task }}
  @endforeach
</x-layout>
```

Sada kada smo definirali naš plan pozicija i listu zadataka, samo trebamo vratiti `tasks` pogled rute:

```
use App\Models\Task;

Route::get('/tasks', function () {
    return view('tasks', ['tasks' => Task::all()]);
});
```

*Plan pozicija koji koriste nasljeđivanje predložaka*

*Definiranje plana pozicija*

Plan pozicija (engl. Layouts) se također može stvoriti pomoću „nasljeđivanja predložaka“ (engl. „template inheritance“). Ovo je bio primarni način izgradnje aplikacija prije uvođenja komponenti.

Za početak, pogledajmo jednostavan primjer. Prvo ćemo ispitati plan pozicija stranice. Budući da većina web aplikacija održava isti opći plan pozicija na različitim stranicama, zgodno je definirati ovaj plan pozicija kao jedan Blade pogled:

```
<!-- resources/views/layouts/app.blade.php -->

<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      Ovo je glavna bočna traka.
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

Kao što vidite, ova datoteka sadrži tipične HTML oznake (engl. mark-up). Međutim, obratite pažnju na `@section` i `@yield` direktive. `@section` direktiva, kao što naziv implicira, definira dio sadržaja, dok `@yield` direktiva koristi za prikaz sadržaja danog bloka.

Sada kada smo definirali plan pozicija za našu aplikaciju, definirajmo podređenu stranicu koja nasljeđuje plan pozicija.

#### Proširivanje plana pozicija

Kada definirate podređeni pogled, koristite `@extends` Blade direktivu da odredite koji plan pozicija podređeni pogled treba „naslijediti“. Pogledi koji proširuju Blade plan pozicija mogu injektirati sadržaj u blokove plana pozicija pomoću `@section` direktiva. Upamtite, kao što se vidi u gornjem primjeru, sadržaj ovih blokova bit će prikazan u pogledu pomoću `@yield`:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>Ovo je dodano glavnoj bočnoj traci.</p>
@endsection

@section('content')
    <p> Ovo je sadržaj mog tijela.</p>
@endsection
```

U ovom primjeru `sidebar` blok koristi `@parent` direktivu za dodavanje (umjesto prepisivanja preko) sadržaja na bočnu traku plana pozicija. `@parent` direktiva će biti zamijenjena sadržajem plana pozicija kada se pogled prikaže.

#### NAPOMENA:



Suprotno prethodnom primjeru, ovaj `sidebar` blok završava s `@endsection` umjesto `@show`. `@endsection` direktiva će samo definirati blok, dok će `@show` definirati i odmah proizvesti blok.

`@yield` direktiva također prihvaća zadanu vrijednost kao drugi parametar. Ova vrijednost će se prikazati ako je blok koji se daje nedefiniran:

```
@yield('content', 'Default content')
```

## Forme (Obrasci)

### CSRF polje

Svaki put kada definirate HTML formu u svojoj aplikaciji, trebali biste uključiti polje skrivenog CSRF tokena u formu kako bi [CSRF zaštita](#) middleware-a mogla ovjeriti zahtjev. Možete koristiti `@csrf` Blade direktivu za generiranje polja tokena:

```
<form method="POST" action="/profile">
    @csrf

    ...
</form>
```

### method polje

Budući da HTML forme ne mogu postavljati `PUT`, `PATCH`, ili `DELETE` zahtjeve, morat ćete dodati skriveno `_method` polje za lažiranje ovih HTTP glagola. `@method` Blade direktiva može kreirati ovo polje za vas:

```
<form action="/foo/bar" method="POST">
    @method('PUT')

    ...
</form>
```

### Greške provjere ispravnosti (validacije)

`@error` direktiva se može koristiti za brzu provjeru postoje li poruke o grešci validnosti za dati atribut. Unutar `@error` direktive, možete prikazati tekst proslijeđen kao argument (engl.echo) `$message` varijabli za prikaz poruke o grešci:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Naslov objave</label>

<input id="title"
    type="text"
    class="@error('title') je nevažeći @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Budući da se `@error` direktiva kompajlira u „if“ naredbu, možete upotrijebiti `@else` direktivu za prikaz sadržaja kada nema greške za atribut:

```
<!-- /resources/views/auth.blade.php -->
```

```
<label for="email">Email address</label>

<input id="email"
      type="email"
      class="@error('email') is-invalid @else is-valid @enderror">
```

Možete proslijediti naziv određene torbe s greškama kao drugi parametar `@error` direktive za dohvaćanje poruka o grešci provjere ispravnosti na stranicama koje sadrže više formi:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email"
      type="email"
      class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

### Stekovi

Blade vam omogućava guranje na imenovane hrpe koje se mogu prikazati negdje drugdje u drugom prikazu ili rasporedu. Ovo može biti osobito korisno za određivanje bilo koje JavaScript biblioteke potrebne vašim podređenim (engl. child) pogledima:

```
@push('scripts')
    <script src="/primjer.js"></script>
@endpush
```

Ako bi željeli `@push` sadržaj ako dani Booleov izraz ima vrijednost true, možete koristiti `@pushIf` direktivu:

```
@pushIf($shouldPush, 'scripts')
    <script src="/primjer.js"></script>
@endPushIf
```

Možete gurati na stek onoliko puta koliko je potrebno. Da biste prikazali cijeli sadržaj steka, proslijedite naziv steka `@stack` direktivi:

```
@push('scripts')
    Ovo će biti drugi...
@endpush
```

```
// Kasnije...

@prepend('scripts')
    Ovo će biti prvi...
@endprepend
```

### Injektiranje usluge

`@inject` direktiva se može koristiti za dohvaćanje usluge iz Laravel spremnika usluge . Prvi argument koji se proslijeđuje `@inject`je naziv varijable u koju će se usluga smjestiti, dok je drugi argument naziv klase ili interface usluge koju želite riješiti:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

### Prikazivanje ugrađenih Blade predložaka

Ponekad ćete možda morati transformirati neobrađeni string Blade predloška u ispravan HTML. To možete postići pomoću `render` metode koju nudi `Blade` fasada. `render` metoda prihvaća string Blade predložak i opcionalnu matricu podataka koje treba dati predlošku:

```
use Illuminate\Support\Facades\Blade;

return Blade::render('Hello, {{ $name }}', ['name' => 'Julian Bashir']);
```

Laravel renderira ugrađene Blade predloške upisujući ih u `storage/framework/views` direktorij. Ako želite da Laravel ukloni ove privremene datoteke nakon prikazivanja Blade predloška, možete dati argument `deleteCachedView` za metodu:

```
return Blade::render(
    'Hello, {{ $name }}',
    ['name' => 'Julian Bashir'],
    deleteCachedView: true
);
```

### Prikazivanje Blade fragmenata

Kada koristite frontend okvire kao što su [Turbo](#) i [htmx](#), možda ćete povremeno trebati vratiti samo dio Blade predloška unutar vašeg HTTP odgovora. Blade „fragmenti“ vam omogućavaju upravo to. Za početak postavite dio svog Blade predloška unutar `@fragment` i `@endfragment` direktiva:

```
@fragment('user-list')
```

```

    <ul>
        @foreach ($users as $user)
            <li>{{ $user->name }}</li>
        @endforeach
    </ul>
@endfragment

```

Zatim, kada prikažete pogled koji koristi ovaj predložak, možete pozvati `fragment` metodu da odredite da samo navedeni fragment treba biti uključen u odlazni HTTP odgovor:

```
return view('dashboard', ['users' => $users])->fragment('user-list');
```

`fragmentIf` metoda vam omogućava uslovno vraćanje fragmenta pogleda na temelju zadanog uslova. U suprotnom će se vratiti cijeli pogled:

```
return view('dashboard', ['users' => $users])
    ->fragmentIf($request->hasHeader('HX-Request'), 'user-list');
```

`fragments` i `fragmentsIf` metode omogućavaju vam vraćanje više fragmenata pogleda u odgovoru. Fragmenti će biti spojeni zajedno:

```
view('dashboard', ['users' => $users])
    ->fragments(['user-list', 'comment-list']);

view('dashboard', ['users' => $users])
    ->fragmentsIf(
        $request->hasHeader('HX-Request'),
        ['user-list', 'comment-list']
    );

```

## Produživanje Blade-a

Blade vam omogućava definiranje vlastitih prilagođenih direktiva pomoću `directive` metode. Kada Blade kompajler naiđe na prilagođenu direktivu, pozvat će ponuđeni argument drugoj funkciji (engl. callback) s izrazom koji sadrži direktiva.

Sljedeći primjer kreira `@datetime($var)` direktivu koja oblikuje dani `$var`, koji bi trebao biti instanca `DateTime`:

```

<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

```

```

class AppServiceProvider extends ServiceProvider
{
    /**
     * Registriraj bilo koju aplikacijsku uslugu.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Pokreni (engl. Bootstrap ) bilo koju aplikacijsku uslugu.
     */
    public function boot(): void
    {
        Blade::directive('datetime', function (string $expression) {
            return "<?php echo ($expression)->format('m.d.Y H:i'); ?>";
        });
    }
}

```

Kao što vidite, ulančat ćemo `format` metodu na bilo koji izraz koji je proslijeđen u direktivu. Dakle, u ovom primjeru, konačni PHP generiran ovom direktivom bit će:

```
<?php echo ($var)->format('m.d.Y H:i'); ?>
```

#### UPOZORENJE:



Nakon ažuriranja logike Blade direktive, morat ćete izbrisati sve keširane Blade poglede. Keširani Blade pogledi mogu se ukloniti pomoću Artisan naredbe `view:clear`.

#### Prilagođeni echo rukovatelji

Ako promatrae „echo“ objekt (op. prev. dakle onaj kojem je tekst proslijeđen kao argument) koristeći Blade, bit će pozvana `__toString` metoda objekta. Metoda `__toString` je jedna od PHP-ovih ugrađenih „čarobnih metoda“. Međutim, ponekad možda nemate kontrolu nad `__toString` metodom dane klase, na primjer kada klasa s kojom ste u interakciji pripada nekoj trećoj biblioteci.

U tim slučajevima Blade vam omogućava registraciju prilagođenog echo rukovatelja za tu određenu vrstu objekta. Da biste to postigli, trebali biste pozvati Bladeovu `stringable` metodu. Metoda `stringable` prihvaća anonimnu funkciju (engl. closure). Ova anonimna funkcija (engl. closure) trebala bi nagovijestiti tip (engl. type-hint) objekta koji je pouzdan za prikazivanje. U pravilu, `stringable` metodu treba pozvati unutar `boot` metode `AppServiceProvider` klase vaše aplikacije:



```

use Illuminate\Support\Facades\Blade;
use Money\Money;

/**
 * Pokrenite (engl. Bootstrap ) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    Blade::stringable(function (Money $money) {
        return $money->formatTo('hr_HR');
    });
}

```

Nakon što je vaš prilagođeni echo rukovatelj definiran, možete jednostavno proslijediti tekst kao argument (engl.echo) objektu u vašem Blade predlošku:

```
Cost: {{ $money }}
```

### Prilagođene If izjave

Programiranje prilagođene direktive ponekad je složenije nego što je potrebno kada se definiraju jednostavni, prilagođeni uslovni iskazi. Iz tog razloga, Blade nudi `Blade::if` metodu koja vam omogućava brzo definiranje prilagođenih uslovnih direktiva pomoću zatvaranja. Na primjer, definirajmo prilagođeni uslov koji provjerava konfigurirani zadani „disk“ za aplikaciju. To možemo učiniti `boot` metodom našim `AppServiceProvider`:

```

use Illuminate\Support\Facades\Blade;

/**
 * * Pokrenite (engl. Bootstrap ) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    Blade::if('disk', function (string $value) {
        return config('filesystems.default') === $value;
    });
}

```

Nakon što definirate prilagođeni uslov, možete ga koristiti unutar svojih predložaka:

```

@disk('local')
    <!-- Aplikacija koristi lokalni disk... -->
@elsedisk('s3')
    <!-- Aplikacija koristi s3 disk... -->

```

```
@else
    <!-- Aplikacija koristi neki drugi disk... -->
@enddisk

@unlessdisk('local')
    <!-- Aplikacija ne koristi lokalni disk... -->
@enddisk
```

## Paket materijala (Vite)

### Uvod

Vite je moderan frontend koji pruža iznimno brzo razvojno okruženje i pakuje (engl. bundle) tj. vaš kod za produkciju. Kada gradite aplikacije s Laravelom, obično ćete koristiti Vite za spajanje CSS i JavaScript datoteka vaše aplikacije u materijale (engl. assets) spremne za produkciju.

Laravel se neprimjetno integrira s Viteom pružajući službeni dodatak (engl. plugin) i Blade direktivu za učitavanje vaših materijala za razvoj i produkciju.

#### BILJEŠKA:



Koristite li Laravel Mix? Vite je zamijenio Laravel Mix u novim Laravel instalacijama. Za Mix dokumentaciju, molimo posjetite [Laravel Mix](#) web stranicu. Ako se želite prebaciti na Vite, pogledajte [vodič za migraciju](#).

### Izbor između Vitea i Laravel Mixa

Prije prelaska na Vite, nove Laravel aplikacije koristile su [Mix](#), koji pokreće [webpack](#), kada pakiramo materijale (engl. bundling assets). Vite se fokusira na pružanje bržeg i produktivnijeg iskustva kod izrade bogatih JavaScript aplikacija. Ako razvijate Aplikacije s jednom stranicom (engl. Single Page Application - SPA), uključujući one razvijene pomoću alata kao što je Inertia, Vite će savršeno odgovarati.

Vite također dobro radi s tradicionalnim aplikacijama prikazanim na strani servera s JavaScript „sprinklesima“, uključujući one koje koriste Livewire. Međutim, nedostaju mu neka svojstva koje Laravel Mix podržava, kao što je mogućnost kopiranja proizvoljnih materijala u build (obično označavamo verzija.podverzija.build) koji se ne poziva direktno u vašoj JavaScript aplikaciji.

### Migracija nazad na Mix

Jeste li pokrenuli novu Laravel aplikaciju pomoću naše Vite skele, ali se trebate vratiti na Laravel Mix i webpack? Nema problema. Pogledajte naš [službeni vodič o prelasku s Vite na Mix](#).

## Instalacija i podešavanje

#### NAPOMENA:



Sljedeća dokumentacija opisuje kako ručno instalirati i konfigurirati Laravel Vite dodatak. Međutim, Laravel [kompleti za početnike \(engl. starter kits\)](#) već uključuju sve ove skele i najbrži su način da počnete s Laravel-om i Vite-om.

### Instaliranje čvora

Morate osigurati da su Node.js (16+) i NPM instalirani prije nego pokrenete Vite i Laravel plugin:

```
node -v
npm -v
```

Najnoviju verziju Nodea i NPM-a možete jednostavno instalirati pomoću jednostavnih grafičkih programa za instalaciju sa [službene Node web stranice](#). Ili, ako koristite Laravel Sail, možete pozvati Node i NPM kroz Sail:

```
./vendor/bin/sail node -v  
./vendor/bin/sail npm -v
```

### *Instaliranje Vite i Laravel plugin-a*

Unutar nove instalacije Laravela, pronaći ćete `package.json` datoteku u početnom direktoriju strukture direktorija vaše aplikacije. Zadana `package.json` datoteka već uključuje sve što vam je potrebno da počnete koristiti Vite i Laravel plugin. Možete instalirati frontend zavisnosti svoje aplikacije pomoću NPM-a:

```
npm install
```

### *Konfiguriranje Vite*

Vite se konfiguriše pomoću datoteke `vite.config.js` u početnom direktoriju vašeg projekta. Možete slobodno prilagoditi ovu datoteku prema svojim potrebama, a također možete instalirati sve druge dodatke koje vaša aplikacija zahtijeva, kao što su `@vitejs/plugin-vue` ili `@vitejs/plugin-react`.

Laravel Vite plugin zahtijeva da navedete ulazne točke za svoju aplikaciju. To mogu biti JavaScript ili CSS datoteke i uključuju pretprocesirane jezike kao što su TypeScript, JSX, TSX i Sass.

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
  
export default defineConfig({  
  plugins: [  
    laravel([  
      'resources/css/app.css',  
      'resources/js/app.js',  
    ]),  
  ],  
});
```

Ako gradite SPA, uključujući aplikacije izgrađene pomoću Inertia, Vite najbolje radi bez CSS ulaznih točaka:

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
  
export default defineConfig({  
  plugins: [  
    laravel([  
      'resources/css/app.css',  
      'resources/js/app.js',  
    ]),  
  ],  
});
```

```
});
```

Umjesto toga, trebali biste uvesti svoj CSS pomoću JavaScripta. Obično se to radi u datoteci vaše aplikacije `resources/js/app.js`:

```
import './bootstrap';  
import '../css/app.css';
```

Laravel plugin također podržava višestruke ulazne točke i napredne konfiguracijske opcije kao što su SSR ulazne točke .

#### *Rad sa sigurnim razvojnim serverom*

Ako vaš lokalni razvojni web server posluhuje vašu aplikaciju pomoću HTTPS, mogli biste naići na probleme kada povezujete s razvojn timerom Vite.

Ako koristite Laravel Herd i imate siguran site ili koristite Laravel Valet i ste pokrenuli sigurnosnu naredbu za svoju aplikaciju, Laravel Vite plugin će automatski otkriti i koristiti generirani TLS certifikat umjesto vas.

Ako ste osigurali svoj site koristeći host koji ne odgovara nazivu direktorija aplikacije, možete ručno navesti host u `vite.config.js` datoteci svoje aplikacije:

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
  
export default defineConfig({  
  plugins: [  
    laravel({  
      // ...  
      detectTls: 'my-app.test',  
    }),  
  ],  
});
```

Kada koristite drugi web server, trebali biste generirati pouzdani certifikat i ručno konfigurirati Vite za korištenje generiranih certifikata:

```
// ...  
import fs from 'fs';  
  
const host = 'my-app.test';  
  
export default defineConfig({  
  // ...  
  server: {
```

```
    host,  
    hmr: { host },  
    https: {  
      key: fs.readFileSync(`/path/to/${host}.key`),  
      cert: fs.readFileSync(`/path/to/${host}.cert`),  
    },  
  },  
});
```

Ako ne možete generirati pouzdani certifikat za svoj sistem, možete instalirati i konfigurirati [@vitejs/plugin-basic-ssl](#) [plugin](#). Kada koristite nepouz dane certifikate, morat ćete prihvatiti upozorenje o certifikatu za Viteov razvojni server u vašem browseru slijedeći „lokalni“ link na vašoj konzoli kada pokrećete `npm run dev` naredbu.

#### *Pokretanje razvojnog servera u Sailu na WSL2*

Kada pokrećete Vite razvojni server unutar Laravel Sail na Windows podsistemu za Linux 2 (WSL2), trebali biste dodati sljedeću konfiguraciju svojoj `vite.config.js` datoteci kako biste osigurali da browser može komunicirati s razvojnim serverom:

```
// ...  
  
export default defineConfig({  
  // ...  
  server: {  
    hmr: {  
      host: 'localhost',  
    },  
  },  
});
```

Ako se promjene vaše datoteke ne odražavaju u browseru dok razvojni server radi, možda ćete morati konfigurirati Vite [opciju](#) `server.watch.usePolling`.

#### *Učitavanje vaših skripti i stilova*

S konfiguriranim vašim Vite ulaznim točkama, sada ih možete referencirati u `@vite()` Blade direktivi koju dodajete u `<head>` predloška početnog direktorija vaše aplikacije:

```
<!doctype html>  
<head>  
  {{-- ... --}}  
  
  @vite(['resources/css/app.css', 'resources/js/app.js'])  
</head>
```

Ako uvozite svoj CSS pomoću JavaScripta, trebate uključiti samo JavaScript ulaznu točku:

```
<!doctype html>
<head>
  {{-- ... --}}

  @vite('resources/js/app.js')
</head>
```

`@vite` direktiva će automatski detektirati Vite razvojni server i ubaciti Vite klijent kako bi se omogućila Hot Module Replacement. U build modu, direktiva će učitati vaše kompajlirane i verzionirane<sup>45</sup> materijale, uključujući bilo koji uvezeni CSS.

Ako je potrebno, također možete navesti putanju izgradnje vaših kompajliranih materijala (engl. assets) kod pozivanja `@vite` direktive:

```
<!doctype html>
<head>
  {{-- Zadana build putanja relativna je u odnosu na javnu putanju. --}}

  @vite('resources/js/app.js', 'vendor/courier/build')
</head>
```

### *Umetnuti materijali*

Ponekad može biti potrebno uključiti sirovi sadržaj materijala umjesto povezivanja na verzionirani URL materijal. Na primjer, možda ćete morati uključiti sadržaj materijal direktno na svoju stranicu kada prosljeđujete HTML sadržaj PDF generatoru. Možete ispisati sadržaj Vite materijala pomoću `content` metode koju nudi `Vite` fasada:

```
@php
use Illuminate\Support\Facades\Vite;
@endphp

<!doctype html>
<head>
  {{-- ... --}}

  <style>
    {!! Vite::content('resources/css/app.css') !!}
  </style>
```

---

<sup>45</sup> Upravljanje izvornim kôdom (često se susreće i naziv verzioniranje izvornog kôda) softverska je procedura kod koje se izvorni kôd čuva u centralnoj bazi podataka odnosno spremištu (verzije izvornog kôda mogu biti uskladištene u bazi podataka odnosno u datotečni sistem) u kojemu se bilježi svaka promjena kôda kroz povijest. Ako se u sistem upravljanja izvornim kôdom uključi i programska dokumentacija, dobiva se sistem upravljanja softverskom konfiguracijom, koji predstavlja upravljanje promjenama dokumentacije, računalnih programa, velikih internet stranica i drugih zbirki informacija.

```
<script>
  {!! Vite::content('resources/js/app.js') !!}
</script>
</head>
```

### Pokretanje Vite

Postoje dva načina na koje možete pokrenuti Vite. Možete pokrenuti razvojni server pomoću `dev` naredbe, što je korisno za lokalni razvoj. Razvojni server će automatski otkriti promjene na vašim datotekama i odmah ih prikazati u svim otvorenim prozorima browsera.

Ili pokretanjem `build` naredbe će verzija i paket sredstava vaše aplikacije biti spremni za implementaciju u produkciji:

```
# Pokreni razvojni server Vite...
npm run dev

# Izgradi (Build) i verziraj materijal za produkciju...
npm run build
```

Ako koristite razvojni server u Sail na WSL2, možda će vam trebati neke [dodatne opcije konfiguracije](#).

### Rad s JavaScriptom

#### *Aliases*

Laravel plugin, prema zadanim postavkama, pruža zajednički pseudonim (alias) koji vam pomaže da počnete pokretati i jednostavno uvezete materijal svoje aplikacije:

```
{
  '@' => '/resources/js'
}
```

Možete prepisati `@` alias dodavanjem vlastitog u `vite.config.js` konfiguracijsku datoteku:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel(['resources/ts/app.tsx']),
  ],
  resolve: {
    alias: {
      '@': '/resources/ts',
    },
  },
});
```



```
});
```

### Vue

Ako želite izgraditi svoj interface pomoću Vue radne okoline, također ćete morati instalirati `@vitejs/plugin-vue` dodatak:

```
npm install --save-dev @vitejs/plugin-vue
```

Možete zatim uključiti dodatak u `vite.config.js` konfiguracijsku datoteku. Postoji nekoliko dodatnih opcija koje ćete trebati kada koristite Vue plugin s Laravel-om:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import vue from '@vitejs/plugin-vue';

export default defineConfig({
  plugins: [
    laravel(['resources/js/app.js']),
    vue({
      template: {
        transformAssetUrls: {
          // Vue plugin će ponovo prepisati materijal URL-ova, kada se
          // referenciraju u Single File Components, kako bi upućivali na
          // Laravel web server. Postavljanje na `null` omogućava Laravel
          // plugin-u da umjesto toga ponovo zapiše materijal (engl.
          // asset) URL-ova kako bi upućivali na Vite server.
          base: null,

          // Vue plugin će analizirati apsolutne URL-ove i tretirati ih
          // kao apsolutne putanje do datoteka na disku. Postavljanjem na
          // `false` ostavit će apsolutne URL-ove nepromijenjenima tako da
          // mogu referencirati materijale u javnom direktoriju kako se
          // očekuje.
          includeAbsolute: false,
        },
      },
    }),
  ],
});
```

### NAPOMENA:



Laravelovi [kompleti za početnike \(engl. starter kits\)](#) već uključuju odgovarajuću Laravel, Vue i Vite konfiguraciju. Provjerite Laravel Breeze za najbrži način da počnete koristiti Laravel, Vue i Vite.

### React

Ako želite izgraditi svoj frontend pomoću React radne okoline, također ćete morati instalirati `@vitejs/plugin-react` radne okoline:

```
npm install --save-dev @vitejs/plugin-react
```

Nakon toga možete uključiti u vašu `vite.config.js` konfiguracijsku datoteku:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [
    laravel(['resources/js/app.jsx']),
    react(),
  ],
});
```

Morat ćete osigurati da sve datoteke koje sadrže JSX imaju `.jsx` ili `.tsx` ekstenziju. Ne zaboravite ažurirati svoju ulaznu točku, ako je potrebno, kao što je [prikazano iznad](#).

Također ćete morati uključiti dodatnu `@viteReactRefresh` Blade direktivu uz svoju postojeću `@vite` direktivu.

```
@viteReactRefresh
@vite('resources/js/app.jsx')
```

`@viteReactRefresh` direktiva se mora pozvati prije `@vite` direktive.

#### NAPOMENA:



Laravelovi [paketi za početnike](#) već uključuju odgovarajuću Laravel, React i Vite konfiguraciju. Provjerite Laravel Breeze za najbrži način da počnete koristiti Laravel, React i Vite.

### Inertija

Laravel Vite plugin pruža praktičnu `resolvePageComponent` funkciju koja pomaže u rješavanju vaših stranica Inertia komponenti. Ispod je primjer pomoćnika koji se koristi s Vue 3; međutim, možete također koristiti funkciju u drugim radnim okolinama kao što je React:

```
import { createApp, h } from 'vue';
import { createInertiaApp } from '@inertiajs/vue3';
import { resolvePageComponent } from 'laravel-vite-plugin/inertia-helpers';

createInertiaApp({
```

```
    resolve: (name) => resolvePageComponent(`./Pages/${name}.vue`,
import.meta.glob('./Pages/**/*.vue')),
    setup({ el, App, props, plugin }) {
      return createApp({ render: () => h(App, props) })
        .use(plugin)
        .mount(el)
    },
  });
```

**NAPOMENA:**

Laravelovi [paketi za početnike](#) već uključuju odgovarajuću Laravel, React i Vite konfiguraciju. Provjerite Laravel Breeze za najbrži način da počnete koristiti Laravel, React i Vite.

*URL procesiranje*

Kada koristite Vite i referencirate materijale u HTML-u, CSS-u ili JS-u svoje aplikacije, potrebno je uzeti u obzir nekoliko upozorenja. Prvo, ako referencirate materijale s apsolutnom putanjom, Vite neće uključiti materijal u izgradnju (engl. build); zato biste trebali osigurati da je materijal dostupan u vašem javnom direktoriju.

Kada referencirate na relativne putanje materijala, trebali biste upamtiti da su putanje relativne u odnosu na datoteku u kojoj su navedeni. Svi materijali navedeni pomoću relativne putanje će Vite ponovo prepisati (engl. re-write), verzionirati i grupirati.

Razmotrite sljedeću strukturu projekta:

```
public/
  taylor.png
resources/
  js/
    Pages/
      Welcome.vue
  images/
    abigail.png
```

Sljedeći primjer pokazuje kako će Vite tretirati relativne i apsolutne URL-ove:

```
<!-- Ovim materijalom ne upravlja Vite i neće biti uključen u build -->


<!--Ove materijale će Vite ponovo prepisati (engl. re-write), verzionirati i
grupirati -->
```

```

```

### Rad s tablicama stilova (engl. Stylesheets)

Možete saznati više o Vite CSS podršci unutar [Vite dokumentacije](#). Ako koristite PostCSS plugin-e kao što je [Tailwind](#), možete stvoriti `postcss.config.js` datoteku u početnom direktoriju svog projekta i Vite će je automatski primijeniti:

```
export default {
  plugins: [
    tailwindcss: {},
    autoprefixer: {},
  ],
};
```

#### NAPOMENA:



Laravelovi [paketi za početnike](#) već uključuju odgovarajuću Tailwind, PostCSS i Vite konfiguraciju. Ili, ako želite koristiti Tailwind i Laravel bez korištenja jednog od naših početnih kompleta, pogledajte [Tailwindov vodič za instalaciju za Laravel](#).

### Rad s Blade-om i rutama

#### *Obrada statičkih materijala s Vite-om*

Kada referencirate materijal u vašem JavaScriptu ili CSS-u, Vite ih automatski obrađuje i verzionira. Osim toga, kada se izrađuju aplikacije temeljene na Bladeu, Vite također može obraditi i verzirati statička sredstva koja referencirate isključivo s Blade predlošcima.

Međutim, kako biste to postigli, Vite morate upoznati sa svojim sredstvima uvozom statičkih materijala u ulaznu točku aplikacije. Na primjer, ako želite obraditi i verzionirati sve slike pohranjene u `resources/images` i svih fontova pohranjenih u `resources/fonts`, trebali biste dodati sljedeće u `resources/js/app.js` ulaznu točku vaše aplikacije:

```
import.meta.glob([
  '../images/**',
  '../fonts/**',
]);
```

Ove materijale će sada obrađivati Vite kada se pokreće `npm run build`. Zatim možete referencirati ta materijale u Blade predlošcima koristeći `Vite::asset` metodu koja će vratiti verzionirani URL za dano sredstvo:

```

```

### Osvježavanje kod snimanja

Kada je vaša aplikacija izgrađena korištenjem tradicionalnog prikazivanja sa strane servera s Bladeom, Vite može poboljšati vaš radni tok razvoja automatskim osvježavanjem browser-a kada napravite promjene datoteka za pogled u svojoj aplikaciji. Za početak možete jednostavno navesti `refresh` opciju kao `true`.

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: true,
    }),
  ],
});
```

Kada je `refresh` opcija postavljena `true`, snimanje datoteka u sljedeće direktorije pokrenut će browser da izvrši osvježavanje cijele stranice dok pokrećete `npm run dev`:

```
- app/View/Components/**
- lang/**
- resources/lang/**
- resources/views/**
- routes/**
```

Gledanje `routes/**` direktorija korisno je ako koristite [Ziggy](#) za generiranje linkova ruta unutar frontend-a vaše aplikacije.

Ako ove zadane putanje ne odgovaraju vašim potrebama, možete odrediti vlastitu listu putanja koje želite promatrati:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: ['resources/views/**'],
    }),
  ],
});
```

Ispod haube, dodatak Laravel Vite koristi paket `vite-plugin-full-reload` koji nudi neke napredne opcije konfiguracije za fino podešavanje ponašanja ovog svojstva. Ako vam je potrebna ovaj nivo prilagođavanja, možete dati `config` definiciju:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      // ...
      refresh: [{
        paths: ['path/to/watch/**'],
        config: { delay: 300 }
      }],
    }),
  ],
});
```

### *Aliasi*

Uobičajeno je u JavaScript aplikacijama [kreirati aliase](#) za redovito referenciranje direktorija. No, također možete stvoriti aliase za korištenje u Bladeu pomoću `macro` metode u `Illuminate\Support\Facades\Vite` klasi. U pravilu, „makroe“ treba definirati unutar `boot` metode [pružatelja usluge](#):

```
/**
 * Pokrenite (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    Vite::macro('image', fn (string $asset) => $this->asset("resources/images/{$asset}"));
}
```

Nakon što je makro definiran, može se pozvati unutar vaših predložaka. Na primjer, možemo koristiti `image` iznad definiranu makronaredbu za referenciranje na materijal koje se nalazi u `resources/images/logo.png`:

```

```

Ranije dohvaćanje materijala (engl. Asset Prefetching)<sup>46</sup>

Prilikom izgradnje SPA-a korištenjem Vite-ova svojstva dijeljenja koda, potrebni materijali se dohvaćaju kod svake navigacije stranice. Ovo ponašanje može dovesti do kašnjenja prikazivanja korisničkog interface-a. Ako je to problem za vašu odabranu frontend radnu okolinu, Laravel nudi mogućnost ranijeg dohvaćanja JavaScript i CSS materijala vaše aplikacije kod početnog učitavanja stranice.

Možete uputiti Laravel da revnosno unaprijed dohvati vaše materijale pozivanjem `Vite::prefetch` metode u boot metodi davatelja usluga:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Vite;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Registriraj bilo koju aplikacijsku uslugu.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
     */
    public function boot(): void
    {
        Vite::prefetch(concurrency: 3);
    }
}
```

U gornjem primjeru, materijali će se unaprijed dohvatiti s najviše 3 istodobna preuzimanja pri svakom učitavanju stranice. Možete izmijeniti istovremenost kako bi odgovarala potrebama vaše aplikacije ili odrediti bez ograničenja istovremenosti ako bi aplikacija trebala preuzeti sva sredstva odjednom:

---

<sup>46</sup> Prethodno dohvaćanje (engl. Asset Prefetching) omogućuje aplikacijama da maksimiziraju performanse i minimiziraju vrijeme čekanja unaprijed učitavanjem resursa koji će korisnicima biti potrebni prije nego što ih zatraže. Browseri koriste Asset Prefetching unaprijed učitavajući stranice kojima se često pristupa. Kada korisnik dođe do stranice, ona se brzo učitava jer je browser povlači iz keša. Neki operativni sistemi, kao što je Windows, spremaju datoteke u keš koje su programu potrebne pri pokretanju radi bržeg učitavanja.

```
/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    Vite::prefetch();
}
```

Prema zadanim postavkama, prethodno dohvaćanje će započeti kada se aktivira događaj učitavanja stranice. Ako želite prilagoditi početak prethodnog dohvaćanja, možete navesti događaj koji će Vite osluškivati:

```
/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    Vite::prefetch(event: 'vite:prefetch');
}
```

S obzirom na gornji kod, prethodno dohvaćanje sada će započeti kada ručno pošaljete događaj `vite:prefetch` na `window` objekt. Na primjer, mogli biste imati početak prethodnog dohvaćanja tri sekunde nakon učitavanja stranice:

```
<script>
    addEventListener('load', () => setTimeout(() => {
        dispatchEvent(new Event('vite:prefetch'))
    }, 3000))
</script>
```

#### Prilagođeni osnovni URL-ovi

Ako su vaši Vite kompajlirani materijali postavljeni na domenu odvojenu od vaše aplikacije, kao što je pomoću CDN<sup>47</sup>-a, morate navesti `ASSET_URL` varijablu okoline unutar vašeg aplikacijske `.env` datoteke:

```
ASSET_URL=https://cdn.primjer.com
```

---

<sup>47</sup> Mreža za isporuku sadržaja ili mreža za distribuciju sadržaja (engl. content delivery network - CDN) geografski je distribuirana mreža proxy servera i njihovih podatkovnih centara. Cilj je učitavanje web stranica za aplikacije koje opterećuju podatke. Primarna svrha CDN-a je smanjiti latenciju ili smanjiti kašnjenje u komunikaciji stvoreno dizajnom mreže. Zbog globalne i složene prirode interneta, komunikacijski promet između web stranica (servera) i njihovih korisnika (klijenata) mora se kretati preko velikih fizičkih udaljenosti. Komunikacija je također dvosmjerna, sa zahtjevima koji idu od klijenta do servera, a odgovori se vraćaju.



Nakon konfiguriranja URL-a materijala, svi ponovo napisani URL-ovi vaših materijala bit će prefiksirani konfiguriranom vrijednošću:

```
https://cdn.primjer.com/build/assets/app.9dce8d17.js
```

Imajte na umu da [apsolutni URL-ovi ne prepisuju Vite](#), tako da neće imati prefiks.

### Varijable okoline

Možete injektirati varijable okoline u svoj JavaScript tako da im dodate prefiks `VITE_` u `.env` datoteci vaše aplikacije:

```
VITE_SENTRY_DSN_PUBLIC=http://primjer.com
```

Injektiranim varijablama okoline možete pristupiti preko `import.meta.env` objekta:

```
import.meta.env.VITE_SENTRY_DSN_PUBLIC
```

### Onemogućavanje Vite-a u testovima

Laravel-ova Vite integracija pokušat će razriješiti vaše materijale tokom izvođenja testova, što zahtijeva da ili pokrenete Vite razvojni server ili izgradite svoje materijale.

Ako biste radije lažni (engl. mock<sup>48</sup>) Vite tokom testiranja, možete pozvati metodu `withoutVite` koja je dostupna za sve testove koji proširuju Laravel-ovu `TestCase` klasu:

```
use Tests\TestCase;

class ExampleTest extends TestCase
{
    public function test_without_vite_example(): void
    {
        $this->withoutVite();

        // ...
    }
}
```

Ako želite onemogućiti Vite za sve testove, možete pozvati metodu `withoutVite` iz `setUp` metode vaše osnovne `TestCase` klase:

```
<?php
```

---

<sup>48</sup> Osnovi postulat jediničnog testiranja je napraviti testirajuću klasu što je moguće više „lokalnom“. To znači da treba testirati samo objekt kojim se trenutno bavimo, a ne druge koji mu trebaju za njegov posao odnosno s kojima surađuje ili im prosljeđuje zadatke. Ovaj cilj maksimalne nezavisnosti je donekle moguće postići korištenjem, tzv. umjetnih ili lažnih objekata (engl. dummy, mock objects). Dakle, zamijenimo „pomoćne“ objekte objektima koji glume ono što nama treba.

```
namespace Tests;

use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;

    protected function setUp(): void
    {
        parent::setUp();

        $this->withoutVite();
    }
}
```

Prikazivanje na strani servera (engl. Server-Side Rendering - SSR)

Laravel Vite plugin olakšava podešavanje prikazivanja na server strani s Viteom. Da biste započeli, stvorite SSR ulaznu točku na `resources/js/ssr.js` i navedite ulaznu točku prosljeđivanjem konfiguracijske opcije Laravel pluginu:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      input: 'resources/js/app.js',
      ssr: 'resources/js/ssr.js',
    }),
  ],
});
```

Kako biste bili sigurni da nećete zaboraviti ponovno izgraditi SSR ulaznu točku, preporučujemo da proširite skriptu „build“ u svojoj aplikaciji `package.json` kako biste kreirali svoj SSR build:

```
"scripts": {
  "dev": "vite",
  "build": "vite build"
  "build": "vite build && vite build --ssr"
}
```

Zatim, za build i pokretanje SSR servera, možete pokrenuti sljedeće naredbe:

```
npm run build
node bootstrap/ssr/ssr.js
```

Ako koristite [SSR s Inertiom](#), umjesto toga možete koristiti `inertia:start-ssr` Artisan naredbu za pokretanje SSR servera:

```
php artisan inertia:start-ssr
```

#### NAPOMENA:



Laravelovi [paketi za početnike](#) već uključuju odgovarajuću Laravel, Inertia SSR i Vite konfiguraciju. Provjerite [Laravel Breeze](#) za najbrži način da počnete koristiti Laravel, Inertia SSR i Vite.

Skripta i atributi tagova stila

*Politika sigurnosti sadržaja (Content Security Policy - CSP) Nonce*

Ako želite uključiti `nonce` atribut u svoju skriptu i tagove stila kao dio svojih [Pravila sigurnosti sadržaja](#), možete generirati ili odrediti nonce koristeći metodu `useCspNonce` unutar prilagođenog middleware:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Vite;
use Symfony\Component\HttpFoundation\Response;

class AddContentSecurityPolicyHeaders
{
    /**
     * Rukovanje dolaznim zahtjevom.
     *
     * @param  \Closure(\Illuminate\Http\Request):
    (\Symfony\Component\HttpFoundation\Response) $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        Vite::useCspNonce();

        return $next($request)->withHeaders([
            'Content-Security-Policy' => "script-src 'nonce-".Vite::cspNonce()."',
        ]);
    }
}
```

```
}  
}
```

Nakon pozivanja metode `useCspNonce`, Laravel će automatski uključiti `nonce` atribut na svim generiranim oznakama skripte i stila.

Ako trebate navesti nonce negdje drugdje, uključujući direktivu `@route` Ziggy uključenu u Laravelove početne pakete, možete je dohvatiti pomoću `cspNonce` metode:

```
@routes(nonce: Vite::cspNonce())
```

Ako već imate `nonce` koji biste željeli naložiti Laravelu da ga koristi, možete proslijediti `nonce` metodi `useCspNonce`:

```
Vite::useCspNonce($nonce);
```

### *Integritet podresursa (SRI)*

Ako vaš Vite manifest (`.vite/manifest.json`) uključuje `integrity` hash-ove za vaše materijale, Laravel će automatski dodati `integrity` atribut svakoj skripti i stilskim tagovima koje generira kako bi se nametnuo [Integritet podresursa \(engl. Subresource Integrity - SRI\)](#). SRI je karakteristika koja browser-ima omogućava da li su li resursi koje dohvaćaju (na primjer, s [CDN-a](#)) isporučeni bez neočekivane manipulacije. Djeluje tako da vam omogućava da pružite kriptografski hash kojemu dohvaćeni resurs mora odgovarati.

Za provjeru integriteta podresursa koji se poslužuje iz originala koji nije dokument u koji je ugrađen, browser-i dodatno provjeravaju resurs koristeći [Dijeljenje resursa s različitim izvorima \(engl. Cross-Origin Resource Sharing - CORS\)](#) kako bi osigurali da original koji poslužuje resurs dopušta njegovo dijeljenje sa traženim originalom.

Prema zadanim postavkama, Vite ne uključuje `integrity` hash u svoj manifest, ali ga možete omogućiti instaliranjem `vite-plugin-manifest-sri` NPM plugin-a:

```
npm install --save-dev vite-plugin-manifest-sri
```

Tada možete omogućiti ovaj plugin u svojoj `vite.config.js` datoteci:

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
import manifestSRI from 'vite-plugin-manifest-sri';  
  
export default defineConfig({  
  plugins: [  
    laravel({  
      // ...  
    }),  
    manifestSRI(),  
  ],  
});
```

Ako je potrebno, također možete prilagoditi ključ manifesta gdje se može pronaći hash integriteta:

```
use Illuminate\Support\Facades\Vite;

Vite::useIntegrityKey('custom-integrity-key');
```

Ako želite potpuno isključiti ovo automatsko otkrivanje, možete poslati `false` `useIntegrityKey` metodi:

```
Vite::useIntegrityKey(false);
```

### *Proizvoljni atributi*

Ako trebate uključiti dodatne atribute u vašu skriptu i tagove stila, kao što je `data-turbo-track` atribut, možete ih odrediti pomoću metoda `useScriptTagAttributes` i `useStyleTagAttributes`. Obično se ove metode trebaju pozvati iz [pružatelja usluga](#):

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes([
    'data-turbo-track' => 'reload', // Navedi vrijednost za atribut...
    'async' => true, // Navedi atribut bez vrijednosti...
    'integrity' => false, // Isključi atribut koji bi inače bio uključen...
]);

Vite::useStyleTagAttributes([
    'data-turbo-track' => 'reload',
]);
```

Ako trebate uslovno dodati atribute, možete proslijediti argument drugoj funkciji (engl. callback) koji će primiti putanju izvora materijala, njegov URL, njegov dio manifesta i cijeli manifest:

```
use Illuminate\Support\Facades\Vite;

Vite::useScriptTagAttributes(fn (string $src, string $url, array|null $chunk,
array|null $manifest) => [
    'data-turbo-track' => $src === 'resources/js/app.js' ? 'reload' : false,
]);

Vite::useStyleTagAttributes(fn (string $src, string $url, array|null $chunk,
array|null $manifest) => [
    'data-turbo-track' => $chunk && $chunk['isEntry'] ? 'reload' : false,
]);
```

**UPOZORENJE:**

Argumenti `$chunk` i `$manifest` bit će `null` dok Vite razvojni server radi.

## Napredna prilagodba

Van okvira, Laravelov plugin Vite koristi razumne konvencije koje bi trebale funkcionirati za većinu aplikacija; međutim, ponekad ćete možda trebati prilagoditi Vite-ovo ponašanje. Kako bismo omogućili dodatne mogućnosti prilagodbe, nudimo sljedeće metode i opcije koje se mogu koristiti umjesto `@vite` Blade direktive:

```
<!doctype html>
<head>
    {{-- ... --}}

    {{
        Vite::useHotFile(storage_path('vite.hot')) // Prilagodi "vruću" datoteku...
        ->useBuildDirectory('bundle') // Prilagodi build direktorij...
        ->useManifestFilename('assets.json') // Prilagodi ime datoteke
        manifest-a...
        ->withEntryPoints(['resources/js/app.js']) // Navedi ulazne točke...
        ->createAssetPathsUsing(function (string $path, ?bool $secure) { //
        Prilagodi generiranje backend putanje za izgradnju materijala...
            return "https://cdn.primjer.com/{$path}";
        })
    }}
</head>
```

Unutar `vite.config.js` datoteke trebate navesti istu konfiguraciju:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            hotFile: 'storage/vite.hot', // Prilagodi "vruću" datoteku...
            buildDirectory: 'bundle', // Prilagodi build direktorij...
            input: ['resources/js/app.js'], // Navedi ulazne točke...
        }),
    ],
    build: {
        manifest: 'assets.json', // Prilagodi ime datoteke manifest-a...
```

```
},  
});
```

### *Ispravljanje URL-ova razvojnog servera*

Neki plugin-i unutar Vite ekosistema pretpostavljaju da će URL-ovi koji počinju kosom crtom uvijek upućivati na Vite razvojni server. Međutim, zbog prirode integracije Laravela, to nije slučaj.

Na primjer, `vite-imagemtools` plugin ispisuje URL-ove kao što su sljedeći dok Vite posluži vaše materijale:

```

```

`vite-imagemtools` plugin očekuje da će izlazni URL presresti Vite i plugin tada može obraditi sve URL-ove koji počinju s `/@imagemtools`. Ako koristite plugin-e koji očekuju ovakvo ponašanje, morat ćete ručno ispraviti URL-ove. To možete učiniti u svojoj `vite.config.js` datoteci pomoću `transformOnServe` opcije.

U ovom konkretnom primjeru, URL razvojnog servera ćemo dodati ispred svih pojavljivanja `/@imagemtools` unutar generiranog koda:

```
import { defineConfig } from 'vite';  
import laravel from 'laravel-vite-plugin';  
import { imagemtools } from 'vite-imagemtools';  
  
export default defineConfig({  
  plugins: [  
    laravel({  
      // ...  
      transformOnServe: (code, devServerUrl) =>  
code.replaceAll('/@imagemtools', devServerUrl+'/@imagemtools'),  
    },  
    imagemtools(),  
  ],  
});
```

Sada, dok Vite posluži Assets, izbacit će URL-ove koji upućuju na Vite razvojni server:

```
-   
+ 
```

## Generiranje URL-ova

### Uvod

Laravel nudi nekoliko pomoćnika (engl. helpers) koji će vam pomoći u generiranju URL-ova za vašu aplikaciju. Ovi pomoćnici prvenstveno su od pomoći kod izgradnje veza u vašim predlošcima i API odgovorima ili kod generiranja odgovora preusmjerenja na drugi dio vaše aplikacije.

### Osnove

#### *Generiranje URL-ova*

`url` pomoćnik (engl. helper) se može koristiti za generiranje proizvoljnih URL-ova za vašu aplikaciju. Generirani URL automatski će koristiti shemu (HTTP ili HTTPS) i host iz trenutnog zahtjeva koji obrađuje aplikacija:

```
$post = App\Models\Post::find(1);

echo url("/posts/{$post->id}");

// http://primjer.com/posts/1
```

Za generiranje URL-a s parametrima stringa upita, možete koristiti `query` metodu:

```
echo url()->query('/posts', ['search' => 'Laravel']);

// https://example.com/posts?search=Laravel

echo url()->query('/posts?sort=latest', ['search' => 'Laravel']);

// http://example.com/posts?sort=latest&search=Laravel
```

Pružanje parametara string upita koji već postoje u putanji prepisat će preko postojeće vrijednosti:

```
echo url()->query('/posts?sort=latest', ['sort' => 'oldest']);

// http://example.com/posts?sort=oldest
```

Matrica vrijednosti također se mogu proslijediti kao parametri upita. Ove vrijednosti bit će pravilno unesene i kodirane u generiranom URL-u:

```
echo $url = url()->query('/posts', ['columns' => ['title', 'body']]);

// http://example.com/posts?columns%5B0%5D=title&columns%5B1%5D=body

echo urldecode($url);

// http://example.com/posts?columns[0]=title&columns[1]=body
```



### *Pristupanje trenutnom URL-u*

Ako putanja nije omogućena `url` pomoćniku, `Illuminate\Routing\UrlGenerator` vraća se instanca koja vam omogućava pristupanje informacijama o trenutnom URL-u:

```
// Dohvaća trenutni URL-a bez stringa upita (engl. query)...
echo url()->current();

// Dohvaća trenutni URL uključujući string upita...
echo url()->full();

// Dohvaća puni URL za prethodni zahtjev...
echo url()->previous();
```

Svakoj od ovih metoda također se može pristupiti preko `URL` fasade :

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

### URL-ovi za imenovane rute

`route` pomoćnik se može koristiti za generiranje URL-ova za [imenovane rute](#). Imenovane rute vam omogućuju generiranje URL-ova bez povezivanja sa stvarnim URL-om definiranim na ruti. Zato ako se promijeni URL rute, ne morate unositi nikakve promjene u svoje pozive `route` funkciji. Na primjer, zamislite da vaša aplikacija sadrži rutu definiranu na sljedeći način:

```
Route::get('/post/{post}', function (Post $post) {
    // ...
})->name('post.show');
```

Za generiranje URL-a za ovu rutu, možete koristiti `route` pomoćnik ovako:

```
echo route('post.show', ['post' => 1]);

// http://primjer.com/post/1
```

Naravno, `route` pomoćnik se također može koristiti za generiranje URL-ova za rute s više parametara:

```
Route::get('/post/{post}/comment/{comment}', function (Post $post, Comment
$comment) {
    // ...
})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]);
```

```
// http://primjer.com/post/1/comment/3
```

Svi dodatni elementi matrice koji ne odgovaraju parametrima definicije rute bit će dodani stringu upita URL-a:

```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);

// http://primjer.com/post/1?search=rocket
```

### *Eloquent modeli*

Često ćete generirati URL-ove pomoću ključa rute (obično primarnog ključa) Eloquent modela. Iz tog razloga možete proslijediti Eloquent modele kao vrijednosti parametra. `route` pomoćnik će automatski izdvojiti ključ rute modela:

```
echo route('post.show', ['post' => $post]);
```

### *Potpisani URL-ovi*

Laravel vam omogućava jednostavno kreiranje „potpisanih“ URL-ova za imenovane rute. Ovi URL-ovi imaju hash „potpis“ pridodan stringu upita koji omogućava Laravelu da potvrdi da URL nije mijenjan otkad je kreiran. Potpisani URL-ovi posebno su korisni za rute koje su javno dostupne, ali im je potreban sloj zaštite od URL manipulacije.

Na primjer, možete koristiti potpisane URL-ove za implementiranje javnog linka „odjave pretplate“ koja se šalje vašim klijentima e-poštom. Za kreiranje potpisanog URL-a za imenovanu rutu, koristite `signedRoute` metodu `URL` fasade:

```
use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);
```

Možete isključiti domenu iz potpisanog hash-a URL-a davanjem `absolute` argumenta `signedRoute` metodi:

```
return URL::signedRoute('unsubscribe', ['user' => 1], absolute: false);
```

Ako želite generirati privremeni potpisani URL rute koji ističe nakon određenog vremena, možete koristiti `temporarySignedRoute` metodu. Kada Laravel potvrdi privremeni potpisani URL rute, osigurat će da vremenska oznaka (engl. timestamp) isteka koja je kodirana u potpisanom URL-u nije istekla:

```
use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(
    'unsubscribe', now()->addMinutes(30), ['user' => 1]
);
```

### *Potvrđivanje potpisanih zahtjeva za rutu*

Kako biste potvrdili da dolazni zahtjev ima važeći potpis, trebali biste pozvati `hasValidSignature` metodu na dolaznoj `Illuminate\Http\Request` instanci:

```
use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (!$request->hasValidSignature()) {
        abort(401);
    }

    // ...
})->name('unsubscribe');
```

Ponekad ćete možda trebati dopustiti frontend-u vaše aplikacije da doda podatke potpisanom URL-u, kao na primjer kada radite paginaciju<sup>49</sup> na strani klijenta. Zato možete navesti parametre upita zahtjeva koje treba zanemariti kada validirate potpisani URL-a pomoću `hasValidSignatureWhileIgnoring` metode. Upamtite, ignoriranje parametara omogućava da bilo tko modificira te parametre u zahtjevu:

```
if (!$request->hasValidSignatureWhileIgnoring(['page', 'order'])) {
    abort(401);
}
```

Umjesto validacije potpisanih URL-ova pomoću instance dolaznog zahtjeva, možete dodijeliti `Illuminate\Routing\Middleware\ValidateSignature` [middleware](#) ruti. Ako dolazni zahtjev nema važeći potpis, middleware će automatski vratiti [403 HTTP odgovor](#):

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed');
```

Ako vaši potpisani URL-ovi ne uključuju domenu u URL hash-u, trebali biste dati `relative` argument u middleware:

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed:relative');
```

---

<sup>49</sup> Paginacija je postupak odvajanja tiskanog ili digitalnog sadržaja na odvojene stranice. Za dokumente za ispis i neke mrežne sadržaje, paginacija se također odnosi na automatizirani postupak dodavanja uzastopnih brojeva kako bi se identificirao redoslijed stranica.

### *Odgovaranje na rute koje su označene kao nevažeće*

Kada netko posjeti potpisani URL koji je istekao, primit će stranicu generičke greške sa `403` HTTP statusnim kodom. Međutim, ovo ponašanje možete prilagoditi definiranjem prilagođenog „prikazivanja“ (engl. „renderable“) anonimne funkcije (engl. closure) za `InvalidSignatureException` iznimku u vašoj `bootstrap/app.php` datoteci:

```
use Illuminate\Routing\Exceptions\InvalidSignatureException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->render(function (InvalidSignatureException $e) {
        return response()->view('errors.link-expired', status: 403);
    });
})
```

### URL-ovi za radnje kontrolera

Funkcija `action` generira URL za danu radnju kontrolera:

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

Ako metoda kontrolera prihvaća parametre rute, možete proslijediti asocijativnu matricu parametara rute kao drugi argument funkciji:

```
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

### Zadane vrijednosti

Za neke aplikacije možda ćete htjeti navesti zadane vrijednosti na nivou zahtjeva za određene URL parametre. Na primjer, zamislite da mnoge vaše rute definiraju `{locale}` parametar:

```
Route::get('/{locale}/posts', function () {
    // ...
})->name('post.index');
```

Nezgrapno je uvijek preskočiti `locale` svaki put kada pozovete `route` pomoćnika. Dakle, možete koristiti `URL::defaults` metodu za definiranje zadane vrijednosti za ovaj parametar koja će se uvijek primjenjivati tokom trenutnog zahtjeva. Možda biste željeli pozvati ovu metodu iz [route middleware-a](#) tako da imate pristup trenutnom zahtjevu:

```
<?php

namespace App\Http\Middleware;

use Closure;
```

```

use Illuminate\Http\Request;
use Illuminate\Support\Facades\URL;
use Symfony\Component\HttpFoundation\Response;

class SetDefaultLocaleForUrls
{
    /**
     * Obrada dolaznog zahtjeva.
     *
     * @param  \Closure(\Illuminate\Http\Request):
     * (\Symfony\Component\HttpFoundation\Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        URL::defaults(['locale' => $request->user()->locale]);

        return $next($request);
    }
}

```

Nakon što je zadana vrijednost za `locale` parametar postavljena, više ne morate proslijeđivati njegovu vrijednost kada generirate URL-ove pomoću `route` pomoćnika.

#### *Zadane postavke URL-a i prioritet middleware-a*

Postavljanje URL podrazumijevanih vrijednosti može ometati Laravel-ovo rukovanje implicitnim modelom vezanja. Zato biste trebali [odrediti prioritete svom middleware-u](#) koji postavlja podrazumijevane URL-ove da se izvrše prije Laravel-ovog vlastitog `SubstituteBindings` middleware-a. To možete postići tako da koristite prioritetnu middleware metodu u datoteci `bootstrap/app.php` vaše aplikacije:

```

->withMiddleware(function (Middleware $middleware) {
    $middleware->priority([
        \Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
        \Illuminate\Cookie\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \Illuminate\Foundation\Http\Middleware\ValidateCsrfToken::class,
        \Illuminate\Contracts\Auth\Middleware\AuthenticatesRequests::class,
        \Illuminate\Routing\Middleware\ThrottleRequests::class,
        \Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
        \Illuminate\Session\Middleware\AuthenticateSession::class,
    ]);
});

```

```
        \App\Http\Middleware\SetDefaultLocaleForUrls::class,  
        \Illuminate\Routing\Middleware\SubstituteBindings::class,  
        \Illuminate\Auth\Middleware\Authorize::class,  
    ];  
}
```

## HTTP sesija

### Uvod

Budući da su aplikacije koje pokreće HTTP bez statusa, sesije pružaju način za pohranu informacija o korisniku kroz više zahtjeva. Ti se podaci o korisniku obično smještaju u trajno spremište/backend kojem se može pristupiti iz naknadnih zahtjeva.

Laravel se isporučuje s različitim backend-ima sesije kojima se pristupa putem ekspresivnog, unificiranog API-ja. Uključena je podrška za popularne backend-ove kao što su [Memcached](#), [Redis](#) i baze podataka.

### Konfiguracija

Konfiguracijska datoteka sesije vaše aplikacije pohranjena je na `config/session.php`. Obavezno pregledajte opcije koje su vam dostupne u ovoj datoteci. Prema zadanim postavkama, Laravel je konfiguriran za korištenje `file` drivera sesije, što će dobro funkcionirati za mnoge aplikacije. Ako će vaša aplikacija imati uravnoteženo opterećenje na više web servera, trebali biste odabrati centralizirano spremište kojem svi serveri mogu pristupiti, kao što je Redis ili baza podataka.

Opcija konfiguracije sesije `driver` definira gdje će podaci o sesiji biti pohranjeni za svaki zahtjev. Laravel se isporučuje s nekoliko sjajnih drivera odmah po otvaranju:

- `file` - sesije su pohranjene u `storage/framework/sessions`.
- `cookie` - sesije su pohranjene u sigurnim, šifriranim kolačićima.
- `database` - sesije su pohranjene u relacijskoj bazi podataka.
- `memcached / redis` - sesije su pohranjene u jednu od ovih brzih pohrana temeljenih na kešu.
- `dynamodb` - sesije su pohranjene u AWS DynamoDB.
- `array` - sesije su pohranjene u PHP matrici i neće biti postojane.

### NAPOMENA:



Driver matrice primarno se koristi tokom testiranja i sprječava zadržavanje podataka pohranjenih u sesiji da ne budu postojani.

### Preduvjeti za driver

#### Baza podataka

Kada koristite `database` driver sesije, morat ćete osigurati da imate tablicu baze podataka koja će sadržavati podatke sesije. Obično je to uključeno u Laravelovu zadanu `0001_01_01_000000_create_users_table.php` migraciju baze podataka; međutim, ako iz bilo kojeg razloga nemate tablicu sesija, možete upotrijebiti naredbu `make:session-table` Artisan za generiranje ove migracije:

```
php artisan make:session-table
```

```
php artisan migrate
```

## Redis

Prije korištenja Redis sesija s Laravelom, morat ćete ili instalirati PhpRedis PHP proširenje pomoću PECL-a ili instalirati paket `predis/predis` (~1.0) pomoću Composer-a. Za više informacija o konfiguraciji Redisa, konzultirajte Laravelovu Redis dokumentaciju .

### NAPOMENA:



Varijabla okruženja `SESSION_CONNECTION` ili opcija povezivanja u konfiguracijskoj datoteci `session.php` može se koristiti za određivanje koja se Redis veza koristi za pohranu sesije.

## Interakcija sa sesijom

### Dohvaćanje podataka

Postoje dva primarna načina rada s podacima sesije u Laravelu: globalni `session` pomoćnik i pomoću `Request` instance. Prvo, pogledajmo pristup sesiji pomoću `Request` instance, koja se može nagovijestiti tip (engl. type-hint) pomoću anonimne funkcije (engl. closure) rute ili metode kontrolera. Upamtite, zavisnosti metode kontrolera automatski se injektiraju pomoću Laravel [servisnog kontejnera](#):

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Prikaži profil danog korisnika.
     */
    public function show(Request $request, string $id): View
    {
        $value = $request->session()->get('key');

        // ...

        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

Kada dohvatite stavku iz sesije, također možete proslijediti zadanu vrijednost kao drugi argument `get` metodi. Ova zadanu vrijednost bit će vraćena ako navedeni ključ ne postoji u sesiji. Ako metodi



prosljedite anonimnom funkcijom (engl. closure) kao zadanu vrijednost `get` metodi a traženi ključ ne postoji, anonimna funkcija (engl. closure) će se izvršiti i vratiti njen rezultat:

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

### Globalni pomoćnik sesije

Također možete koristiti globalnu `session` PHP funkciju za dohvaćanje i pohranjivanje podataka u sesiji. Kada se `session` pomoćnik pozove s jednim string argumentom, vratit će vrijednost tog ključa sesije. Kada se pomoćnik pozove s matricom parova ključ/vrijednost, te će vrijednosti biti pohranjene u sesiji:

```
Route::get('/home', function () {
    // Dohvati dio podataka iz sesije...
    $value = session('key');

    // Odredi zadanu vrijednost...
    $value = session('key', 'default');

    // Pohrani dio podataka u sesiju...
    session(['key' => 'value']);
});
```

### NAPOMENA:



Mala je praktična razlika između korištenja sesije pomoću instance HTTP zahtjeva i korištenja globalnog `session` pomoćnika. Obje metode se mogu testirati pomoću `assertSessionHas` metode koja je dostupna u svim vašim testnim slučajevima.

### Dohvaćanje svih podataka sesije

Ako želite dohvatiti sve podatke u sesiji, možete koristiti `all` metodu:

```
$data = $request->session()->all();
```

### Dohvaćanje dijela podataka sesije

Metode `only` i `except` mogu se koristiti za dohvaćanje podskupa podataka o sesiji:

```
$data = $request->session()->only(['username', 'email']);

$data = $request->session()->except(['username', 'email']);
```

### Određivanje da li stavka postoji u sesiji

Da biste utvrdili je li stavka prisutna u sesiji, možete koristiti `has` metodu. `has` metoda vraća `true` ako je stavka prisutna, a nije `null`:

```
if ($request->session()->has('users')) {  
    // ...  
}
```

Da biste utvrdili je li stavka prisutna u sesiji, čak i ako je njezina vrijednost `null`, možete koristiti `exists` metodu:

```
if ($request->session()->exists('users')) {  
    // ...  
}
```

Da biste utvrdili da stavka nije prisutna u sesiji, možete koristiti `missing` metodu. `missing` metoda vraća `true` ako stavka nije prisutna:

```
if ($request->session()->missing('users')) {  
    // ...  
}
```

### Pohranjivanje podataka

Za pohranu podataka u sesiji obično ćete koristiti `put` metodu instance zahtjeva ili globalnog `session` pomoćnika:

```
// Preko zahtjeva instance...  
$request->session()->put('key', 'value');  
  
// Preko globalnog "session" pomoćnika...  
session(['key' => 'value']);
```

### Guranje vrijednosti u matricu sesije

`push` metoda se može koristiti za guranje nove vrijednosti uz sesiju koja je matrica. Na primjer, ako `user.teams` ključ sadrži matricu imena timova, možete unijeti novu vrijednost u matricu ovako:

```
$request->session()->push('user.teams', 'developers');
```

### Dohvaćanje i brisanje stavke

`pull` metoda će dohvatiti i izbrisati stavku iz sesije u jednoj izjavi:

```
$value = $request->session()->pull('key', 'default');
```

### *Uvećavanje i umanjivanje vrijednosti sesije*

Ako podaci vaše sesije sadrže cijeli broj koji želite povećati ili smanjiti, možete koristiti `increment` i `decrement` metode:

```
$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);
```

### *Flash podataka*

Ponekad ćete možda poželjeti pohraniti stavke u sesiji za sljedeći zahtjev. To možete učiniti pomoću `flash` metode. Podaci pohranjeni u sesiji pomoću ove metode bit će dostupni odmah i tokom sljedećeg HTTP zahtjeva. Nakon sljedećeg HTTP zahtjeva, flash podaci bit će izbrisani. Flash podaci prvenstveno su korisni za kratkotrajne statusne poruke:

```
$request->session()->flash('status', 'Zadatak je bio uspješan!');
```

Ako trebate zadržati svoje flash podatke za nekoliko zahtjeva, možete koristiti `reflash` metodu koja će zadržati sve flash podatke za dodatni zahtjev. Ako trebate samo zadržati određene flash podatke, možete koristiti `keep` metodu:

```
$request->session()->reflash();

$request->session()->keep(['username', 'email']);
```

Kako biste zadržali svoje flash podatke samo za trenutni zahtjev, možete koristiti `now` metodu:

```
$request->session()->now('status', ' Zadatak je bio uspješan!');
```

### *Brisanje podataka*

`forget` metoda će ukloniti dio podataka iz sesije. Ako želite ukloniti sve podatke iz sesije, možete koristiti `flush` metodu:

```
// Zaboravi jedan ključ...
$request->session()->forget('name');

// Zaboravi više ključeva...
$request->session()->forget(['name', 'status']);

$request->session()->flush();
```

### *Regeneriranje ID-a sesije*

Regeneriranje ID-a sesije često se radi kako bi se spriječilo zlonamjerne korisnike da iskoriste [napad fiksacijom sesije](#) na vašu aplikaciju.

Fiksacija sesije je napad koji dopušta napadaču da otme ispravnu korisničku sesiju. Napad istražuje ograničenje u načinu na koji web aplikacija upravlja ID-om sesije, točnije ranjivu web aplikaciju. Kod autentifikacije korisnika, ne dodjeljuje se novi ID sesije, što omogućava korištenje postojećeg ID-a sesije. Napad se sastoji od dobivanja važećeg ID-a sesije (npr. povezivanjem s aplikacijom), navođenja korisnika da se autentificira s tim ID-om sesije, a zatim otimanja sesije koju je potvrdio korisnik saznanjem o korištenom ID-u sesije. Napadač mora dati legitiman ID sesije web aplikacije i pokušati natjerati žrtvin browser da ga koristi. Dakle napad počinje prije prijave korisnika. Neke od najčešćih tehnika za ovaj napad su:

- token sesije u argumentu URL-a i ID sesije šalje se žrtvi u hiperlinku i žrtva pristupa stranici pomoću zlonamjernog URL-a.;
- token sesije u polju skrivene forme: u ovoj metodi žrtva mora biti prevarena da se autentificira na ciljnom web serveru, koristeći formu za prijavu razvijen za napadača, forma se može nalaziti na zlom web serveru ili direktno u e-pošti u HTML formatu;
- ID sesije u kolačiću tj. napad skriptom na strani klijenta: većina browsera podržava izvršavanje skriptiranja na strani klijenta. U ovom slučaju, agresor bi mogao koristiti napade ubacivanja koda kao XSS (Cross-site scripting) napad za umetanje zlonamjernog koda u hipervezu poslanu žrtvi i popravljanje ID-a sesije u kolačiću. Korištenjem funkcije document.cookie, browser koji izvršava naredbu postaje sposoban popraviti vrijednosti unutar kolačića koje će koristiti za održavanje sesije između klijenta i web aplikacije.
  - o Umetanje taga između <META> tagova također se smatra napadom injektiranjem koda, međutim, razlikuje se od XSS napada gdje se nepoželjne skripte mogu onemogućiti ili se može odbiti izvršenje. Napad ovom metodom postaje puno učinkovitiji jer je nemoguće onemogućiti obradu ovih tagova u browserima
  - o HTTP odgovor zaglavlja – ova metoda istražuje odgovor servera kako bi se lažirao ID sesije u browseru žrtve. Uključivanjem parametra Set-Cookie u odgovor HTTP zaglavlja, napadač može umetnuti vrijednost ID-a sesije u kolačić i poslati ga žrtvinom browser-u.

Laravel automatski regenerira ID sesije tokom autentifikacije ako koristite jedan od Laravel [ovih aplikacijskih paketa za početnike](#) ili Laravel Fortify ; međutim, ako trebate ručno ponovno generirati ID sesije, možete koristiti regenerate metodu:

```
$request->session()->regenerate();
```

Ako trebate ponovno generirati ID sesije i ukloniti sve podatke iz sesije u jednoj izjavi, možete koristiti [invalidate](#) metodu:

```
$request->session()->invalidate();
```

## Blokiranje sesije

### UPOZORENJE:



Da biste koristili blokiranje sesije, vaša aplikacija mora koristiti keš driver koji podržava atomska zaključavanja. Trenutačno ti keš driveri (upravljački programi) uključuju `memcached`, `dynamodb`, `redis`, `database`, `file` i `array` drivere. Osim toga, ne smijete koristiti `cookie` driver sesije.

Prema zadanim postavkama, Laravel dopušta istovremeno izvršavanje zahtjeva koji koriste istu sesiju. Tako, na primjer, ako koristite JavaScript HTTP biblioteku da napravite dva HTTP zahtjeva svojoj aplikaciji, oba će se izvršiti u isto vrijeme. Za mnoge aplikacije to nije problem; međutim, gubitak podataka sesije može se dogoditi u malom podskupu aplikacija koje upućuju istovremene zahtjeve dvjema različitim krajnjim točkama aplikacije koje obje zapisuju podatke u sesiju.

Kako bi to ublažio, Laravel pruža funkcionalnost koja vam omogućava ograničavanje istovremenih zahtjeva za određenu sesiju. Za početak možete jednostavno ulančati `block` metodu u svojoj definiciji rute. U ovom primjeru, dolazni zahtjev prema `/profile` krajnjoj točki usvojio bi zaključavanje sesije. Dok se ovo zaključavanje održava, svi dolazni zahtjevi prema `/profile` ili `/order` krajnjim točkama koje dijele isti ID sesije čekat će da prvi zahtjev završi izvršavanje prije nego što se nastavi s izvršenjem:

```
Route::post('/profile', function () {
    // ...
})->block($lockSeconds = 10, $waitSeconds = 10)

Route::post('/order', function () {
    // ...
})->block($lockSeconds = 10, $waitSeconds = 10)
```

`block` metoda prihvaća dva opsijska argumenta. Prvi argument koji `block` metoda prihvaća je maksimalni broj sekundi koje zaključavanje sesije treba zadržavati prije nego što se otpusti. Naravno, ako zahtjev završi s izvršavanjem prije tog vremena, zaključavanje će biti raskinuto ranije.

Drugi argument koji prihvaća `block` metoda je broj sekundi koje bi zahtjev trebao čekati dok pokušava dobiti zaključavanje sesije. `Illuminate\Contracts\Cache\LockTimeoutException` će biti izbačen ako zahtjev ne može dobiti zaključavanje sesije unutar zadanog broja sekundi.

Ako nijedan od ovih argumenata nije proslijeđen, zaključavanje će se dobiti najviše 10 sekundi, a zahtjevi će čekati najviše 10 sekundi dok pokušavaju dobiti zaključavanje:

```
Route::post('/profile', function () {
    // ...
})->block()
```

## Dodavanje prilagođenih upravljačkih programa sesije

### Implementacija upravljačkog programa

Ako niti jedan od postojećih sesijskih drivera ne odgovara potrebama vaše aplikacije, Laravel omogućava pisanje vlastitog Upravljačkog sklopa za rukovanje sesijama (engl. session handler). Vaš

prilagođeni Upravljački sklop za rukovanje sesijama (engl. session handler) trebao bi implementirati ugrađeni PHP `SessionHandlerInterface`. Ovaj interface sadrži samo nekoliko jednostavnih metoda. Jogunasta MongoDB implementacija izgleda ovako:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

#### NAPOMENA:



Laravel se ne isporučuje s direktorijem koji sadrži vaše ekstenzije. Možete ih slobodno postaviti gdje god želite. U ovom primjeru, stvorili smo Extensions direktorij za smještaj `MongoSessionHandler`.

Budući da svrha ovih metoda nije lako razumljiva, pokrijmo brzo što svaka od metoda radi:

- `open` metoda bi se obično koristila u sistemima pohrane sesija temeljenim na datotekama. Budući da se Laravel isporučuje s `file` sesijskim driverom, rijetko ćete morati nešto staviti u ovu metodu. Možete jednostavno ostaviti ovu metodu praznom.
- `close` metoda, kao i `open` metoda, obično može zanemariti. Za većinu driver-a to nije potrebno.
- `read` metoda bi trebala vratiti string verziju podataka o sesiji povezanih s danim `$sessionId`. Nema potrebe za bilo kakvom serijalizacijom<sup>50</sup> ili drugim kodiranjem kada dohvaćate ili pohranjujete podatke o sesiji u vašem driveru, jer će Laravel izvršiti serijalizaciju za vas.
- `write` metoda bi trebala zapisati zadani `$data` string povezan s `$sessionId` nekim trajnim sistemom za pohranu, kao što je MongoDB ili drugi sistem za pohranu po vašem izboru. Opet, ne biste trebali izvoditi nikakvu serijalizaciju - Laravel će to već riješiti umjesto vas.
- `destroy` metoda bi trebala ukloniti podatke povezane s `$sessionId` iz trajne pohrane.

<sup>50</sup> Serijalizacija je proces kroz koji dobivate string koji možete sačuvati na bilo kom mediju (disk, baza, sesija, ...) i kasnije od nje deserijalizacijom rekreirati dani objekat.

```
$serijalizirajMe = array("Ovo", "je", "PHP", "blog");
```

```
$serijalizirano = serialize($serijalizujMe);
```

```
echo $serijalizirano;
```

```
Biće ispisano: a:4:{i:0;s:3:"Ovo";i:1;s:2:"je";i:2;s:3:"PHP";i:3;s:4:"blog";}
```

- `gc` metoda bi trebala uništiti sve podatke o sesiji koji su stariji od danog `$lifetime`, što je vremenska oznaka UNIX-a. Za sisteme koji sami ističu kao što su Memcached i Redis, ova se metoda može ostaviti prazna.

### Registracija driver-a

Nakon što je vaš driver implementiran, spremni ste ga registrirati u Laravelu. Za dodavanje dodatnih drivera Laravelovoj backend sesije, možete koristiti `extend` metodu koju pruža `Session facade`. Trebali biste pozvati `extend` metodu iz `boot` metode `davatelja usluga`. To možete učiniti od postojećeg `App\Providers\AppServiceProvider` ili kreirati potpuno novog davatelja:

```
<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Registrirajte bilo koju aplikacijsku uslugu.
     */
    public function register(): void
    {
        // ...
    }

    /**
     * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
     */
    public function boot(): void
    {
        Session::extend('mongo', function (Application $app) {
            // Vraća implementaciju SessionHandlerInterface...
            return new MongoSessionHandler;
        });
    }
}
```

Nakon što se sesijski driver registrira, možete koristiti mongo driver u vašoj `config/session.php` konfiguracijskoj datoteci.

## Validacija

### Uvod

Laravel nudi nekoliko različitih pristupa za provjeru ispravnosti ulaznih podataka vaše aplikacije. Najčešće se koristi `validate` metoda dostupna na svim dolaznim HTTP zahtjevima. Međutim, raspraviti ćemo i druge pristupe validaciji.

Laravel uključuje širok izbor praktičnih pravila provjere ispravnosti koja možete primijeniti na podatke, čak pruža mogućnost provjere jesu li vrijednosti jednoznačne u danoj tablici baze podataka. Detaljno ćemo pokriti svako od ovih pravila provjere kako biste bili upoznati sa svim svojstvima ispravnosti Laravel-a.

### Brzi početak provjere ispravnosti

Kako bismo saznali više o Laravel-ovim moćnim svojstvima provjere ispravnosti, pogledajmo potpuni primjer provjere ispravnosti forme i prikazivanja poruka o grešci natrag korisniku. Čitajući ovaj pregled, steći ćete dobro opće razumijevanje o tome kako validirati podatke dolaznog zahtjeva koristeći Laravel:

### Definiranje ruta

Prvo, pretpostavimo da imamo sljedeće rute definirane u našoj `routes/web.php` datoteci:

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

`GET` ruta će korisniku prikazati formu za stvaranje novog bloga, dok će `POST` ruta pohraniti novi blog post u bazu podataka.

### Kreiranje kontrolera

Zatim, pogledajmo jednostavan kontroler koji obrađuje dolazne zahtjeve za ove rute. Ostavit ćemo `store` metodu za sada praznom:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\View\View;

class PostController extends Controller
{
    /**
     * Pokaži formu za stvaranje novog posta na blogu.
     */
    public function create(): View
    {
```



```
        return view('post.create');
    }

    /**
     * Pohrani novi blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        // Provjeri ispravnost i pohrani blog post...

        $post = /** ... */

        return to_route('post.show', ['post' => $post->id]);
    }
}
```

### *Zapisivanje i provjera ispravnosti logike*

Sada smo spremni ispuniti našu `store` metodu logikom za provjeru validnosti novog posta na blogu. Da bismo to učinili, koristit ćemo `validate` metodu koju nudi `Illuminate\Http\Request` objekt. Ako prođu pravila validnosti, vaš će se kôd nastaviti normalno izvršavati; međutim, ako provjera validnosti ne uspije, `Illuminate\Validation\ValidationException` iznimka bit će izbačena i odgovarajući odgovor o grešci automatski će biti poslan nazad korisniku.

Ako provjera validnosti ne uspije tokom tradicionalnog HTTP zahtjeva, generirat će se odgovor preusmjerenja na prethodni URL. Ako je dolazni zahtjev XHR zahtjev, vratit će se [JSON odgovor koji sadrži poruke o grešci validnosti](#).

Da bismo bolje razumjeli `validate` metodu, vratimo se na `store` metodu:

```
/**
 * Pohrani novi blog post.
 */
public function store(Request $request): RedirectResponse
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // Blog post je validan...

    return redirect('/posts');
}
```

Kao što vidite, pravila provjere validnosti prosljeđuju se u `validate` metodu. Ne brinite - sva dostupna pravila provjere validnosti su [dokumentirana](#). Opet, ako provjera validnosti ne uspije, automatski će se generirati odgovarajući odgovor. Ako prođe provjera validnosti, naš kontroler će nastaviti normalno izvršavati.

Alternativno, pravila validnosti mogu se navesti kao matrice pravila umjesto jednog `|` razgraničenog stringa:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Osim toga, možete koristiti `validateWithBag` metodu za provjeru validnosti zahtjeva i pohranjivanje svih poruka o grešci unutar [imenovane torbe s greškama](#):

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

#### *Zaustavljanje nakon prve neuspjele provjere validnosti*

Ponekad ćete možda htjeti zaustaviti izvođenje pravila provjere validnosti na atributu nakon prve neuspjele provjere validnosti. Da biste to učinili, dodijelite `bail` pravilo atributu:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

U ovom primjeru, ako `unique` pravilo na `title` atributu ne uspije, `max` pravilo se neće provjeriti. Pravila će biti potvrđena redoslijedom kojim su dodijeljena.

#### *Napomena o ugniježđenim atributima*

Ako dolazni HTTP zahtjev sadrži „ugniježdene“ podatke polja, možete navesti ta polja u svojim pravilima provjere pomoću „točka“ sintakse:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

S druge strane, ako naziv vašeg polja sadrži doslovnu točku, možete izričito spriječiti da se to tumači kao „točka“ sintaksa tako da točku izbjegnute kosom crtom unazad:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'v1\.0' => 'required',
]);
```

### Prikaz validacijskih grešaka

Dakle, što ako polja dolaznog zahtjeva ne prođu zadana pravila validacije? Kao što je ranije spomenuto, Laravel će automatski preusmjeriti korisnika natrag na njegovu prethodnu lokaciju. Osim toga, sve greške provjere validnosti i [unos zahtjeva](#) automatski će se [prikazati kao flash u sesiji](#).

Varijabla `$errors` se dijeli sa svim pogledima vaše aplikacije pomoću middleware `Illuminate\View\Middleware\ShareErrorsFromSession`, koji osigurava web middleware grupa. Kada se primijeni ovaj middleware, `$errors` varijabla će uvijek biti dostupna u vašim pogledima, što vam omogućava da pretpostavite da `$errors` je varijabla uvijek definirana i da se može sigurno koristiti. Varijabla `$errors` će biti instanca `Illuminate\Support\MessageBag`. Za više informacija o radu s ovim objektom, pogledajte njegovu dokumentaciju.

Dakle, u našem primjeru, korisnik će biti preusmjeren na `create` metodu našeg kontrolera kada provjera validacije ne uspije, što nam omogućava prikaz poruka o greškama u pogledu:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!--Kreirana forma za poštu -->
```

### Prilagođavanje poruka o grešci

Svako od Laravel-ovih ugrađenih validacijskih pravila ima poruku o grešci koja se nalazi u `lang/en/validation.php` datoteci vaše aplikacije. Ako vaša aplikacija nema `lang` direktorij, možete uputiti Laravel da ga kreira pomoću `lang:publish` Artisan naredbe.

Unutar `lang/en/validation.php` datoteke pronaći ćete stavku prijevoda za svako validacijsko pravilo. Možete slobodno promijeniti ili modificirati ove poruke na temelju potreba vaše aplikacije.

Osim toga, možete kopirati ovu datoteku u direktorij drugog jezika kako biste preveli poruke za jezik svoje aplikacije. Kako biste saznali više o Laravel lokalizaciji, pogledajte kompletnu dokumentaciju o lokalizaciji.

Osim toga, možete kopirati ovu datoteku u direktorij drugog jezika kako biste preveli poruke za jezik svoje aplikacije. Kako biste saznali više o Laravel lokalizaciji, pogledajte kompletnu dokumentaciju o lokalizaciji.

**UPOZORENJE:**

Prema zadanim postavkama skelet aplikacije Laravel ne uključuje `lang` direktorij. Ako želite prilagoditi Laravel-ove jezične datoteke, možete ih objaviti pomoću `lang:publish` Artisan naredbe.

*XHR zahtjevi i validacija*

U ovom smo primjeru koristili tradicionalnu formu za slanje podataka aplikaciji. Međutim, mnoge aplikacije primaju XHR zahtjeve od frontend-a kojeg pokreće JavaScript. Kada koristite `validate` metodu tokom XHR zahtjeva, Laravel neće generirati odgovor preusmjeravanja. Umjesto toga, Laravel generira [JSON odgovor koji sadrži sve validacijske greške](#). Ovaj JSON odgovor bit će poslan s HTTP statusnim kodom 422.

*@error direktiva*

Možete koristiti `@error` Blade direktivu da brzo odredite postoje li poruke o grešci validnosti za dati atribut. Unutar `@error` direktive, možete ponoviti `$message` varijablu za prikaz poruke o grešci:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
  type="text"
  name="title"
  class="@error('title') is-invalid @enderror">

@error('title')
  <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Ako koristite [imenovane torbe za greške \(engl. named error bags\)](#), možete proslijediti naziv torbe za greške kao drugi argument `@error` direktivi:

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

### Ponovno popunjavanje formi

Kada Laravel generira odgovor za preusmjeravanje zbog validacijske greške, radna okolina će automatski [prikazati kao flash sav zahtjevani unos u sesiju](#). Ovo je učinjeno kako biste mogli jednostavno pristupiti unosu tokom sljedećeg zahtjeva i ponovno popuniti formu koji je korisnik pokušao poslati.

Da biste dohvatili flash unos iz prethodnog zahtjeva, pozovite `old` metodu na `Illuminate\Http\Request` instanci. `old` metoda će povući prethodno blicane ulazne podatke iz [sesije](#):

```
$title = $request->old('title');
```

Laravel također nudi globalnog `old` pomoćnika (engl. helper). Ako prikazujete stari unos ulaz [Blade predloška](#), praktičnije je koristiti `old` pomoćnik za ponovno popunjavanje formi. Ako za dano polje ne postoji stari unos, `null` će se vratiti:

```
<input type="text" name="title" value="{{ old('title') }}">
```

### Napomena o opcionalnim poljima

Prema zadanim postavkama, Laravel uključuje `TrimStrings` i `ConvertEmptyStringsToNull` middleware u vaš aplikacijski globalni middleware stek. Ovaj middleware je izlistan u steku prema `App\Http\Kernel` klasi. Zbog toga ćete često morati označiti „opcionalna“ polja zahtjeva kao `nullable` ako ne želite da validator smatra `null` vrijednosti nevažećima. Na primjer:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

U ovom primjeru specificiramo da `publish_at` polje može biti `null` ili validan prikaz datuma. Ako se `nullable` modifikator ne doda definiciji pravila, validator bi smatrao `null` nevažećim datumom.

### Format odgovora validacijske greške

Kada vaša aplikacija izbací `Illuminate\Validation\ValidationException` iznimku i dolazni HTTP zahtjev očekuje JSON odgovor, Laravel će automatski formatirati poruke o grešci umjesto vas i vratiti `422 Unprocessable Entity` HTTP odgovor.

U nastavku možete pregledati primjer JSON formata odgovora za greške provjere validnosti. Imajte na umu da su ugniježđeni ključevi greške poravnati u format notacije „točka“:

```
{
  "message": "Ime tima mora biti string (i još 4 greške) ",
  "errors": {
    "team_name": [
      "Ime tima mora biti string.",

```

```
        "Ime tima mora biti barem jedan znak."
    ],
    "authorization.role": [
        "Odabrana autorizacija.role nije važeća."
    ],
    "users.0.email": [
        "Polje users.0.email je obavezno."
    ],
    "users.2.email": [
        "users.2.email mora biti ispravna email adresa."
    ]
  }
}
```

## Validacija zahtjeva forme

### *Izrada zahtjeva za formu*

Za složenije scenarije provjere validnosti, možda ćete htjeti stvoriti „zahtjev za formu“. Zahtjevi forme prilagođene su klase zahtjeva koje sadrže vlastitu logiku provjere validnosti i autorizacije. Za izradu klase zahtjeva forme, možete koristiti `make:request` Artisan CLI naredbu:

```
php artisan make:request StorePostRequest
```

Generirana forma zahtjeva klase bit će smještena u `app/Http/Requests` direktorij. Ako ovaj direktorij ne postoji, bit će kreiran kada pokrenete `make:request` naredbu. Svaki zahtjev za formu koju generira Laravel ima dvije metode: `authorize` i `rules`.

Kao što ste možda pogodili, `authorize` metoda je odgovorna za utvrđivanje može li trenutno autentificirani korisnik izvršiti radnju predstavljenu zahtjevom, dok `rules` metoda vraća pravila provjere validnosti koja bi se trebala primijeniti na podatke zahtjeva:

```
/**
 * Dobij pravila provjere validacije koja se primjenjuju na zahtjev.
 *
 * @return array<string, \Illuminate\Contracts\Validation\Rule|array|string>
 */
public function rules(): array
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

}

**NAPOMENA:**

Možete nagovijestiti tip (engl. type-hint) bilo koju zavisnosti koje su vam potrebne unutar `rules` potpisa metode. Oni će se automatski riješiti pomoću [Laravel servisnog kontejnera](#).

Dakle, kako se procjenjuju pravila validacije? Sve što trebate učiniti je nagovijestiti tip (engl. type-hint) zahtjeva na vašoj metodi kontrolera. Zahtjev dolaznog forme provjerava se prije poziva metode kontrolera, što znači da ne morate zatrpiti svoj kontroler bilo kakvom validacijskom logikom:

```
/**
 * Pohrani novi blog post.
 */
public function store(StorePostRequest $request): RedirectResponse
{
    // Dolazni zahtjev je validan...

    // Dohvaćanje validiranih ulaznih podataka...
    $validated = $request->validated();

    // Dohvaćanje dijela validiranih ulaznih podataka...
    $validated = $request->safe()->only(['name', 'email']);
    $validated = $request->safe()->except(['name', 'email']);

    // Pohrani post na blogu...

    return redirect('/posts');
}
```

Ako validacija ne uspije, generirat će se odgovor za preusmjeravanje kako bi se korisnik vratio na njegovu prethodnu lokaciju. Greške će također biti blicane u sesiji tako da budu dostupne za prikazivanje. Ako je zahtjev bio XHR zahtjev, HTTP odgovor sa statusnim kodom 422 bit će vraćen korisniku uključujući [JSON prikaz grešaka validacije](#).

**NAPOMENA:**

Trebate li dodati provjeru zahtjeva forme u stvarnom vremenu na svoj Laravel frontend koje pokreće Inertia? Provjerite [Laravel Precognition](#).

*Izvođenje dodatne provjere validnosti*

Ponekad morate izvršiti dodatnu provjeru validnosti nakon što je vaša početna provjera validnosti dovršena. To možete postići pomoću `after` metode forme zahtjeva.

`after` metoda bi trebala vratiti matricu callable-ova ili anonimnih funkcija (engl. closures) koji će se pozvati nakon dovršetka provjere validnosti. Zadani pozivi će primiti `Illuminate\Validation\Validator` instancu, što vam omogućava dobivanje dodatnih poruka o grešci ako je potrebno:

```
use Illuminate\Validation\Validator;

/**
 * Dobij "naknadno" (engl. "after") validaciju callable-ova za zahtjev.
 */
public function after(): array
{
    return [
        function (Validator $validator) {
            if ($this->somethingElseIsInvalid()) {
                $validator->errors()->add(
                    'field',
                    'Nešto nije u redu s ovim poljem!'
                );
            }
        }
    ];
}
```

Kao što je navedeno, matrica koji vraća `after` metodom također može sadržavati klase koje se mogu pozvati. `__invoke` metoda ovih klasa će dobiti `Illuminate\Validation\Validator` instancu:

```
use App\Validation\ValidateShippingTime;
use App\Validation\ValidateUserStatus;
use Illuminate\Validation\Validator;

/**
 * Dobij "naknadno" (engl. "after") validaciju callable-ova za zahtjev.
 */
public function after(): array
{
    return [
        new ValidateUserStatus,
        new ValidateShippingTime,
        function (Validator $validator) {
            //
        }
    ];
}
```



### *Zaustavljanje nakon prve neuspjele provjere validnosti*

Dodavanjem `stopOnFirstFailure` svojstva vašoj klasi zahtjeva možete obavijestiti validator da bi trebao prestati provjeravati validnost svih atributa nakon što se dogodi jedna neuspješna provjere validnosti:

```
/**
 * Označava treba li se validator zaustaviti kod prvog neuspješnog pravila.
 *
 * @var bool
 */
protected $stopOnFirstFailure = true;
```

### *Prilagođavanje lokacije preusmjerenja*

Kao što je prethodno objašnjeno, generira se odgovor za preusmjerenje kako bi se korisnik vratio na njegovu prethodnu lokaciju kada provjera validnosti zahtjeva forme ne uspije. Međutim, slobodno možete prilagoditi ovo ponašanje. Da biste to učinili, definirajte `$redirect` svojstvo na svom zahtjevu forme:

```
/**
 * URI na koji bi korisnici trebali biti preusmjereni ako provjera validnosti ne
 * uspije.
 *
 * @var string
 */
protected $redirect = '/dashboard';
```

Ili, ako želite preusmjeriti korisnike na imenovanu rutu, umjesto toga možete definirati `$redirectToRoute` svojstvo:

```
/**
 * URI na koji bi korisnici trebali biti preusmjereni ako provjera validnosti ne
 * uspije.
 *
 * @var string
 */
protected $redirectToRoute = 'dashboard';
```

### *Autorizacija zahtjeva formi*

Klasa zahtjeva formi također sadrži `authorize` metodu. Unutar ove metode možete utvrditi ima li autentificirani korisnik stvarno ovlaštenje za ažuriranje danog resursa. Na primjer, možete utvrditi posjeduje li korisnik doista komentar na blogu koji pokušava ažurirati. Najvjerojatnije ćete komunicirati sa svojim ulazima za autorizaciju i pravilima unutar ove metode:

```
use App\Models\Comment;
```

```
/**
 * Odredi da li je korisnik ovlašten za podnošenje ovog zahtjeva.
 */
public function authorize(): bool
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

Budući da svi zahtjevi forme proširuju osnovnu Laravel klasu zahtjeva, možemo upotrijebiti metodu `user` za pristup trenutno autentificiranom korisniku. Također, obratite pažnju na poziv `route` metode u primjeru iznad. Ova metoda vam omogućava pristup URI parametrima definiranim na ruti koja se poziva, kao što je `{comment}` parametar u primjeru u nastavku:

```
Route::post('/comment/{comment}');
```

Zato, ako vaša aplikacija iskorištava prednost [povezivanja modela rute \(engl. route model binding\)](#), vaš kod može biti još sažetiji pristupom razriješenom modelu kao svojstvu zahtjeva:

```
return $this->user()->can('update', $this->comment);
```

Ako `authorize` metoda vrati `false`, HTTP odgovor sa statusnim kodom 403 automatski će se vratiti i vaša se metoda kontrolera neće izvršiti.

Ako planirate rukovati autorizacijskom logikom za zahtjev u drugom dijelu vaše aplikacije, možete `authorize` potpuno ukloniti metodu ili jednostavno vratiti `true`:

```
/**
 * Odredite je li korisnik ovlašten za podnošenje ovog zahtjeva.
 */
public function authorize(): bool
{
    return true;
}
```

#### NAPOMENA:



Možete nagovijestiti tip (engl. type-hint) svih zavisnosti koje su vam potrebne unutar `authorize` potpisa metode. Oni će se automatski riješiti pomoću [Laravel kontejnera usluga](#).

*Prilagođavanje poruka o grešci*

Možete prilagoditi poruke o greškama koje koristi formu zahtjeva nadjačavanjem `messages` metode. Ova metoda bi trebala vratiti matricu atribut/pravilo parove i njihove odgovarajuće poruke o grešci:

```
/**
 * Dobijte poruke o grešci za definirana pravila provjere validnosti.
 *
 * @return array<string, string>
 */
public function messages(): array
{
    return [
        'title.required' => 'Potreban je naslov',
        'body.required' => 'Potrebna je poruka',
    ];
}
```

*Prilagođavanje poruke o grešci*

Mnoge Laravelova ugrađena validacijska pravila poruka o greškama sadrže `:attribute` rezervirano mjesto. Ako želite da se `:attribute` rezervirano mjesto vaše validacijske poruke zamijeni imenom prilagođenog atributa, možete navesti prilagođena imena nadjačavanjem `attributes` metode. Ova metoda bi trebala vratiti matricu atribut/ime parova:

```
/**
 * Dobijte prilagođene attribute za greške validatora.
 *
 * @return array<string, string>
 */
public function attributes(): array
{
    return [
        'email' => 'email address',
    ];
}
```

*Priprema unosa za provjeru validnosti*

Ako trebate pripremiti ili očistiti bilo koje podatke iz zahtjeva prije nego što primijenite svoja pravila provjere validnosti, možete koristiti `prepareForValidation` metodu:

```
use Illuminate\Support\Str;

/**
 * Pripremi podatke za validaciju.
 */
protected function prepareForValidation(): void
{
}
```

```
$this->merge([
    'slug' => Str::slug($this->slug),
]);
}
```

Isto tako, ako trebate normalizirati podatke zahtjeva nakon dovršetka provjere validnosti, možete koristiti `passedValidation` metodu:

```
/**
 * Obrada pokušaja provjere validnosti.
 */
protected function passedValidation(): void
{
    $this->replace(['name' => 'Taylor']);
}
```

Ručno kreiranje validatora

Ako ne želite koristiti `validate` metodu na zahtjevu, možete kreirati instancu validatora ručno pomoću `Validator` [fasade](#). `make` metoda na fasadi generira novu instancu validatora:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * Pohrani novi blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                ->withErrors($validator)
        }
    }
}
```

```
        ->withInput();
    }

    // Dohvati validirani unos...
    $validated = $validator->validated();

    // Retrieve a portion of the validated input...
    $validated = $validator->safe()->only(['name', 'email']);
    $validated = $validator->safe()->except(['name', 'email']);

    // Pohrani blog post...

    return redirect('/posts');
}
}
```

Prvi argument proslijeđen `make` metodi su podaci koji se validiraju. Drugi argument je matrica pravila validnosti koja se trebaju primijeniti na podatke.

Nakon utvrđivanja da li provjera validnosti zahtjeva nije uspjela, možete upotrijebiti `withErrors` metodu za flash poruke o grešci u sesiji. Kada koristite ovu metodu, `$errors` varijabla će se automatski dijeliti s vašim pogledima nakon preusmjeravanja, što vam omogućava da ih jednostavno prikazete natrag korisniku. `withErrors` metoda prihvaća validator, `MessageBag` ili PHP `array`.

#### *Zaustavljanje nakon prve neuspjele provjere validnosti*

`stopOnFirstFailure` metoda će obavijestiti validator da treba prestati provjeravati validnost svih atributa nakon što se dogodi jedna neuspješna provjera validnosti:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

#### *Automatsko preusmjeravanje*

Ako želite ručno kreirati instancu validatora, ali i dalje iskoristiti prednost automatskog preusmjeravanja koje nudi `validate` metoda HTTP zahtjeva, možete pozvati `validate` metodu na postojećoj instanci validatora. Ako provjera validnosti ne uspije, korisnik će automatski biti preusmjeren ili, u slučaju XHR zahtjeva, [bit će vraćen JSON odgovor](#):

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Možete upotrijebiti `validateWithBag` metodu za pohranjivanje poruka o grešci u [imenovanu torbu za greške](#) ako provjera validnosti ne uspije:

```
Validator::make($request->all(), [  
    'title' => 'required|unique:posts|max:255',  
    'body' => 'required',  
)->validateWithBag('post');
```

#### *Imenovane torbe s greškama (engl. Named Error Bags)*

Ako imate više formi na jednoj stranici, možda ćete htjeti imenovati one `MessageBag` koji sadrže validacijske greške, što vam omogućava da dohvatite poruka o greškama za određenu formu. Da biste to postigli, proslijedite ime kao drugi argument u `withErrors`:

```
return redirect('register')->withErrors($validator, 'login');
```

Zatim možete pristupiti imenovanoj `MessageBag` instanci iz `$errors` varijable:

```
{{ $errors->login->first('email') }}
```

#### *Prilagođavanje poruka o grešci*

Ako je potrebno, možete omogućiti prilagođene poruke o grešci koje bi instanca validatora trebala koristiti umjesto zadanih poruka o grešci koje daje Laravel. Postoji nekoliko načina za određivanje prilagođenih poruka. Prvo, možete proslijediti prilagođene poruke kao treći argument `Validator::make` metode:

```
$validator = Validator::make($input, $rules, $messages = [  
    'required' => ':attribute polje je obavezno.',  
)
```

U ovom će primjeru `:attribute` rezervirano mjesto biti zamijenjeno stvarnim nazivom polja koje se validira. Također možete koristiti druga rezervirana mjesta u validacijskim porukama. Na primjer:

```
$messages = [  
    'same' => ':attribute i :other moraju se podudarati.',  
    'size' => ':attribute mora biti točno :size.',  
    'between' => ':attribute vrijednost :input nije između :min - :max.',  
    'in' => ':attribute mora biti jedan od sljedećih tipova: :values',  
)
```

#### *Određivanje prilagođene poruke za dani atribut*

Ponekad ćete možda htjeti navesti prilagođenu poruku o grešci samo za određeni atribut. To možete učiniti pomoću „točka“ notaciju. Prvo navedite naziv atributa, a zatim pravilo:

```
$messages = [  
    'email.required' => 'Moramo znati vašu email adresu!',  
];
```

#### *Određivanje prilagođenih vrijednosti atributa*

Mnoge Laravel-ove ugrađene poruke o greškama uključuju `:attribute` rezervirano mjesto koje je zamijenjeno nazivom polja ili atributa koji provjeravamo. Da biste prilagodili vrijednosti koje se koriste za zamjenu ovih rezerviranih mjesta za određena polja, možete proslijediti matricu prilagođenih atributa kao četvrti argument `Validator::make` metode:

```
$validator = Validator::make($input, $rules, $messages, [  
    'email' => 'email address',  
]);
```

#### *Izvođenje dodatne provjere validacije*

Ponekad morate izvršiti dodatnu provjeru validnosti nakon što je vaša početna provjera validnosti dovršena. To možete postići validatorovom `after` metodom. `after` metoda prihvaća anonimnu funkciju (engl. closure) ili matricu callable tipa koja će se pozvati nakon dovršetka validacije. Zadani callable-ovi će primiti `Illuminate\Validation\Validator` instancu, što vam omogućava dobivanje dodatnih poruka o grešci, ako je to potrebno:

```
use Illuminate\Support\Facades\Validator;  
  
$validator = Validator::make(/* ... */);  
  
$validator->after(function ($validator) {  
    if ($this->somethingElseIsInvalid()) {  
        $validator->errors()->add(  
            'field', 'Nešto nije u redu s ovim poljem!'  
        );  
    }  
});  
  
if ($validator->fails()) {  
    // ...  
}
```

Kao što je navedeno, `after` metoda također prihvaća matricu callables tipa, što je posebno zgodno ako je vaša „nakon validacije“ logika omotana (enkapsulirana) u klase koje je moguće pozvati, a koje će primiti instancu `Illuminate\Validation\Validator` kroz svoje `__invoke` metode:

```
use App\Validation\ValidateShippingTime;  
use App\Validation\ValidateUserStatus;
```

```
$validator->after([
    new ValidateUserStatus,
    new ValidateShippingTime,
    function ($validator) {
        // ...
    },
]);
```

### Rad s validiranim unosom

Nakon validiranja podataka dolaznog zahtjeva korištenjem forme zahtjeva ili ručno stvorene instance validatora, možda ćete poželjeti dohvatiti podatke dolaznog zahtjeva koji su zapravo prošli validaciju. To se može postići na nekoliko načina. Prvo, možete pozvati `validated` metodu na zahtjevu forme ili instanci validatora. Ova metoda vraća matricu podataka koji su validirani:

```
$validated = $request->validated();

$validated = $validator->validated();
```

Alternativno, možete pozvati `safe` metodu na zahtjevu forme ili instancu validatora. Ova metoda vraća instancu `Illuminate\Support\ValidatedInput`. Ovaj objekt izlaže `only`, `except`, i `all` metode za dohvaćanje podskupa validiranih podataka ili cijele matrice validiranih podataka:

```
$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();
```

Osim toga, `Illuminate\Support\ValidatedInput` instanca se može ponavljati i pristupiti joj se kao matrici:

```
// Validirani podaci mogu se ponavljati...
foreach ($request->safe() as $key => $value) {
    // ...
}

// Validiranim podacima može se pristupiti kao matrici...
$validated = $request->safe();

$email = $validated['email'];
```

Ako želite dodati dodatna polja validiranim podacima, možete pozvati `merge` metodu:



```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

Ako želite dohvatiti validirane podatke kao instancu kolekcije, možete pozvati `collect` metodu:

```
$collection = $request->safe()->collect();
```

### Rad s porukama o greškama

Nakon pozivanja `errors` metode na Validator instanci, primit ćete `Illuminate\Support\MessageBag` instancu koja ima razne prikladne metode za rad s porukama o greškama. Varijabla `$errors` koja je automatski dostupna svim pogledima također je instanca `MessageBag` klase.

#### *Dohvaćanje prve poruke o grešci za polje*

Da biste dohvatili prvu poruku o grešci za određeno polje, koristite `first` metodu:

```
$errors = $validator->errors();  
  
echo $errors->first('email');
```

#### *Dohvaćanje svih poruka o grešci za polje*

Ako trebate dohvatiti matricu svih poruka za određeno polje, koristite `get` metodu:

```
foreach ($errors->get('email') as $message) {  
    // ...  
}
```

Ako validirate matricu polja forme, možete dohvatiti sve poruke za svaki od elemenata matrice koristeći znak `*`:

```
foreach ($errors->get('attachments.*') as $message) {  
    // ...  
}
```

#### *Dohvaćanje svih poruka o grešci za sva polja*

Da biste dohvatili matricu svih poruka za sva polja, koristite `all` metodu:

```
foreach ($errors->all() as $message) {  
    // ...  
}
```

### Utvrđivanje postoji li poruka za polje

`has` metoda se može koristiti za utvrđivanje postoje li poruke o grešci za određeno polje:

```
if ($errors->has('email')) {  
    // ...  
}
```

### Određivanje prilagođenih poruka u jezičnim datotekama

Svako od Laravel-ovih ugrađenih pravila provjere validnosti ima poruku o grešci koja se nalazi u `lang/en/validation.php` datoteci vaše aplikacije. Ako vaša aplikacija nema `lang` direktorij, možete uputiti Laravel da ga kreira pomoću `lang:publish` Artisan naredbe.

Unutar `lang/en/validation.php` datoteke pronaći ćete unos prijevoda za svako pravilo provjere validnosti. Možete slobodno promijeniti ili modificirati ove poruke na temelju potreba vaše aplikacije.

Osim toga, možete kopirati ovu datoteku u direktorij drugog jezika kako biste preveli poruke za jezik svoje aplikacije. Kako biste saznali više o Laravel lokalizaciji, pogledajte kompletnu dokumentaciju o lokalizaciji.

#### UPOZORENJE:



Prema zadanim postavkama skelet aplikacije Laravel ne uključuje `lang` direktorij. Ako želite prilagoditi Laravelove jezične datoteke, možete ih objaviti pomoću `lang:publish` Artisan naredbe.

### Prilagođene poruke za određene atribute

Možete prilagoditi poruke o greškama koje se koriste za navedene kombinacije atributa i pravila unutar jezičnih datoteka za provjeru validnosti vaše aplikacije. Da biste to učinili, dodajte svoje prilagodbe u `custom` matricu svoje aplikacije `lang/xx/validation.php` jezične datoteke:

```
'custom' => [  
    'email' => [  
        'required' => 'Moramo znati vašu email adresu!',  
        'max' => 'Vaša email adresa je predugačka!',  
    ],  
],
```

### Određivanje atributa u jezičnim datotekama

Mnoge Laravelove ugrađene poruke o grešci uključuju `:attribute` rezervirano mjesto koje je zamijenjeno nazivom polja ili atributom pod validacijom. Ako želite da se `:attribute` dio vaše poruke provjere validnosti zamijeni prilagođenom vrijednošću, možete navesti naziv prilagođenog atributa u `attributes` matrici vaše `lang/xx/validation.php` jezične datoteke:

```
'attributes' => [  
    'email' => 'email address',  
],
```

**UPOZORENJE:**

Prema zadanim postavkama, skelet aplikacije Laravel ne uključuje `lang` direktorij. Ako želite prilagoditi Laravelove jezične datoteke, možete ih objaviti pomoću `lang:publish` Artisan naredbe.

*Određivanje vrijednosti u jezičnim datotekama*

Neke od Laravel-ovih ugrađenih poruka o grešci validacijskih pravila sadrže `:value` rezervirano mjesto koje je zamijenjeno trenutnom vrijednošću atributa zahtjeva. Međutim, povremeno ćete možda trebati `:value` zamijeniti dio vaše validacijske poruke prilagođenim prikazom vrijednosti. Na primjer, razmotrite sljedeće pravilo koje navodi da je broj kreditne kartice potreban ako ima `payment_type` vrijednost `cc`:

```
Validator::make($request->all(), [  
    'credit_card_number' => 'required_if:payment_type,cc'  
]);
```

Ako ovo validacijsko pravilo ne uspije, proizvest će sljedeću poruku o grešci:

```
The credit card number field is required when payment type is cc.
```

Umjesto prikazivanja `cc` kao vrijednosti vrste plaćanja, možete navesti prikaz vrijednosti koji je lakši za korištenje u vašoj `lang/xx/validation.php` jezičnoj datoteci definiranjem values matrice:

```
'values' => [  
    'payment_type' => [  
        'cc' => 'credit card'  
    ],  
],
```

**UPOZORENJE:**

Prema zadanim postavkama skelet aplikacije Laravel ne uključuje `lang` direktorij. Ako želite prilagoditi Laravelove jezične datoteke, možete ih objaviti pomoću `lang:publish` Artisan naredbe.

Nakon definiranja ove vrijednosti, validacijsko pravilo i proizvest će sljedeću poruku o grešci:

```
The credit card number field is required when payment type is credit card.
```

## Dostupna pravila provjere ispravnosti

<a href="#">Accepted</a>	<a href="#">Exclude If</a>	<a href="#">Not Regex</a>
<a href="#">Accepted If</a>	<a href="#">Exclude Unless</a>	<a href="#">Nullable</a>
<a href="#">Active URL</a>	<a href="#">Exclude With</a>	<a href="#">Numeric</a>
<a href="#">After (Date)</a>	<a href="#">Exclude Without</a>	<a href="#">Present</a>
<a href="#">After Or Equal (Date)</a>	<a href="#">Exist (Database)</a>	<a href="#">Present If</a>
<a href="#">Alpha</a>	<a href="#">Extensions</a>	<a href="#">Present Unless</a>
<a href="#">Alpha Dash</a>	<a href="#">File</a>	<a href="#">Present With</a>
<a href="#">Alpha Numeric</a>	<a href="#">Filled</a>	<a href="#">Present With All</a>
<a href="#">Array</a>	<a href="#">Greater Than</a>	<a href="#">Prohibited</a>
<a href="#">Ascii</a>	<a href="#">Greater Than Or Equal</a>	<a href="#">Prohibited If</a>
<a href="#">Bail</a>	<a href="#">Hex Color</a>	<a href="#">Prohibited Unless</a>
<a href="#">Before (Date)</a>	<a href="#">Image (File)</a>	<a href="#">Prohibits</a>
<a href="#">Before Or Equal (Date)</a>	<a href="#">In</a>	<a href="#">Regular Expression</a>
<a href="#">Between</a>	<a href="#">In Array</a>	<a href="#">Required</a>
<a href="#">Boolean</a>	<a href="#">Integer</a>	<a href="#">Required If</a>
<a href="#">Confirmed</a>	<a href="#">IP Address</a>	<a href="#">Required If Accepted</a>
<a href="#">Current Password</a>	<a href="#">JSON</a>	<a href="#">Required Unless</a>
<a href="#">Date</a>	<a href="#">Less Than</a>	<a href="#">Required With</a>
<a href="#">Date Equals</a>	<a href="#">Less Than Or Equal</a>	<a href="#">Required With All</a>
<a href="#">Date Format</a>	<a href="#">Lowercase</a>	<a href="#">Required Without</a>
<a href="#">Decimal</a>	<a href="#">MAC Address</a>	<a href="#">Required Without All</a>
<a href="#">Declined</a>	<a href="#">Max</a>	<a href="#">Required Array Keys</a>
<a href="#">Declined If</a>	<a href="#">Max Digits</a>	<a href="#">Same</a>
<a href="#">Different</a>	<a href="#">MIME Types</a>	<a href="#">Size</a>
<a href="#">Digits</a>	<a href="#">MIME Type By File Extensions</a>	<a href="#">Sometimes</a>
<a href="#">Digits Between</a>	<a href="#">Min</a>	<a href="#">Starts With</a>
<a href="#">Dimensions (Image Files)</a>	<a href="#">Min Digits</a>	<a href="#">String</a>
<a href="#">Distinct</a>	<a href="#">Missing</a>	<a href="#">Timezone</a>
<a href="#">Doesnt Start With</a>	<a href="#">Missing If</a>	<a href="#">Unique (Database)</a>
<a href="#">Doesnt End With</a>	<a href="#">Missing Unless</a>	<a href="#">Uppercase</a>
<a href="#">Email</a>	<a href="#">Missing With</a>	<a href="#">URL</a>
<a href="#">Ends With</a>	<a href="#">Missing With All</a>	<a href="#">ULID</a>
<a href="#">Enum</a>	<a href="#">Multiple Of</a>	<a href="#">UUID</a>
<a href="#">Exclude</a>	<a href="#">Not In</a>	

*accepted*

Polje koje se provjerava mora biti „yes“, „on“, 1, „1“, true ili „true“. Ovo je korisno za provjeru prihvatanja „Uslovi usluge“ ili sličnih polja.

*accepted If: drugo\_polje, vrijednost,...*

Polje koje se provjerava mora biti „yes“, „on“, 1 ili true ako je drugo polje koje se provjerava jednako navedenoj vrijednosti. Ovo je korisno za provjeru prihvatanja „Uslovi usluge“ ili sličnih polja.

```
[
  'independent_financial_advisor' => 'required|boolean',
```

```
'understand_objective' =>
'required|boolean|accepted_if:independent_financial_advisor,false',
'confirm_objective' =>
'required|boolean|accepted_if:independent_financial_advisor,false',
'understand_term_held' =>
'required|boolean|accepted_if:independent_financial_advisor,false',
'tax_relief' => 'required|boolean|accepted_if:independent_financial_advisor,false',
]
```

#### *active URL*

Polje koje se provjerava mora imati važeći A ili AAAA zapis u skladu s `dns_get_record` PHP funkcijom. Naziv hosta navedenog URL-a izdvaja se pomoću `parse_url` PHP funkcije prije prosljeđivanja u `dns_get_record`.

#### *after:(datum)*

Polje koje se provjerava mora biti vrijednost nakon zadanog datuma. Datumi će biti prosljeđeni u `strtotime` PHP funkciju kako bi se konvertirali u validnu `DateTime` instancu:

```
'start_date' => 'required|date|after:tomorrow'
```

Umjesto prosljeđivanja stringa datuma koji treba procijeniti `strtotime`<sup>51</sup>, možete navesti drugo polje za usporedbu s datumom:

```
'finish_date' => 'required|date|after:start_date'
```

#### *after\_or\_equal:date*

Polje koje se provjerava mora biti vrijednost nakon ili jednaka navedenom datumu. Za više informacija pogledajte [after](#) pravilo.

#### *alpha*

Polje koje se provjerava mora sadržavati u potpunosti Unicode abecedne znakove sadržane u `\p{L}` i `\p{M}`.

Da biste ograničili ovo validacijsko pravilo na znakove u ASCII rasponu (`a-z` i `A-Z`), možete dati `ascii` opciju za validacijsko pravilo:

```
'username' => 'alpha_dash:ascii',
```

---

<sup>51</sup> Ugrađena PHP fukncija koja pretvara datum iz stringa u objekt. Ispravno je `$timestamp1 = strtotime("2024-02-13 13:45:00");` ili je `$timestamp1 = strtotime("+1 week 2 days 4 hours 2 seconds");`. Nakon toga možemo napisati `$timezone = new DateTimeZone("CET");` `$timestamp = strtotime($dateTimeString, 0, $timezone);`

### *alpha Dash*

Polje koje validiramo mora sadržavati isključivo Unicode alfanumeričke znakove sadržane u `\p{L}`, `\p{M}`, `\p{N}`, kao i ASCII crtice (`-`) i ASCII linije podvlačenja (`_`).

Da biste ograničili ovo pravilo provjere validnosti na znakove u ASCII rasponu (`a-z` i `A-Z`), možete dati `ascii` opciju za validacijsko pravilo:

```
'username' => 'alpha_num:ascii',
```

Iako će nam ovo rijetko trebati.

Prije ćemo koristiti

```
'username' => "required|regex:/^[0-9A-Za-z.\s,'-]*$/",
```

```
public function store(Request $request){  
    $this->validate($request, ['username' => 'regex:/^[ A-Za-zšŠđĐžŽčĆć0-9_\-  
]*$/']);  
}
```

Ovakva validacija će prihvatiti prazne nazive korisnika; ako želite prihvatiti samo one koji nisu prazni promijenite `*` u `+`.

### *alpha\_num*

Polje koje se validiramo mora biti u potpunosti sastavljeno od alfanumeričkih znakova Unicode sadržanih u `\p{L}`, `\p{M}`, i `\p{N}`.

Da biste ograničili ovo pravilo validacije na znakove u ASCII rasponu (`a-z` i `A-Z`), možete dati `ascii` opciju kao validacijsko pravilo:

```
'username' => 'alpha_num:ascii',
```

### *array*

Polje koje validiramo mora biti PHP `array`, to jest matrica.

Kada se dodaju dodatne vrijednosti `array` pravilu, svaki ključ u ulaznoj matrici mora biti prisutan na listi vrijednosti danih pravilu. U sljedećem primjeru `admin` ključ u ulaznom polju nije važeći jer nije sadržan na listi vrijednosti danih `array` pravilu:

```
use Illuminate\Support\Facades\Validator;  
  
$input = [  
    'user' => [  
        'name' => 'Taylor Otwell',  
        'username' => 'taylorotwell',  
        'admin' => true,  
    ],  
],
```

```
];  
  
Validator::make($input, [  
    'user' => 'array:name,username',  
]);
```

Općenito, uvijek biste trebali navesti ključeve matrice koji smiju biti prisutni unutar vaše matrice.

### *Ascii*

Polje koje se validira mora sadržavati samo 7-bitne ASCII znakove (ovo baš i nećemo koristiti).

### *bail*

Zaustavite izvođenje validacijskog pravila za polje nakon prve neuspjele provjere validacije.

Dok će `bail` pravilo zaustaviti validaciju određenog polja samo kada naiđe na neuspješnu validaciju, `stopOnFirstFailure` metoda će obavijestiti validator da treba zaustaviti validaciju svih atributa nakon što se dogodi jedna validacijska greška:

```
if ($validator->stopOnFirstFailure()->fails()) {  
    // ...  
}
```

### *before (datum)*

Polje koje se provjerava mora biti vrijednost koja prethodi danom datumu. Datumi će biti proslijeđeni u PHP `strtotime` funkciju kako bi se konvertirali u validnu `DateTime` instancu. Nadalje, kao i `after` pravilo, naziv drugog polja koje se provjerava može se navesti kao vrijednost `date`.

### *before\_or\_equal (datum)*

Polje koje se provjerava mora biti vrijednost koja prethodi danom datumu ili mu je jednaka. Datumi će biti proslijeđeni u PHP `strtotime` funkciju kako bi se pretvorili u validnu `DateTime` instancu. Nadalje, kao i `after` pravilo, naziv drugog polja koje se provjerava može se navesti kao vrijednost `date`.

```
'sleep_date' => ['required', 'before_or_equal:' . Date('Y-m-d')]
```

Možete koristiti laravelov `now()` pomoćnik umjesto `Date()` na isti način:

```
'sleep_date' => ['required', 'before_or_equal:' . now()->format('Y-m-d')]
```

### *between:min, max*

Polje koje se validira mora imati vrijednost između zadane min i max (uključujući). Stringovi, brojevi, matrice i datoteke procjenjuju se na isti način kao `size` pravilo.

```
'check' => 'required|digits_between:9,12'
```

Digits\_between računa dužinu znamenki a ne dužinu stringa. Ako to trebate, koristite [integer](#) ili [numeric](#), po potrebi. Za to je potreban ovakav izraz:

```
'integer|between:9,12'  
// ili  
'numeric|between:9,12'
```

#### *boolean*

Polje koje se provjerava mora se moći pretvoriti tip (engl. cast) u boolean. Prihvatljiv unos je `true`, `false`, `1`, `0`, `"1"` i `"0"`.

#### *confirmed*

Polje koje se provjerava mora imati polje koje se podudara `{field}_confirmation`. Na primjer, ako je polje koje se provjerava `password`, odgovarajuće `password_confirmation` polje mora biti prisutno u unosu.

Pravilo `confirmed` vam dopušta da dvaput zahtijevate određeno polje kako biste provjerili jesu li podaci točni.

#### *current\_password*

Polje koje se provjerava mora odgovarati lozinci autentificiranog korisnika. Možete navesti zaštitu autentifikacije pomoću prvog parametra pravila:

```
'password' => 'current_password:api'
```

#### *date*

Polje koje se provjerava mora biti važeći, ne-relativni datum prema `strtotime` PHP funkciji.

```
$request->validate(  
    'date_field' => 'date'  
);
```

#### *date\_equals:datum*

Polje koje se provjerava mora biti jednako zadanom datumu. Datumi će biti proslijeđeni u PHP `strtotime` funkciju kako bi se pretvorili u validnu `DateTime` instancu.

```
$request->validate(  
    'date_field' => 'date_equals:02/16/2024'  
);
```

Ili određeno vrijeme i datum:

```
$request->validate(  
    'date_field' => 'date_equals:02/16/2024 15:00'  
);
```



### *date\_format:format,...*

Polje koje se provjerava mora odgovarati jednom od zadanih formata . Kada provjeravate validnost polja trebali biste koristiti ili `date` ili `date_format`, a ne oboje. Ovo pravilo validacije podržava sve formate koje podržava PHP [DateTime](#) klasa.

### *decimal:min, max*

Polje koje se provjerava mora biti numeričko i mora sadržavati navedeni broj decimalnih mjesta:

```
// Mora imati točno dvije decimale (9,99)...  
'price' => 'decimal:2'  
  
// Mora imati između 2 i 4 decimalna mjesta...  
'price' => 'decimal:2,4'
```

### *declined*

Polje koje se provjerava mora biti `"no"`, `"off"`, `0`, `"0"`, `false` ili `"false"`.

### *declined\_if: drugopolje, vrijednost,...*

Polje koje se provjerava mora biti `"no"`, `"off"`, `0`, `"0"`, `false` ili `"false"` ili ako je drugo polje koje validiramo jednako navedenoj vrijednosti.

### *different:polje*

Polje koje se provjerava mora imati različitu vrijednost od polje.

### *digits*

Cjelobrojni broj koji se provjerava mora imati točnu dužinu vrijednosti.

### *digits\_between:min,max*

Cjelobrojni broj koji se provjerava mora imati dužinu između zadanog min i max.

### *dimensions*

Datoteka koja se provjerava mora biti slika koja zadovoljava ograničenja dimenzija kako je navedeno u parametrima pravila:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Dostupna ograničenja su: `min_širina` ,`max_širina` ,`min_visina` ,`max_visina` ,`širina` ,`visina` ,`proporcija`.

Ograničenje proporcije treba biti predstavljeno kao širina podijeljena s visinom. To se može specificirati ili razlomkom kao što je `3/2` ili s float brojem kao npr. `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Budući da ovo pravilo zahtijeva nekoliko argumenata, možete koristiti `Rule::dimensions` metodu za tečnu konstrukciju pravila:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
    ],
]);
```

### *distinct*

Kod validacije matrica, polje koje se provjerava ne smije imati duple vrijednosti:

```
'foo.*.id' => 'distinct'
```

Distinct prema zadanim postavkama koristi labave usporedbe varijabli. Da biste koristili stroge usporedbe, možete dodati `strict` parametar svojoj definiciji pravila provjere validnosti:

```
'foo.*.id' => 'distinct:strict'
```

Možete dodati `ignore_case` argumentima pravila validacije možete dodati kako bi ignorirali razlike među malim i velikim slovima:

```
'foo.*.id' => 'distinct:ignore_case'
```

Recimo da trebamo validirati matricu tako da svaki `abc_id` bude jedinstven u matrici ali ne mora biti jedinstven u tablici.

```
$validator = Validator::make($request->all(), [
    'tests.*.abc_id' => 'ne bi trebao biti isti u matrici'
]);
```

To napravimo na sljedeći način:

```
$validator = Validator::make(
    ['products' =>
        ['product_id' => 1, 'quantity' => 5],
        ['product_id' => 1, 'quantity' => 99],
        ['product_id' => 2, 'quantity' => 1],
    ],
    ['products.*.product_id' => 'distinct']
);

dd($validator->passes());
```

*doesn't start with:foo,bar...*

Polje koje se provjerava ne smije početi s jednom od zadanih vrijednosti.

*doesn't end with:foo,bar...*

Polje koje se provjerava ne smije završiti jednom od zadanih vrijednosti.

### email

Polje koje se provjerava mora biti formatirano kao adresa e-pošte. Ovo pravilo validnosti koristi `egulias/email-validator` paket za validiranje email adrese. Prema zadanim postavkama, `RFCValidation` validator se primjenjuje, ali možete primijeniti i druge provjere validacijskih stilova:

```
'email' => 'email:rfc,dns'
```

Gornji primjer primijenit će provjere `RFCValidation` i `DNSCheckValidation`. Evo cjelovite liste stilova validacije koje možete primijeniti:

- `rfc: RFCValidation`
- `strict: NoRFCWarningsValidation`
- `dns: DNSCheckValidation`
- `spooof: SpoofCheckValidation`
- `filter: FilterEmailValidation`
- `filter_unicode: FilterEmailValidation::unicode()`

Validator filter, koji koristi PHP-ovu `filter_var` funkciju, isporučuje se s Laravelom i bio je Laravelovo zadano ponašanje provjere validnosti emaila prije Laravel verzije 5.8.

### UPOZORENJE:



Validatori `dns` i `spooof` zahtijevaju PHP `intl` ekstenziju.

### *ends\_with:foo,bar*

Polje koje se provjerava mora završiti jednom od zadanih vrijednosti.

### *enum*

**Enum** pravilo je klasa temeljena na pravilu validacije koja provjerava sadrži li polje koje se provjerava validnu enum vrijednost. Enum pravilo prihvaća ime enum kao jedini argument konstruktora. Kada validirate osnovne (engl. primitive) vrijednosti, potpomognuti Enum trebao bi biti dostavljen **Enum** pravilu:

```
use App\Enums\ServerStatus;
use Illuminate\Validation\Rule;

$request->validate([
    'status' => [Rule::enum(ServerStatus::class)],
]);
```

Enum pravila **only** i **except** mogu se koristiti za ograničavanje slučajeva enuma koji se smatraju važećim:

```
Rule::enum(ServerStatus::class)
    ->only([ServerStatus::Pending, ServerStatus::Active]);

Rule::enum(ServerStatus::class)
    ->except([ServerStatus::Pending, ServerStatus::Active]);
```

Metoda **when** može se koristiti za uslovnu izmjenu **Enum** pravila:

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\Rule;

Rule::enum(ServerStatus::class)
    ->when(
        Auth::user()->isAdmin(),
        fn ($rule) => $rule->only(...),
        fn ($rule) => $rule->only(...),
    );
```

### *exclude*

Polje koje se provjerava bit će isključeno iz podataka zahtjeva koje vraćaju **validate** i **validated** metode.

### *exclude\_if:drugopolje,vrijednost*

Polje koje se provjerava bit će isključeno iz podataka zahtjeva koje vraćaju `validate` i `validated` ako je polje *drugopolje* jednako *vrijednost*.

Ako je potrebna složena logika uslovnog isključivanja, možete upotrijebiti `Rule::excludeIf` metodu. Ova metoda prihvata boolean ili anonimnu funkciju (engl. closure). Kada se dobije anonimna funkcija (engl. closure), anonimna funkcija (engl. closure) bi trebalo vratiti `true` ili `false` da naznačiti treba li polje koje se provjerava isključiti:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf(fn () => $request->user()->is_admin),
]);
```

### *exclude\_unless:drugopolje, vrijednost*

Polje koje se provjerava bit će isključeno iz podataka zahtjeva koje vraćaju `validate` i `validated` metode osim ako je *drugopolje* jednako *vrijednost*. Ako je *vrijednost* null (`exclude_unless:name,null`), polje koje validiramo će biti isključeno osim ako je polje za uspoređivanje null ili polje za uspoređivanje nedostaje u podacima zahtjeva.

### *exclude\_with:anotherfield*

Polje koje se provjerava bit će isključeno iz podataka zahtjeva koje vraćaju `validate` i `validated` metode ako je prisutno *drugopolje* polje.

### *exclude\_without:anotherfield*

Polje koje se provjerava bit će isključeno iz podataka zahtjeva koje vraćaju metode `validate` i `validated` ako ako nije prisutno *drugopolje* polje.

### *exists:tablica,stupac*

Polje koje se provjerava mora postojati u danoj tablici baze podataka.

### *Osnovna upotreba exist pravila*

```
'state' => 'exists:states'
```

Ako `column` opcija nije navedena, koristit će se naziv polja. Dakle, u ovom slučaju, pravilo će validirati da `states` tablica baze podataka sadrži zapis s `state` vrijednošću stupca koja odgovara vrijednosti atributa zahtjeva `state`.

#### *Određivanje naziva prilagođenog stupca*

Možete eksplicitno navesti naziv stupca baze podataka koji bi trebalo koristiti pravilo provjere tako da ga postavite iza naziva tablice baze podataka:

```
'state' => 'exists:states,abbreviation'
```

Povremeno ćete možda morati navesti određenu vezu s bazom podataka koja će se koristiti za `exists` upit. To možete postići dodavanjem naziva veze ispred naziva tablice:

```
'email' => 'exists:connection.staff,email'
```

Umjesto da direktno specificirate naziv tablice, možete specificirati Eloquent model koji bi se trebao koristiti za određivanje naziva tablice:

```
'user_id' => 'exists:App\Models\User,id'
```

Ako želite prilagoditi upit koji izvršava pravilo validnosti, možete koristiti `Rule` klasu za tečno definiranje pravila. U ovom primjeru također ćemo navesti pravila validnosti kao matricu umjesto upotrebe `|` znaka za njihovo razgraničenje:

```
use Illuminate\Database\Query\Builder;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function (Builder $query) {
            return $query->where('account_id', 1);
        }),
    ],
]);
```

Možete eksplicitno navesti naziv stupca baze podataka koji bi trebao koristiti pravilo koje `exists` generira `Rule::exists` metodom davanjem naziva stupca kao drugog argumenta `exists` metodi:

```
'state' => Rule::exists('states', 'abbreviation'),
```

*extensions:foo,bar,...*

Datoteka koju validiramo mora imati ekstenziju koju je dodijelio korisnik i koja odgovara jednoj od navedenih ekstenzija:

```
'photo' => ['required', 'extensions:jpg,png'],
```

#### UPOZORENJE:



Nikada se ne biste smjeli oslanjati na validiranje datoteke samo pomoću ekstenzije koju je dodijelio korisnik. Ovo se pravilo obično uvijek treba koristiti u kombinaciji s `mimes` ili `mimetypes` pravilima.

*file*

Polje koje se provjerava mora biti uspješno učitana datoteka.

*filed*

Polje koje se provjerava ne smije biti prazno ako postoji.

*gt:field*

Polje koje se provjerava mora biti veće od danog polja ili vrijednosti . Dva polja moraju biti istog tipa. Stringovi, brojevi, matrice i datoteke procjenjuju se prema istim konvencijama kao `size` pravilo.

*gte:field*

Polje koje se provjerava mora biti veće ili jednako od danog polja ili vrijednosti . Dva polja moraju biti istog tipa. Stringovi, brojevi, matrice i datoteke procjenjuju se prema istim konvencijama kao `size` pravilo.

*hex\_color*

Polje koje se provjerava mora sadržavati valjanu vrijednost boje u heksadecimalnom formatu.

*image*

Datoteka koja se provjerava mora biti slika (jpg, jpeg, png, bmp, gif, svg ili webp).

*in:foo,bar*

Polje koje validira mora biti uključeno u danu listu vrijednosti. Budući da ovo pravilo često zahtijeva `implode` matricu, `Rule::in` metoda se može koristiti za tečnu konstrukciju pravila:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
```

```

        'required',
        Rule::in(['prva-zona', 'druga-zona']),
    ],
]);

```

Kada se `in` pravilo kombinira s `array` pravilom, svaka vrijednost u ulaznoj matrici mora biti prisutna unutar liste vrijednosti danih `in` pravilu. U sljedećem primjeru `SJJ` kod aerodroma (Sarajevo) u ulaznoj matrici nije ispravan budući da se ne nalazi na listi aerodroma danih `in` pravilom:

```

use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$input = [
    'airports' => ['BEG', 'SJJ'],
];

Validator::make($input, [
    'airports' => [
        'required',
        'array',
    ],
    'airports.*' => Rule::in(['BEG', 'ZAG']),
]);

```

*in\_array:drugopolje.\**

Polje koje se provjerava mora postojati u vrijednostima *drugopolje*.

Želimo provjeriti valjanost prije spremanja podataka u bazu podataka da li se dani unos podudara s vrijednošću matrice

```

$this->allslots=array('10:00:00', '10:10:00', '10:20:00', '10:30:00', '10:40:00', '10:50:00', '11:00:00',
'11:10:00', '11:20:00', '11:30:00', '11:40:00', '11:50:00', '12:00:00', '12:10:00', '12:20:00', '12:30:00',
'12:40:00', '12:50:00', '13:00:00', '13:10:00', '13:20:00', '13:30:00', '13:40:00', '13:50:00', '14:00:00',
'14:10:00', '14:20:00', '14:30:00', '14:40:00', '14:50:00', '15:00:00', '15:10:00', '15:20:00', '15:30:00',
'15:40:00', '15:50:00', '16:00:00', '16:10:00', '16:20:00', '16:30:00', '16:40:00', '16:50:00');

```

Moguće rješenje može biti ovako sa implode pravilom

```

'field' => 'required|in:' . implode(',', $this->allslots),

```

A moguće je i ovako:

```

$request['allslots'] = $this->allslots;

validate($request, [

```



```
'field' => 'required|in_array:allslots.*',  
]);
```

Validacija `in_array` očekuje da matrica bude jedan od parametara u zahtjevu. Prihvaćeni odgovor neće funkcionirati ako u nizu imate zareze.

### *integer*

Polje koje se provjerava mora biti cijeli broj.

#### **UPOZORENJE:**



Ovo pravilo provjere validnosti ne provjerava je li unos tipa varijable „integer“, samo da je unos tipa koji prihvaća PHP-ovo `FILTER_VALIDATE_INT` pravilo. Ako trebate validirati unos kao broj, koristite ovo pravilo u kombinaciji s `numeric` pravilom za provjeru validnosti.

### *ip*

Polje koje se provjerava mora biti IP adresa.

### *ip4*

Polje koje se provjerava mora biti IPv4 adresa.

### *ip6*

Polje koje se provjerava mora biti IPv6 adresa.

### *json*

Polje koje se provjerava mora biti važeći JSON string.

### *lt:polje*

Polje koje se provjerava mora biti manje od danog *polje*. Dva polja moraju biti istog tipa. Stringovi, brojevi, matrice i datoteke procjenjuju se prema istim konvencijama kao `size` pravilo.

### *lte:polje*

Polje koje se provjerava mora biti manje ili jednako od danog *polje*. Dva polja moraju biti istog tipa. Stringovi, brojevi, matrice i datoteke procjenjuju se prema istim konvencijama kao `size` pravilo.

### *lowercase*

Polje koje se provjerava mora biti napisano malim slovima.

#### *mac\_address*

Polje koje se provjerava mora biti MAC adresa.

#### *max:vrijednost*

Polje koje se provjerava mora biti manje ili jednako maksimalnoj *vrijednost*. Stringovi, brojevi, matrice i datoteke procjenjuju se na isti način kao *size* pravilo.

#### *max\_digits:vrijednost*

Cijeli broj koji se provjerava mora imati maksimalnu duljinu *vrijednost*.

#### *mimetypes:text/plain,...*

Datoteka koja se provjerava mora odgovarati jednoj od danih MIME tipova:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

Da bi se odredio MIME tip učitane datoteke, sadržaj datoteke će se pročitati i radna okolina će pokušati pogoditi MIME tip, koji se može razlikovati od klijentovog MIME tipa.

#### *mimes:foo,bar...*

Datoteka koja se provjerava mora imati MIME tip koja odgovara jednoj od navedenih ekstenzija:

```
'photo' => 'mimes:jpg,bmp,png'
```

Iako samo trebate navesti ekstenzije, ovo pravilo zapravo potvrđuje MIME tip datoteke čitanjem sadržaja datoteke i pogađanjem njezinog MIME tipa. Potpuna lista MIME tipova i njihovih odgovarajućih ekstenzija može se pronaći na sljedećoj lokaciji:

<https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

#### *MIME vrste i ekstenzije*

Validacijsko pravilo ne provjerava slaganje između MIME tipa i ekstenzije koju je korisnik dodijelio datoteci. Na primjer, *mimes:png* pravilo validacije smatralo bi datoteku koja sadrži validan PNG sadržaj validnom PNG slikom, čak i ako je datoteka nazvana *photo.txt*. Ako želite provjeriti ekstenziju datoteke koju je dodijelio korisnik, možete upotrijebiti *extensions* pravilo.

#### *min:vrijednost*

Polje koje se provjerava mora imati minimalnu *vrijednost*. Stringovi, brojevi, matrice i datoteke procjenjuju se na isti način kao *size* pravilo.

#### *min\_digits:vrijednost*

Cijeli broj koji se provjerava mora imati minimalnu dužinu *vrijednost*.

### *missing*

Polje koje se provjerava ne smije biti prisutno u ulaznim podacima.

### *missing\_if:drugopolje,vrijednost,...*

Polje koje se provjerava ne smije biti prisutno ako je polje *drugopolje* jednako bilo kojoj *vrijednost*.

### *missing\_unless:drugopolje,vrijednost*

Polje koje se provjerava ne smije biti prisutno osim ako je polje *drugopolje* jednako bilo kojoj *vrijednost*.

### *missing\_with:foo,bar,...*

Polje koje se provjerava ne smije biti prisutno samo ako je prisutno bilo koje od ostalih navedenih polja.

### *missing\_with\_all:foo,bar,...*

Polje koje se provjerava ne smije biti prisutno samo ako su prisutna sva druga navedena polja.

### *not\_in:foo,bar...*

Polje koje se provjerava ne smije biti uključeno u danu listu vrijednosti. `Rule::notIn` metoda se može koristiti za tečnu konstrukciju pravila:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

### *not\_regex:uzorak*

Polje koje se provjerava ne smije odgovarati danom regularnom izrazu.

Interno, ovo pravilo koristi PHP `preg_match` funkciju. Navedeni uzorak trebao bi se pridržavati istog oblikovanja koje zahtijeva `preg_match` i zato također uključivati validne graničnike. Na primjer: `'email' => 'not_regex:/^.$$/i'`.

#### UPOZORENJE:



Kada koristite `regex` / `not_regex` uzorke, možda će biti potrebno navesti svoja validacijska pravila koristeći matricu umjesto goristeći `|` graničnike, posebno ako regularni izraz sadrži `|` znak.

#### *nullable*

Polje koje se provjerava može biti null.

#### *numeric*

Polje koje se provjerava mora biti [numeric](#).

#### *present*

Polje koje se provjerava mora postojati u ulaznim podacima.

#### *present\_if:drugopolje,vrijednost,...*

Polje koje se provjerava mora biti prisutno ako je polje *drugopolje* jednako bilo kojoj *vrijednost*.

#### *present\_unless:drugopolje,vrijednost*

Polje koje se provjerava mora biti prisutno osim ako polje *drugopolje* nije jednako bilo kojoj *vrijednost*.

#### *present\_with:foo,bar...*

Polje koje se provjerava mora biti prisutno samo ako je prisutno bilo koje od ostalih navedenih polja.

#### *present\_with\_all:foo,bar*

Polje koje se provjerava mora biti prisutno samo ako su prisutna sva druga navedena polja.

#### *prohibited*

Polje koje se provjerava mora nedostajati ili mora biti prazno. Polje je „prazno“ ako zadovoljava jedan od sljedećih kriterija:

- Vrijednost je `null`.
- Vrijednost je prazan string.
- Vrijednost je prazna matrica ili prazan `Countable` objekt.
- Vrijednost je učitana datoteka s praznom putanjom.

#### *prohibited\_if:drugopolje,vrijednost,...*

Polje koje se provjerava mora nedostajati ili mora biti prazno ako je polje *drugopolje* jednako bilo kojoj *vrijednost*. Polje je „prazno“ ako zadovoljava jedan od sljedećih kriterija:

- Vrijednost je `null`.
- Vrijednost je prazan string.
- Vrijednost je prazna matrica ili prazan `Countable` objekt.
- Vrijednost je učitana datoteka s praznom putanjom.

Ako je potrebna složena logika uslovne zabrane, možete upotrijebiti `Rule::prohibitedIf` metodu. Ova metoda prihvaća boolean vrijednost ili anonimnu funkciju (engl. closure). Kada se dobije anonimna funkcija (engl. closure), anonimna funkcija (engl. closure) bi trebala vratiti true ili false kako bi se naznačilo treba li polje koje validiramo biti zabranjeno:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf(fn () => $request->user()->is_admin),
]);
```

#### *prohibited\_unless: drugopolje, vrijednost, ...*

Polje koje se provjerava mora nedostajati ili mora biti prazno osim ako polje *drugopolje* nije jednako bilo kojoj *vrijednost*. Polje je „prazno“ ako zadovoljava jedan od sljedećih kriterija:

- Vrijednost je `null`.
- Vrijednost je prazan string.
- Vrijednost je prazna matrica ili prazan `Countable` objekt.
- Vrijednost je učitana datoteka s praznom putanjom.

#### *prohibits: drugopolje, ...*

Ako polje koje se provjerava ne nedostaje ili je prazno, sva polja u *drugopolje* moraju nedostajati ili biti prazna. Polje je „prazno“ ako zadovoljava jedan od sljedećih kriterija:

- Vrijednost je `null`.
- Vrijednost je prazan string.
- Vrijednost je prazna matrica ili prazan `Countable` objekt.
- Vrijednost je učitana datoteka s praznom putanjom.

#### *regex: uzorak*

Polje koje se provjerava mora odgovarati danom regularnom izrazu.

Interno, ovo pravilo koristi PHP `preg_match` funkciju. Navedeni uzorak trebao bi se pridržavati istog oblikovanja koje zahtijeva `preg_match` i zato također uključivati validne graničnike. Na primjer: `'email' => 'regex:/^.+@.+$/'`.

#### **UPOZORENJE:**



Kada koristite `regex` / `not_regex` uzorke, možda će biti potrebno navesti pravila u matrici umjesto korištenja `|` graničnika, posebno ako regularni izraz sadrži `|` znak.

### *required*

Polje koje se provjerava mora biti prisutno u ulaznim podacima i ne smije biti prazno. Polje je „prazno“ ako zadovoljava jedan od sljedećih kriterija:

- Vrijednost je `null`.
- Vrijednost je prazan string.
- Vrijednost je prazna matrica ili prazan `Countable` objekt.
- Vrijednost je učitana datoteka s praznom putanjom.

### *required\_if:drugopolje,vrijednost,...*

Polje koje se provjerava mora biti prisutno i ne smije biti prazno ako je polje *drugopolje* jednako bilo kojoj *vrijednost*.

Ako želite konstruirati složeniji uvjet za `required_if` pravilo, možete koristiti `Rule::requiredIf` metodu. Ova metoda prihvaća boolean ili anonimnu funkciju (engl. closure). Kada se prođe anonimna funkcija (engl. closure), anonimna funkcija (engl. closure) bi trebala vratiti `true` ili `false` da naznači da li potrebno polje koje se validira:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf(fn () => $request->user()->is_admin),
]);
```

Evo još jednog primjera. Recimo da imamo dropdown meni kao što je ovaj:

```
<select name="selection">
  <option value="1">Option 1</option>
  <option value="2">Option 2</option>
  <option value="3">Option 3</option>
</select>
<input type="text" name="stext">
```

U Laravelu bi ovo bilo loše rješenje jer ako odaberemo opciju 1, `stext` je još uvijek potreban.

```
public static myfunction(){
    $input = \Input::only('selection','stext');
    $rule = array(
        'selection' => 'required',
        'stext' => 'required_if:selection,2,3',
    );
}
```

```
);  
$validate = \Validator::make($input,$rule);  
}
```

Trebali bismo poslati sve vrijednosti kao parametre odvojene zarezom:

```
$rule = array(  
    'selection' => 'required',  
    'stext' => 'required_if:selection,2,3',  
);
```

Na žalost, ni ovo ne radi jer `required_if` validacija prihvaća samo jednu vrijednost istovremeno. Potrebno je promijeniti validaciju na sljedeći način:

```
$rule = array(  
    'selection' => 'required',  
    'stext' => 'required_if:selection,2|required_if:selection,3',  
);
```

#### *required\_if\_accepted:drugopolje,...*

Polje koje se provjerava mora biti prisutno i ne smije biti prazno ako je polje drugopolje jednako „yes“, „on“, 1, „1“, true ili „true“.

#### *required\_unless:drugopolje,vrijednost,...*

Polje koje se provjerava mora biti prisutno i ne smije biti prazno osim ako polje *drugopolje* nije jednako bilo kojoj *vrijednost*. Ovo također znači da drugo polje mora biti prisutno u podacima zahtjeva osim ako je *vrijednost* null. Ako je *vrijednost* null (`required_unless:name,null`), polje pod validacijom bit će potrebno osim ako je polje za usporedbu null ili polje za usporedbu nedostaje u podacima zahtjeva.

#### *required\_with:foo,bar,...*

Polje koje se provjerava mora biti prisutno i ne prazno samo ako je bilo koje od ostalih navedenih polja prisutno i nije prazno.

#### *required\_with\_all:foo,bar,...*

Polje koje se provjerava mora biti prisutno i ne prazno samo ako su sva druga navedena polja prisutna i nisu prazna.

#### *required\_without:foo,bar,...*

Polje koje se provjerava mora biti prisutno i ne mora biti prazno samo kada je neko od ostalih navedenih polja prazno ili nije prisutno.

*required\_without\_all:foo,bar,...*

Polje koje se provjerava mora biti prisutno, a ne prazno samo kada su sva druga navedena polja prazna ili nisu prisutna.

*required\_array\_keys:foo,bar,...*

Polje koje se provjerava mora biti matrica i mora sadržavati barem navedene ključeve.

*same:polje*

Zadano polje mora odgovarati polju koje se provjerava.

*size:vrijednost*

Polje koje se provjerava mora imati veličinu koja odgovara danoj *vrijednost*. Za podatke stringa, *vrijednost* odgovara broju znakova. Za numeričke podatke, *vrijednost* odgovara zadanoj cjelobrojnoj vrijednosti (atribut također mora imati `numeric` ili `integer` pravilo). Za matricu, size odgovara `count` matrice. Za datoteke `size` odgovara veličini datoteke u kilobajtima. Pogledajmo neke primjere:

```
// Validiraj da li string ima točno 12 znakova...  
'title' => 'size:12';  
  
// Validiraj da je navedeni integer veličine 10...  
'seats' => 'integer|size:10';  
  
// Validiraj da li matrica ima točno 5 elemenata...  
'tags' => 'array|size:5';  
  
// Validiraj da li učitana datoteka ima točno 512 kilobajta...  
'image' => 'file|size:512';
```

*starts\_with:foo,bar,...*

Polje koje se provjerava mora započeti jednom od zadanih vrijednosti.

*string*

Polje koje se provjerava mora biti string. Ako želite dopustiti da polje također bude `null`, trebali biste `nullable` polju dodijeliti pravilo.

*timezone*

Polje koje se provjerava mora biti važeći identifikator vremenske zone prema `DateTimeZone::listIdentifiers` metodi.



Argumenti koje prihvaća `DateTimeZone::listIdentifiers` metoda također se mogu dati ovom pravilu provjere validnosti:

```
'timezone' => 'required|timezone:all';

'timezone' => 'required|timezone:Africa';

'timezone' => 'required|timezone:per_country,US';
```

#### *unique:table,column*

Polje koje se provjerava ne smije postojati unutar dane tablice baze podataka.

Određivanje prilagođene tablice/naziva stupca:

Umjesto da direktno navedete naziv tablice, možete navesti model Eloquent koji bi se trebao koristiti za određivanje naziva tablice:

```
'email' => 'unique:App\Models\User,email_address'
```

`column` opcija se može koristiti za određivanje odgovarajućeg stupca baze podataka polja. Ako `column` opcija nije navedena, koristit će se naziv polja koje se provjerava.

```
'email' => 'unique:users,email_address'
```

#### *Određivanje prilagođene veze s bazom podataka*

Povremeno ćete možda trebati postaviti prilagođenu vezu za upite baze podataka koje postavlja Validator. Da biste to postigli, možete dodati naziv veze ispred naziva tablice:

```
'email' => 'unique:connection.users,email_address'
```

#### *Prisiljavanje Unique pravila da ignorira dani ID*

Ponekad ćete možda poželjeti zanemariti dati ID tokom jedinstvene provjere validnosti. Na primjer, razmotrite „ažuriraj profil“ ekran koji uključuje ime korisnika, email adresu i lokaciju. Vjerojatno ćete htjeti provjeriti je li adresa e-pošte jedinstvena. Međutim, ako korisnik promijeni samo polje imena, a ne polje emaila, ne želite da se pojavi greška provjere jer je korisnik već vlasnik dotične email adrese.

Da bismo instruirali validator da zanemari ID korisnika, upotrijebit ćemo `Rule` klasu za tečno definiranje pravila. U ovom primjeru također ćemo navesti pravila provjere validnosti kao matrice umjesto upotrebe `|` znaka za razgraničenje pravila:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

```
],
]);
```

**UPOZORENJE:**

U metodu nikada ne biste smjeli proslijeđivati bilo koji unos zahtjeva koji kontrolira korisnik u `ignore` metodi. Umjesto toga, trebali biste samo proslijediti sistemski generirani jedinstveni ID kao što je ID s automatskim uvećanjem ili UUID iz Eloquent modela instance. Inače će vaša aplikacija biti ranjiva na napad SQL injektiranjem.

Umjesto proslijeđivanja vrijednosti ključa modela `ignore` metodi, također možete proslijediti cijelu instancu modela. Laravel će automatski izdvojiti ključ iz modela:

```
Rule::unique('users')->ignore($user)
```

Ako vaša tablica koristi naziv stupca primarnog ključa koji nije `id`, možete navesti naziv stupca kada poziva `ignore` metodu:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

Prema zadanim postavkama, `unique` pravilo će provjeriti unikatnost stupca koji odgovara nazivu atributa koji se provjerava. Međutim, možete proslijediti drugačiji naziv stupca kao drugi argument `unique` metode:

```
Rule::unique('users', 'email_address')->ignore($user->id)
```

*Dodavanje dodatnih Where klauzula*

Možete navesti dodatne uslove upita prilagođavanjem upita pomoću `where` metode. Na primjer, dodajmo uslov upita koji obuhvaća upit samo za pretraživanje zapisa koji imaju `account_id` vrijednost stupca `1`:

```
'email' => Rule::unique('users')->where(fn (Builder $query) => $query->where('account_id', 1))
```

*uppercase*

Polje koje se provjerava mora biti napisano velikim slovima.

*url*

Polje koje se provjerava mora biti važeći URL.

Ako želite navesti URL protokole koji bi se trebali smatrati važećim, možete proslijediti protokole kao parametre pravila provjere validnosti:

```
'url' => 'url:http,https',
```

```
'game' => 'url:minecraft,steam',
```

#### *ulid*

Polje koje se provjerava mora biti validan [Univerzalno jedinstveni leksikografski sortirani identifikator \(ULID\)](#).

#### *uuid*

Polje koje se provjerava mora biti važeći RFC 4122 (verzija 1, 3, 4 ili 5) univerzalno jedinstveni identifikator (UUID).

### Pravila uslovnog dodavanja

#### *Preskakanje provjere validnosti kada polja imaju određene vrijednosti*

Možda ćete povremeno htjeti ne validirati dano polje ako drugo polje ima danu vrijednost. To možete postići korištenjem `exclude_if` validacijskog pravila. U ovom primjeru, polja `appointment_date` i `doctor_name` neće biti potvrđena ako `has_appointment` polje ima vrijednost `false`:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' => 'exclude_if:has_appointment,false|required|date',
    'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);
```

Alternativno, možete koristiti `exclude_unless` pravilo da ne potvrdite dano polje osim ako drugo polje ima danu vrijednost:

```
$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' => 'exclude_unless:has_appointment,true|required|date',
    'doctor_name' => 'exclude_unless:has_appointment,true|required|string',
]);
```

#### *Validacija kada je prisutna*

U nekim situacijama možda ćete htjeti pokrenuti provjere validnosti za polje samo ako je to polje prisutno u podacima koji se validiraju. Da biste to brzo postigli, dodajte `sometimes` pravilo na svoju listu pravila:

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

U primjeru iznad, `email` polje će biti validirano samo ako je prisutno u `$data` matrici.

#### UPOZORENJE:



Ako pokušavate validirati polje koje bi uvijek trebalo biti prisutno, ali može biti prazno, pogledajte ovu [napomenu o opcionalnim poljima](#).

#### Složena uslovna validacija

Ponekad ćete možda poželjeti dodati pravila provjere bazirana na složenijoj uvjetnoj logici. Na primjer, možda ćete htjeti zahtijevati određeno polje samo ako drugo polje ima vrijednost veću od 100. Ili ćete možda trebati da dva polja imaju danu vrijednost samo kada je prisutno drugo polje. Dodavanje ovih pravila provjere validnosti ne mora biti mučno. Prvo stvorite `Validator` instancu sa svojim *statičkim pravilima* koja se nikada ne mijenjaju:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Pretpostavimo da je naša web aplikacija za kolekcionare igara. Ako se kolekcionar igara registrira na našu aplikaciju i posjeduje više od 100 igara, želimo da nam objasni zašto posjeduje toliko igara. Na primjer, možda vodi trgovinu za preprodaju igara ili možda samo uživa biti kolekcionar. Za uslovno dodavanje ovog zahtjeva, možemo koristiti `sometimes` metodu na `Validator` instanci.

```
use Illuminate\Support\Fluent;

$validator->sometimes('reason', 'required|max:500', function (Fluent $input) {
    return $input->games >= 100;
});
```

Prvi argument proslijeđen metodi `sometimes` je naziv polja koje uslovno provjeravamo. Drugi argument je popis pravila koja želimo dodati. Ako je anonimna funkcija (engl. closure) proslijeđena treći argument vraća `true`, pravila će biti dodana. Ova metoda olakšava izradu složenih uslovnih validacija. Možete čak dodati uslovne provjere za nekoliko polja odjednom:

```
$validator->sometimes(['reason', 'cost'], 'required', function (Fluent $input) {
    return $input->games >= 100;
});
```

#### NAPOMENA:



`$input` parametar proslijeđen vašoj anonimna funkcija (engl. closure) bit će `Illuminate\Support\Fluent` instanca i može se koristiti za pristup vašem unosu i datotekama koje se provjeravaju.

*Složena uslovna validacija matrice*

Ponekad ćete možda htjeti potvrditi polje na temelju drugog polja u istom ugniježđenoj matrici čiji indeks ne znate. U ovim situacijama možete dopustiti da vaša anonimna funkcija (engl. closure) primi drugi argument koji će biti trenutna pojedinačna stavka u matrici koji se validira:

```
$input = [  
  'channels' => [  
    [  
      'type' => 'email',  
      'address' => 'abigail@example.com',  
    ],  
    [  
      'type' => 'url',  
      'address' => 'https://example.com',  
    ],  
  ],  
];  
  
$validator->sometimes('channels.*.address', 'email', function (Fluent $input,  
Fluent $item) {  
  return $item->type === 'email';  
});  
  
$validator->sometimes('channels.*.address', 'url', function (Fluent $input, Fluent  
$item) {  
  return $item->type !== 'email';  
});
```

Kao i `$input` parametar koji se prosljeđuje zatvaranju, `$item` parametar je instanca `Illuminate\Support\Fluent` kada su podaci atributa matrica; inače, to je string.

*Validacija matrice*

Kao što je objašnjeno u [array dokumentaciji pravila validnosti](#), `array` pravilo prihvaća listu dopuštenih ključeva matrice. Ako postoje dodatni ključevi unutar matrice, provjera validnosti neće uspjeti:

```
use Illuminate\Support\Facades\Validator;  
  
$input = [  
  'user' => [  
    'name' => 'Taylor Otwell',  
    'username' => 'taylorotwell',  
    'admin' => true,  
  ],  
];
```

```
Validator::make($input, [  
    'user' => 'array:name,username',  
]);
```

Općenito, uvijek biste trebali navesti ključeve niza koji smiju biti prisutni unutar vaše matrice. Inače će validatorove `validate` i `validated` metode vratiti sve validirane podatke, uključujući matricu i sve njegove ključeve, čak i ako ti ključevi nisu potvrđeni drugim pravilima validnosti ugniježdene matrice.

#### *Validacija unosa ugniježđenog polja*

Provjera validnosti matrice za unos forme na temelju ugniježdene matrice ne mora biti mučna. Možete koristiti „točka notaciju“ za validiranje atributa unutar matrice. Na primjer, ako dolazni HTTP zahtjev sadrži `photos[profile]` polje, možete ga potvrditi ovako:

```
use Illuminate\Support\Facades\Validator;  
  
$validator = Validator::make($request->all(), [  
    'photos.profile' => 'required|image',  
]);
```

Također možete validirati svaki element matrice. Na primjer, da biste validirali da je svaki email u danoj matrici za unos jedinstvena, možete učiniti sljedeće:

```
$validator = Validator::make($request->all(), [  
    'person.*.email' => 'email|unique:users',  
    'person.*.first_name' => 'required_with:person.*.last_name',  
]);
```

Isto tako, možete koristiti `*` znak kada specificirate prilagođene [validacijske poruke u svojim jezičnim datotekama](#), što olakšava korištenje jedne validacijske poruke za matricu temeljena na poljima:

```
'custom' => [  
    'person.*.email' => [  
        'unique' => ' Svaka osoba mora imati jedinstvenu adresu e-pošte',  
    ]  
],
```

#### *Pristup podacima ugniježdene matrice*

Ponekad ćete možda trebati pristupiti vrijednosti za određeni ugniježđeni element matrice kada atributu dodjeljujete pravila provjere validnosti. To možete postići pomoću ove `Rule::forEach` metode. Metoda `forEach` prihvaća anonimnu funkciju (engl. closure) koje će se pozvati za svaku iteraciju atributa polja pod validnošću i primit će vrijednost atributa i eksplicitan, potpuno prošireni

naziv atributa. Anonimna funkcija (engl. closure) bi trebala vratiti matricu pravila za dodjelu elementu matrice:

```
use App\Rules\HasPermission;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$validator = Validator::make($request->all(), [
    'companies.*.id' => Rule::forEach(function (string|null $value, string
$attribute) {
        return [
            Rule::exists(Company::class, 'id'),
            new HasPermission('manage-company', $value),
        ];
    }
]),
]);
```

#### *Indeksi i pozicije poruka o grešci*

Kada validirate matrice, možda ćete željeti referencu indeksa ili pozicije određene stavke koja nije prošla provjeru unutar poruke o grešci koju prikazuje vaša aplikacija. Da biste to postigli, možete uključiti rezervirana mjesta `:indeks` (počinje od `0`) i `:position` (počinje od `1`) unutar svoje [prilagođene validacijske poruke](#):

```
use Illuminate\Support\Facades\Validator;

$input = [
    'photos' => [
        [
            'name' => 'OdmorPlaža.jpg',
            'description' => 'Fotografija mog odmora na plaži!',
        ],
        [
            'name' => 'KorčulanskaMoreška.jpg',
            'description' => '',
        ],
    ],
];

Validator::validate($input, [
    'photos.*.description' => 'required',
], [
    'photos.*.description.required' => 'Molim opišite fotografiju #:position.',
]);
```

S obzirom na gornji primjer, provjera valjanosti neće uspjeti i korisniku će se prikazati sljedeća greška „Molimo opišite fotografiju #2.“

Ako je potrebno, možete referencirati dublje ugniježdene indekse i pozicije pomoću `second-index`, `second-position`, `third-index`, `third-position`, itd.

```
'photos.*.attributes.*.string' => 'Nevažeći atribut za fotografiju #:second-position.'
```

### Validacija datoteka

Laravel pruža niz validacijskih pravila koja se mogu koristiti za validiranje učitanih datoteka, kao što su `mimes`, `image`, `min` i `max`. Dok možete slobodno specificirati ova pravila pojedinačno kada provjeravate validnost datoteka, Laravel također nudi tečan alat za izradu pravila za provjeru validnosti datoteke koji bi vam mogao biti prikladan:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'attachment' => [
        'required',
        File::types(['mp3', 'wav'])
            ->min(1024)
            ->max(12 * 1024),
    ],
]);
```

Ako vaša aplikacija prihvaća slike koje su postavili vaši korisnici, možete koristiti `File` pravila metodu `image` konstruktora da naznačite da bi učitana datoteka trebala biti slika. Osim toga, `dimensions` pravilo se može koristiti za ograničavanje dimenzija slike:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;
use Illuminate\Validation\Rules\File;

Validator::validate($input, [
    'photo' => [
        'required',
        File::image()
            ->min(1024)
            ->max(12 * 1024)
            ->dimensions(Rule::dimensions()->maxWidth(1000)->maxHeight(500)),
    ],
]);
```



```
]);
```

**NAPOMENA:**

Više informacija o potvrđivanju dimenzija slike možete pronaći u [dokumentaciji pravila o dimenzijama](#).

*Veličina datoteka*

Radi praktičnosti, minimalna i najveća veličina datoteke mogu se navesti kao stringovi sa sufiksom koji označava jedinice veličine datoteke. Podržani su sufiksi `kb`, `mb`, `gb` i `tb`:

```
File::image()  
    ->min('1kb')  
    ->max('10mb')
```

*Tipovi datoteka*

Iako trebate samo navesti ekstenzije kada pozivate `types` metodu, ova metoda zapravo validira MIME tip datoteke čitanjem sadržaja datoteke i pogađanjem njenog MIME tipa. Potpuna lista MIME tipova i njihovih odgovarajućih ekstenzija može se pronaći na sljedećoj lokaciji:

<https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types>

## Validacija šifara

Kako biste osigurali da šifre imaju odgovarajući nivo složenosti, možete koristiti Laravelov `Password` objekt pravila:

```
use Illuminate\Support\Facades\Validator;  
use Illuminate\Validation\Rules>Password;  
  
$validator = Validator::make($request->all(), [  
    'password' => ['required', 'confirmed', Password::min(8)],  
]);
```

`password` objekt pravila omogućava vam jednostavno prilagođavanje zahtjeva složenosti šifre za vašu aplikaciju, kao što je navođenje da šifre zahtijevaju najmanje jedno slovo, broj, simbol ili znakove s mješovitim malim i malim slovima:

```
// Zahtijevaj najmanje 8 znakova...  
Password::min(8)  
// Zahtijevaj najmanje jedno slovo...  
Password::min(8)->letters()  
// Zahtijevaj barem jedno veliko i jedno malo slovo...  
Password::min(8)->mixedCase()  
// Zahtijevaj barem jedan broj...
```

```
Password::min(8)->numbers()  
// Zahtijevaj najmanje jedan simbol...  
Password::min(8)->symbols()
```

Osim toga, možete osigurati da šifra nije ugrožena u curenju podataka o javnoj šifri korištenjem metode `uncompromised`:

```
Password::min(8)->uncompromised()
```

Interno, `Password` objekt pravila koristi [model k-anonimnosti](#) kako bi utvrdio da li je šifra procurila pomoću usluge [haveibeenpwned.com](https://haveibeenpwned.com) bez žrtvovanja privatnosti ili sigurnosti korisnika.

Prema zadanim postavkama, ako se šifra pojavi barem jednom u curenju podataka, smatrat će se da je ugrožena. Ovaj prag možete prilagoditi pomoću prvog argumenta metode `uncompromised`:

```
// Osigurajte da se šifra pojavljuje manje od 3 puta u istom curenju podataka...  
Password::min(8)->uncompromised(3);
```

Naravno, možete lančano povezati sve metode u gornjim primjerima:

```
Password::min(8)  
  ->letters()  
  ->mixedCase()  
  ->numbers()  
  ->symbols()  
  ->uncompromised()
```

### *Definiranje podrazumijevanih pravila za šifre*

Možda će vam biti zgodno navesti podrazumijevana pravila validacije za šifre na jednom mjestu vaše aplikacije. To možete lako postići koristeći `Password::defaults` metodu koja prihvaća anonimnu funkciju (engl. closure). Anonimna funkcija (engl. closure) dana `defaults` metodi trebalo bi vratiti podrazumijevanu konfiguraciju pravila za šifru. Obično `defaults` pravilo se treba pozvati unutar `boot` metode jednog od pružatelja usluga vaše aplikacije:

```
use Illuminate\Validation\Rules\Password;  
  
/**  
 * Pokrenite (engl. Bootstrap) bilo koju aplikacijsku uslugu.  
 */  
public function boot(): void  
{  
    Password::defaults(function () {  
        $rule = Password::min(8);  
    });  
}
```

```
        return $this->app->isProduction()
            ? $rule->mixedCase()->uncompromised()
            : $rule;
    });
}
```

Zatim, kada želite primijeniti podrazumijevana pravila na određenu šifru koju validirate, možete pozvati metodu `defaults` bez argumenata:

```
'password' => ['required', Password::defaults()],
```

Povremeno ćete možda htjeti priložiti dodatna pravila validacije vašim podrazumijevanim pravilima validacije šifre. Možete upotrijebiti `rules` metodu da to postignete:

```
use App\Rules\ZxcvbnRule;

Password::defaults(function () {
    $rule = Password::min(8)->rules([new ZxcvbnRule]);

    // ...
});
```

## Prilagođena pravila validnosti

### *Korištenje objekata pravila*

Laravel pruža niz korisnih pravila validnosti; međutim, možda ćete htjeti navesti neka svoja. Jedna od metoda registracije prilagođenih pravila provjere je korištenje objekata pravila. Za generiranje novog objekta pravila, možete koristiti `make:rule` Artisan naredbu. Upotrijebimo ovu naredbu za generiranje pravila koje provjerava da li je string napisan velikim slovima. Laravel će postaviti novo pravilo u `app/Rules` direktorij. Ako ovaj direktorij ne postoji, Laravel će ga kreirati kada izvršite Artisan naredbu za kreiranje vašeg pravila:

```
php artisan make:rule Uppercase
```

Nakon što je pravilo kreirano, spremni smo definirati njegovo ponašanje. Objekt pravila sadrži jednu metodu: `validate`. Ova metoda prima naziv atributa, njegovu vrijednost i funkciju koje se prosljeđuje kao argument drugoj funkciji (engl. callback) koji bi se trebao pozvati u slučaju neuspjeha s porukom o validacijskoj grešci:

```
<?php

namespace App\Rules;

use Closure;
```

```
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements ValidationRule
{
    /**
     * Pokreni pravilo validacije.
     */
    public function validate(string $attribute, mixed $value, Closure $fail): void
    {
        if (strtoupper($value) !== $value) {
            $fail(':attribute moraju biti velika slova.');
```

Nakon što je pravilo definirano, možete ga priložiti validatoru proslijeđivanjem instance objekta pravila s drugim validacijskim pravilima:

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
]);
```

#### *Prevođenje poruka validnosti*

Umjesto osiguravanja doslovne poruke o grešci na `$fail` funkciju koje se proslijeđuje kao argument drugoj funkciji (engl. callback), također možete dati ključ prijevoda stringa i uputiti Laravel da prevede poruku o grešci:

```
if (strtoupper($value) !== $value) {
    $fail('validation.uppercase')->translate();
}
```

Ako je potrebno, možete dati zamjene rezerviranih mjesta i preferirani jezik kao prvi i drugi argument `translate` metode:

```
$fail('validation.location')->translate([
    'value' => $this->value,
], 'fr')
```

### Pristup dodatnim podacima

Ako vaša prilagođena klasa validacijskih pravila mora pristupiti svim drugim podacima u postupku validacije, vaša klasa pravila može implementirati interface `Illuminate\Contracts\Validation\DataAwareRule`. Ovaj interface zahtijeva da vaša klasa definira `setData` metodu. Ovu će metodu automatski pozvati Laravel (prije nego što se validacija nastavi) sa svim podacima koji se validiraju:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\DataAwareRule;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements DataAwareRule, ValidationRule
{
    /**
     * Svi podaci koji se validiraju.
     *
     * @var array<string, mixed>
     */
    protected $data = [];

    // ...

    /**
     * Postavlja podatke koji se validiraju.
     *
     * @param array<string, mixed> $data
     */
    public function setData(array $data): static
    {
        $this->data = $data;

        return $this;
    }
}
```

Ili, ako vaše pravilo provjere validnosti zahtijeva pristup instanci validatora koja provodi validaciju, možete implementirati `ValidatorAwareRule` interface:

```
<?php

namespace App\Rules;
```

```
use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
use Illuminate\Validation\Validator;

class Uppercase implements ValidationRule, ValidatorAwareRule
{
    /**
     * Instanca validatora.
     *
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...

    /**
     * Postavi tekući validator.
     */
    public function setValidator(Validator $validator): static
    {
        $this->validator = $validator;

        return $this;
    }
}
```

#### *Korištenje anonimnih funkcija (engl. closures)*

Ako vam je funkcija prilagođenog pravila potrebna samo jednom u vašoj aplikaciji, možete koristiti anonimnu funkciju (engl. closure) umjesto objekta pravila. Anonimna funkcija (engl. closure) prima naziv atributa, vrijednost atributa i `$fail` funkciju koje se proslijeđuje kao argument drugoj funkciji (engl. callback) koji treba pozvati ako provjera validnosti ne uspije:

```
use Illuminate\Support\Facades\Validator;
use Closure;

$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function (string $attribute, mixed $value, Closure $fail) {
            if ($value === 'foo') {
                $fail("{$attribute} nije važeći.");
            }
        },
    ],
],
```

```
],  
]);
```

### *Implicitna pravila*

Prema zadanim postavkama, kada atribut koji se provjerava nije prisutan ili sadrži prazan string, uobičajena pravila validnosti, uključujući prilagođena pravila, se ne pokreću. Na primjer, `unique` pravilo se neće pokrenuti za prazan string:

```
use Illuminate\Support\Facades\Validator;  
  
$rules = ['name' => 'unique:users,name'];  
  
$input = ['name' => ''];  
  
Validator::make($input, $rules)->passes(); // true
```

Da bi se prilagođeno pravilo pokrenulo čak i kada je atribut prazan, pravilo mora podrazumijevati da je atribut obavezan. Za brzo generiranje novog implicitnog objekta pravila, možete koristiti `make:rule` Artisan naredbu s `-implicit` opcijom:

```
php artisan make:rule Uppercase --implicit
```

### **UPOZORENJE:**



„Implicitno“ pravilo samo *implicira* da je atribut potreban. Hoće li doista staviti van snage atribut koji nedostaje ili koji je prazan ovisi o vama.

## Upravljački sklop za rukovanje greškama (engl. Error Handling)

### Uvod

Kada pokrenete novi Laravel projekt, rukovanje greškama i iznimkama već je konfigurirano za vas. `App\Exceptions\Handler` klasa je mjesto gdje se bilježe sve iznimke koje izbaci vaša aplikacija i zatim prikazuju korisniku. Zaronit ćemo dublje u ovu klasu kroz ovu dokumentaciju.

### Konfiguracija

`debug` opcija u vašoj `config/app.php` konfiguracijskoj datoteci određuje koliko se informacija o grešci zapravo prikazuje korisniku. Prema zadanim postavkama, ova je opcija postavljena tako da poštuje vrijednost varijable `APP_DEBUG` okoline koja je pohranjena u vašoj `.env` datoteci.

Tokom lokalnog razvoja trebali biste postaviti `APP_DEBUG` varijablu okoline na `true`. U vašem proizvodnom okruženju ova bi vrijednost uvijek trebala biti `false`. Ako je vrijednost postavljena na `true` u produkciji, rizikujete izlaganje osjetljivih konfiguracijskih vrijednosti krajnjim korisnicima vaše aplikacije.

## Upravljački sklop za rukovanje iznimkama (engl. Handling Exception)

### Izveštavanje o iznimkama

Sve iznimke obrađuje klasa `App\Exceptions\Handler`. Ova klasa sadrži `register` metodu u kojoj možete registrirati prilagođeno izvještavanje o iznimkama i prikazivanje funkcija koja se proslijeđujete kao argument drugim funkcijama (eng. callbacks). Detaljno ćemo ispitati svaki od ovih koncepata. Izvještavanja o iznimkama koristi se za bilježenje iznimaka (engl. log exceptions) ili njihovo slanje vanjskoj usluzi kao što su [Flare](#), [Bugsnag](#) ili [Sentry](#). Prema zadanim postavkama iznimke će se bilježiti na temelju vaše konfiguracije bilježenja. Međutim, slobodni ste prijaviti iznimke kako god želite.

Ako trebate prijaviti različite vrste iznimaka na različite načine, možete upotrijebiti `reportable` metodu za registraciju zatvaranja koje bi se trebalo izvršiti kada je potrebno prijaviti iznimku određene vrste. Laravel će utvrditi koji tip iznimke anonimna funkcija (engl. closure) prijavljuje nagovještavanjem tipa (engl. type-hint) anonimne funkcije (engl. closure):

```
->withExceptions(function (Exceptions $exceptions) {
    $exceptions->report(function (InvalidOrderException $e) {
        // ...
    });
})
```

Kada registrirate prilagođenu funkciju koja se proslijeđujete kao argument drugim funkcijama (eng. callbacks) za izvještavanje o iznimkama pomoću `reportable` metode, Laravel će i dalje bilježiti iznimku (engl. exception) koristeći zadanu konfiguraciju bilježenja (engl. logging configuration) za aplikaciju. Ako želite zaustaviti širenje iznimke na zadani stek logiranja, možete upotrijebiti `stop` metodu kada definirate funkcije koja se proslijeđujete kao argument (eng. callbacks) izvještavanju ili vraća `false` iz funkcije koja se proslijeđujete kao argument drugim funkcijama (eng. callback):

```
$this->reportable(function (InvalidOrderException $e) {
    // ...
})
```



```
}->stop();

$this->reportable(function (InvalidOrderException $e) {
    return false;
});
```

#### NAPOMENA:



Da biste prilagodili izvještavanje o iznimkama za određenu iznimku, također možete upotrijebiti [iznimke koje je moguće prijaviti](#).

#### Globalni kontekst dnevnika

Ako je dostupno, Laravel automatski dodaje ID trenutnog korisnika svakoj poruci log iznimke kao kontekstualni podatak. Možete definirati vlastite globalne kontekstualne podatke definiranjem `context` metode iznimke u `bootstrap/app.php` datoteci. Ove informacije bit će uključene u svaku poruku log iznimke koju je napisala vaša aplikacija:

```
/**
 * Dobij podrazumijevane varijable konteksta za log evidentiranje.
 *
 * @return array<string, mixed>
 */
protected function context(): array
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

#### Kontekst log izuzetaka

Iako dodavanje konteksta svakoj log poruci može biti korisno, ponekad određena iznimka može imati jedinstven kontekst koji želite uključiti u svoje log-ove. Definiranjem `context` metode na jednoj od iznimki vaše aplikacije, možete navesti sve podatke relevantne za tu iznimku koje treba dodati u log iznimke:

```
<?php

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception
{
    // ...
}
```

```
/**
 * Dohvati informacije o kontekstu iznimke.
 *
 * @return array<string, mixed>
 */
public function context(): array
{
    return ['order_id' => $this->orderId];
}
}
```

### *report pomoćnik*

Ponekad ćete možda trebati prijaviti iznimku, ali nastaviti s obradom trenutnog zahtjeva. `report` pomoćnik (engl. helper) funkcija vam omogućava brzu prijavu iznimke pomoću rukovatelja iznimkama bez prikazivanja stranice s greškom korisniku:

```
public function isValid(string $value): bool
{
    try {
        // Validiraj vrijednost...
    } catch (Throwable $e) {
        report($e);

        return false;
    }
}
```

### *Uklanjanje dvostruko prijavljenih iznimaka*

Ako koristite ovu `report` funkciju u cijeloj svojoj aplikaciji, možete povremeno prijaviti istu iznimku više puta, stvarajući dvostruke unose u svojim zapisima.

Ako želite osigurati da se jedna instanca iznimke prijavljuje samo jednom, možete postaviti `$withoutDuplicates` svojstvo iznimke u datoteci `bootstrap/app.php` svoje aplikacije:

```
namespace App\Exceptions;

use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    /**
     * Označava da se instanca iznimke treba prijaviti samo jednom.
     *
     * @var bool
     */
}
```

```
*/  
protected $withoutDuplicates = true;  
  
// ...  
}
```

Sada, kada `report` pomoćnik se pozove s istom instancom iznimke, prijavit će se samo prvi poziv:

```
$original = new RuntimeException('Ups!');  
  
report($original); // prijavljen  
  
try {  
    throw $original;  
} catch (Throwable $caught) {  
    report($caught); // ignorirano  
}  
  
report($original); // ignorirano  
report($caught); // ignorirano
```

#### *Nivoi zapisnika izuzetaka*

Kada se poruke zapisuju u log vaše aplikacije, poruke se zapisuju na određenom nivou log-a, koji ukazuje na ozbiljnost ili važnost poruke koja se bilježi.

Kao što je iznad navedeno, čak i kada registrirate prilagođenu funkciju koja se prosleđujete kao argument drugim funkcijama (eng. callback) za izvještavanje o iznimci koristeći `reportable` metodu, Laravel će i dalje puniti log iznimkama koristeći zadanu konfiguraciju zapisivanja za aplikaciju; međutim, budući da log nivo ponekad može utjecati na kanale na kojima se poruka bilježi, možda ćete poželjeti konfigurirati log nivo na kojoj se bilježe određene iznimke.

Da biste to postigli, možete definirati `$levels` svojstvo na upravljačkom sklopu za rukovanje iznimkama (engl. exception handler) vaše aplikacije. Ovo svojstvo treba sadržavati matricu tipova iznimaka i njima pridružene log nivoe:

```
use PDOException;  
use Psr\Log\LogLevel;  
  
/**  
 * Lista tipova izuzetaka s odgovarajućim prilagođenim log nivoima.  
 *  
 * @var array<class-string<\Throwable>, \Psr\Log\LogLevel::*>  
 */  
protected $levels = [  
    PDOException::class => LogLevel::CRITICAL,  
];
```

### *Ignoriranje iznimaka prema tipu*

Kada izrađujete vaše aplikacije postojat će neki tipovi iznimaka koje nikada ne želite prijaviti. Da biste zanemarili ove iznimke, definirajte `$dontReport` svojstvo na upravljačkom sklopu za rukovanje iznimkama (engl. exception handler) svoje aplikacije. Sve klase koje dodate ovom svojstvu nikada neće biti prijavljene; međutim, još uvijek mogu imati prilagođenu logiku prikazivanja:

```
use App\Exceptions\InvalidOrderException;

/**
 * Lista tipova izuzetaka koji se ne prijavljuju.
 *
 * @var array<int, class-string<\Throwable>>
 */
protected $dontReport = [
    InvalidOrderException::class,
];
```

Alternativno, možete jednostavno "označiti" klasu iznimke sa interface-om `Illuminate\Contracts\Debug\ShouldntReport`. Kada je iznimka označena ovim interface-om, Laravelov rukovatelj iznimkama (engl. exception handler) je nikada neće prijaviti:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Contracts\Debug\ShouldntReport;

class PodcastProcessingException extends Exception implements ShouldntReport
{
    //
}
```

Interno, Laravel već zanemaruje neke vrste grešaka umjesto vas, kao što su iznimke koje proizlaze iz 404 HTTP greške ili 419 HTTP odgovora generiranih nevažećim CSRF tokenima. Ako želite uputiti Laravel da prestane ignorirati određenu vrstu iznimke, možete pozvati `stopIgnoring` metodu unutar vaše `bootstrap/app.php` datoteke:

```
use Symfony\Component\HttpKernel\Exception\HttpException;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->stopIgnoring(HttpException::class);
})
```

### *Iznimke kod prikazivanja*

Prema zadanim postavkama, Laravel upravljački sklopa za rukovanje iznimkama (engl. exception handler) pretvorit će iznimke u HTTP odgovor za vas. Međutim, slobodni ste registrirati prilagođeno prikazivanje anonimnom funkcijom (engl. closure) za iznimke određenog tipa. To možete postići pozivanjem `renderable` metode unutar vašeg sklopa za rukovanje iznimkama (engl. exception handler).

Anonimna funkcija (engl. closure) koja se prosleđuje `renderable` metodi treba vratiti `Illuminate\Http\Response` instancu, koja se može generirati pomoću `response` pomoćnika. Laravel će odrediti koji tip iznimke anonimna funkcija (engl. closure) prikazuje ispitivanjem nagovještavanjem tipa (engl. type-hint) anonimne funkcije (engl. closure):

```
use App\Exceptions\InvalidOrderException;
use Illuminate\Http\Request;

/**
 * Registrirajte callback-ove za rukovanje iznimkama za aplikaciju.
 */
public function register(): void
{
    $this->renderable(function (InvalidOrderException $e, Request $request) {
        return response()->view('errors.invalid-order', [], 500);
    });
}
```

Također možete koristiti `renderable` metodu za nadjačavanje prikazivanja ponašanja za ugrađene Laravel ili Symfony iznimke kao što su `NotFoundHttpException`. Ako anonimna funkcija (engl. closure) dana `renderable` metodi ne vrati vrijednost, koristit će se Laravelovo podrazumijevano prikazivanje iznimke:

```
use Illuminate\Http\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

/**
 * Registrirajte callback-ove za rukovanje iznimkama za aplikaciju.
 */
public function register(): void
{
    $this->renderable(function (NotFoundHttpException $e, Request $request) {
        if ($request->is('api/*')) {
            return response()->json([
                'message' => 'Slog nije pronađen.'
            ], 404);
        }
    });
}
```

### *Prikazivanje iznimki kao JSON*

Prilikom prikazivanja iznimke, Laravel će automatski odrediti treba li se iznimka prikazati kao HTML ili JSON odgovor na temelju Accept zaglavlja zahtjeva. Ako želite prilagoditi način na koji Laravel određuje hoće li prikazati HTML ili JSON izuzetne odgovore, možete upotrijebiti metodu `shouldRenderJsonWhen`:

```
use Illuminate\Http\Request;
use Throwable;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->shouldRenderJsonWhen(function (Request $request, Throwable $e) {
        if ($request->is('admin/*')) {
            return true;
        }

        return $request->expectsJson();
    });
})
```

### *Prilagođavanje odgovora na iznimku*

Rijetko ćete možda trebati prilagoditi cijeli HTTP odgovor koji prikazuje Laravelov rukovatelj iznimkama (engl. exception handler). Da biste to postigli, možete registrirati anonimnu funkciju (engl. closure) za prilagođavanje odgovora pomoću metode odgovora:

```
use Symfony\Component\HttpFoundation\Response;

->withExceptions(function (Exceptions $exceptions) {
    $exceptions->respond(function (Response $response) {
        if ($response->getStatusCode() === 419) {
            return back()->with([
                'message' => ' Stranica je istekla, pokušajte ponovno.',
            ]);
        }

        return $response;
    });
})
```

### *Iznimke koje je moguće prijaviti i prikazati*

Umjesto definiranja prilagođenog ponašanja izvještavanja i prikazivanja u upravljačkom sklopu za rukovanje iznimkama (engl. exception handler) metodi `register`, možete definirati `report` i `render` metode direktno na iznimkama svoje aplikacije. Kada ove metode postoje, radna okolina će ih automatski pozvati:

```
<?php

namespace App\Exceptions;

use Exception;
use Illuminate\Http\Request;
use Illuminate\Http\Response;

class InvalidOrderException extends Exception
{
    /**
     * Prijavi iznimku.
     */
    public function report(): void
    {
        // ...
    }

    /**
     * Prikaži iznimku u HTTP odgovoru.
     */
    public function render(Request $request): Response
    {
        return response(/* ... */);
    }
}
```

Ako vaša iznimka proširuje iznimku koja se već može prikazati, kao što je ugrađena Laravel ili Symfony iznimka, možete vratiti `false` iz `render` metode iznimke da biste prikazali podrazumijevani HTTP odgovor:

```
/**
 * Prikaži iznimku u HTTP odgovoru.
 */
public function render(Request $request): Response|bool
{
    if (/* Odredi treba li izuzetku prilagođeno izvještavanje */) {

        return response(/* ... */);
    }

    return false;
}
```

Ako vaša iznimka sadrži prilagođenu logiku izvještavanja koja je potrebna samo kada su ispunjeni određeni uslovi, možda ćete morati uputiti Laravel da ponekad prijavi iznimku koristeći zadanu konfiguraciju rukovanja iznimkom. Da biste to postigli, možete se vratiti `false` iz `report` metode iznimke:

```
/**
 * Prijavi iznimku.
 */
public function report(): bool
{
    if (/** Odredi treba li izuzetku prilagođeno izvještavanje */) {

        // ...

        return true;
    }

    return false;
}
```

**NAPOMENA:**

Možete nagovijestiti tip (engl. type-hint) svih potrebnih zavisnosti `report` metode i one će automatski biti injektirane u metodu pomoću Laravelova [kontejnera usluga](#).

### Prigušivanje prijavljenih iznimaka

Ako vaša aplikacija prijavljuje vrlo velik broj iznimaka, možda ćete htjeti smanjiti koliko se iznimaka zapravo bilježi ili šalje vanjskoj usluzi praćenja grešaka vaše aplikacije.

Da biste uzeli slučajnu stopu uzorkovanja iznimaka, možete vratiti instancu `Lottery` iz rukovatelja iznimkama `throttle` metode. Ako vaša `App\Exceptions\Handler` klasa ne sadrži ovu metodu, možete je jednostavno dodati klasi:

```
use Illuminate\Support\Lottery;
use Throwable;

/**
 * Prigušivanje (engl. Throttle) dolaznih iznimaka.
 */
protected function throttle(Throwable $e): mixed
{
    return Lottery::odds(1, 1000);
}
```



Također je moguće uslovno uzorkovanje na temelju tipa iznimke. Ako želite samo uzorkovati instance određene klase iznimke, možete vratiti Lottery instancu samo za tu klasu:

```
use App\Exceptions\ApiMonitoringException;
use Illuminate\Support\Lottery;
use Throwable;

/**
 * Prigušivanje (engl. Throttle) dolaznih iznimaka.
 */
protected function throttle(Throwable $e): mixed
{
    if ($e instanceof ApiMonitoringException) {
        return Lottery::odds(1, 1000);
    }
}
```

Također možete mjeriti ograničenja iznimke zabilježene ili poslane vanjskoj usluzi praćenja grešaka vraćanjem `Limit` instance umjesto `Lottery`. Ovo je korisno ako se želite zaštititi od iznenadnih iznimaka koji preplavljaju vaše log-ove, na primjer, kada usluga treće strane koju koristi vaša aplikacija ne radi:

```
use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;
use Throwable;

/**
 * Prigušivanje (engl. Throttle) dolaznih iznimaka.
 */
protected function throttle(Throwable $e): mixed
{
    if ($e instanceof BroadcastException) {
        return Limit::perMinute(300);
    }
}
```

Prema podrazumijevanim postavkama, ograničenja će koristiti klasu iznimke kao ključ ograničenja brzine. Ovo možete prilagoditi navođenjem vlastitog ključa pomoću `by` metode na `Limit`:

```
use Illuminate\Broadcasting\BroadcastException;
use Illuminate\Cache\RateLimiting\Limit;
use Throwable;

/**
```

```
* Prigušivanje (engl. Throttle) dolaznih iznimaka.  
*/  
protected function throttle(Throwable $e): mixed  
{  
    if ($e instanceof BroadcastException) {  
        return Limit::perMinute(300)->by($e->getMessage());  
    }  
}
```

Naravno, možete vratiti mješavinu `Lottery` i `Limit` instanci za različite iznimke:

```
use App\Exceptions\ApiMonitoringException;  
use Illuminate\Broadcasting\BroadcastException;  
use Illuminate\Cache\RateLimiting\Limit;  
use Illuminate\Support\Lottery;  
use Throwable;  
  
/**  
 * Prigušivanje (engl. Throttle) dolaznih iznimaka.  
 */  
protected function throttle(Throwable $e): mixed  
{  
    return match (true) {  
        $e instanceof BroadcastException => Limit::perMinute(300),  
        $e instanceof ApiMonitoringException => Lottery::odds(1, 1000),  
        default => Limit::none(),  
    };  
}
```

### HTTP iznimke

Neke iznimke opisuju HTTP kodove grešaka s servera. Na primjer, to može biti „stranica nije pronađena - page not found“ greška (404), „greška neovlaštenog pristupa - unauthorized error“ (401) ili čak greška 500 koju je generirao programer. Kako biste generirali takav odgovor s bilo kojeg mjesta u svojoj aplikaciji, možete koristiti `abort` pomoćnika:

```
abort(404);
```

### *Prilagođene stranice s HTTP greškama*

Laravel olakšava prikaz prilagođenih stranica s greškama za različite HTTP statusne kodove. Na primjer, da biste prilagodili stranicu greške za 404 HTTP statusne kodove, kreirajte `resources/views/errors/404.blade.php` predložak pogleda. Ovaj pogled (engl. view) će se prikazati za sve 404 greške koje je generirala vaša aplikacija. Pogledi unutar ovog direktorija trebaju biti imenovani tako da odgovaraju HTTP statusnom kodu kojem odgovaraju. Instanca

`Symfony\Component\HttpFoundation\Exception\HttpException` koju pokreće `abort` funkcija bit će proslijeđena pogledu kao `$exception` varijabla:

```
<h2>{{ $exception->getMessage() }}</h2>
```

Možete objaviti Laravelove zadane predloške stranica s greškama pomoću `vendor:publish` Artisan naredbe. Nakon što su predlošci objavljeni, možete ih prilagoditi svojim željama:

```
php artisan vendor:publish --tag=laravel-errors
```

### *Zamjenske HTTP stranice s greškama*

Također možete definirati „zamjensku“ stranicu s greškom za određenu seriju HTTP statusnih kodova. Ova će se stranica prikazati ako ne postoji odgovarajuća stranica za specifični HTTP statusni kod koji se pojavio. Da biste to postigli, definirajte `4xx.blade.php` predložak i `5xx.blade.php` predložak svoje aplikacije u `resources/views/errors` direktoriju.

Prilikom definiranja zamjenskih stranica s greškama, zamjenske stranice neće utjecati na odgovore o grešci 404, 500 i 503 budući da Laravel ima interne, namjenske stranice za te statusne kodove. Da biste prilagodili stranice prikazane za te statusne kodove, trebali biste definirati prilagođenu stranicu s greškom za svaku od njih pojedinačno.

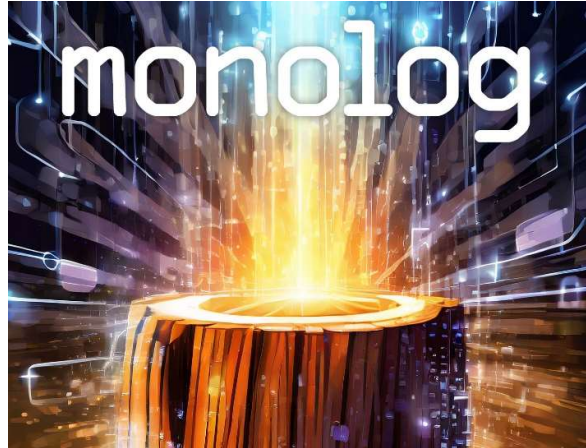
## Evidentiranje

### Uvod

Kako bi vam pomogao da saznate više o tome što se događa unutar vaše aplikacije, Laravel pruža robusne usluge evidentiranja (engl. logging) koje vam omogućuju da bilježite (engl. log) poruke u datoteke, zapisnik grešaka sistema, pa čak i Slack da obavijestite cijeli svoj tim.

Laravel evidentiranje se temelji na „kanalima“. Svaki kanal predstavlja specifičan način zapisivanja log podataka. Na primjer, `single` kanal zapisuje log datoteke u jednu log datoteku, dok `slack` kanal šalje log poruke Slacku. Log poruke mogu se zapisati na više kanala na temelju njihove težine.

Ispod haube, Laravel koristi [Monolog](#) biblioteku, koja pruža podršku za niz moćnih rukovatelja log-ovima. Laravel olakšava konfiguriranje ovih rukovatelja, omogućujući vam da ih kombinirate kako biste prilagodili rukovanje log-ovima svoje aplikacije.



### Konfiguracija

Sve konfiguracijske opcije za evidenciju ponašanja vaše aplikacije nalaze se u `config/logging.php` konfiguracijskoj datoteci. Ova datoteka vam omogućava da konfigurirate log kanale vaše aplikacije, zato svakako pregledajte svaki od dostupnih kanala i njihove opcije. U nastavku ćemo pregledati nekoliko uobičajenih opcija.

Prema zadanim postavkama, Laravel će koristiti `stack` kanal kada evidentira poruke. `stack` kanal se koristi za agregiranje više kanala dnevnika u jedan kanal. Za više informacija o izgradnji stekova, pogledajte [dokumentaciju u nastavku](#).

### Dostupni driveri kanala

Svaki log kanal je pogonjen sa upravljačkim programom (engl. driver). Upravljački program određuje kako i gdje se log poruka u stvari bilježi. Sljedeći upravljački programi log kanala dostupni su u svakoj Laravel aplikaciji. Unos za većinu ovih upravljačkih programa već je prisutan u `config/logging.php` konfiguracijskoj datoteci vaše aplikacije, zato svakako pregledajte ovu datoteku kako biste se upoznali s njezinim sadržajem:

Ime	Opis
<code>custom</code>	Upravljački program koji poziva određenu tvornicu za stvaranje kanala
<code>daily</code>	<code>RotatingFileHandler</code> temeljen na Monologu upravljačkom programu koji se svakodnevno mijenja
<code>errorlog</code>	<code>ErrorLogHandler</code> upravljački program temeljen na Monologu
<code>monolog</code>	Monolog tvornički upravljački program koji može koristiti bilo koji podržani Monolog rukovatelj
<code>papertrail</code>	<code>SyslogUdpHandler</code> temeljen na Monolog upravljačkom programu

<code>single</code>	Jedna datoteka ili putanja temeljena na kanalu evidencije ( <code>StreamHandler</code> )
<code>slack</code>	<code>SlackWebhookHandler</code> temeljen na Monolog upravljačkom programu
<code>stack</code>	Omotač za olakšavanje stvaranja "višekanalnih" kanala
<code>syslog</code>	SyslogHandler temeljen na Monolog upravljačkom programu

**NAPOMENA:**

Provjerite dokumentaciju o naprednoj prilagođavanju kanala da biste saznali više o `monolog` i `custom` upravljačkim programima.

*Konfiguriranje naziva kanala*

Prema zadanim postavkama, Monolog se instancira s "nazivom kanala" koji odgovara trenutnoj okolini, kao što je `production` ili `local`. Da biste promijenili ovu vrijednost, dodajte `name` opciju u konfiguraciju svog kanala:

```
'stack' => [
    'driver' => 'stack',
    'name' => 'channel-name',
    'channels' => ['single', 'slack'],
],
```

*Preduslovi kanala**Konfiguriranje pojedinačnih i dnevnih kanala*

`single` i `daily` kanali imaju tri opcionalne mogućnosti konfiguracije: `bubble`, `permission` i `locking`.

Ime	Opis	Podrazumijevano
<code>bubble</code>	Označava trebaju li se poruke pojaviti u mjehurićima na drugim kanalima nakon obrade	<code>true</code>
<code>locking</code>	Pokušaj zaključati log datoteku prije pisanja u nju	<code>false</code>
<code>permission</code>	Dopuštenja log datoteke	<code>0644</code>

Dodatno, politika zadržavanja za `daily` kanal može se konfigurirati pomoću `days` opcije:

Ime	Opis	Podrazumijevano
<code>days</code>	Broj dana tokom kojih se dnevne datoteke dnevnika trebaju čuvati	<code>7</code>

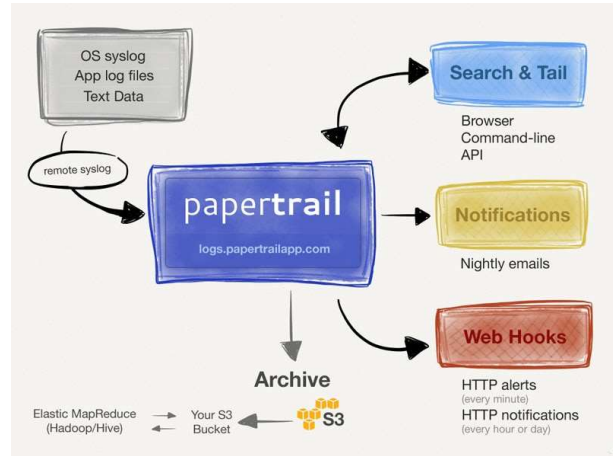
### Konfiguriranje Papertrail kanala

`papertrail` kanal zahtijeva opcije konfiguracije hosti port. Ove vrijednosti možete dobiti od [Papertraila](#).

### Konfiguriranje Slack kanala

`slack` kanal zahtijeva `url` opciju konfiguracije. Ovaj URL trebao bi odgovarati URL-u za dolazni webhook koji ste konfigurirali za svoj Slack<sup>52</sup> tim.

Prema zadanim postavkama, Slack će primati zapise samo na `critical` nivou i višem; međutim, to možete podesiti u svojoj `config/logging.php` konfiguracijskoj datoteci izmjenom `level` konfiguracijske opcije unutar konfiguracijske matrice vašeg Slack log kanala.



### Upozorenja o zastarjelosti evidentiranja

PHP, Laravel i druge biblioteke često obavještavaju svoje korisnike da su neke od njihovih svojstava zastarjela i da će biti uklonjene u budućoj verziji. Ako želite zabilježiti ova upozorenja o zastarjevanju, možete navesti željeni `deprecations` log kanal vaše aplikacije u `config/logging.php` konfiguracijskoj datoteci:

```
'deprecations' => [
    'channel' => env('LOG_DEPRECATIONS_CHANNEL', 'null'),
    'trace' => env('LOG_DEPRECATIONS_TRACE', false),
],

'channels' => [
    // ...
]
```

Ili, možete definirati log kanal pod nazivom `deprecations`. Ako postoji log kanal s ovim imenom, on će se uvijek koristiti za bilježenje zastara:

```
'channels' => [
    'deprecations' => [
        'driver' => 'single',
        'path' => storage_path('logs/php-deprecation-warnings.log'),
    ],
]
```

<sup>52</sup> Slack je timska komunikacijska platforma temeljena na oblaku koju je razvio Slack Technologies, koji je od 2020. u vlasništvu Salesforcea, koji je poznat po Sales Cloud, najpopularnijem CRM software-u

```
],
```

### Izgradnja evidencije steka

Kao što je ranije spomenuto, stack upravljački program vam omogućava kombiniranje više kanala u jedan log kanal radi praktičnosti. Da bismo ilustrirali kako koristiti log stekove, pogledajmo primjer konfiguracije koju možete vidjeti u proizvodnoj aplikaciji:

```
'channels' => [
    'stack' => [
        'driver' => 'stack',
        'channels' => ['syslog', 'slack'],
    ],

    'syslog' => [
        'driver' => 'syslog',
        'level' => 'debug',
    ],

    'slack' => [
        'driver' => 'slack',
        'url' => env('LOG_SLACK_WEBHOOK_URL'),
        'username' => 'Laravel Log',
        'emoji' => ':boom:',
        'level' => 'critical',
    ],
],
```

Raščlanimo ovu konfiguraciju. Prvo, primijetite da naš `stack` kanal spaja dva druga kanala putem svoje `channels` opcije: `syslog` i `slack`. Dakle, kada bilježite poruku, oba ova kanala će imati priliku zabilježiti poruku. Međutim, kao što ćemo vidjeti u nastavku, hoće li ovi kanali stvarno zabilježiti poruku može biti određeno ozbiljnošću / „nivom“ poruke.

### Nivoi log-ova

Obratite pažnju na `level` opciju konfiguracije koja je prisutna na `syslog` i `slack` konfiguracijama kanala u primjeru iznad. Ova opcija određuje minimalni „nivo“ koji poruka mora imati da bi je kanal zabilježio. Monolog, koji pokreće Laravel-ove usluge zapisivanja, nudi sve log nivoe definirane u [RFC 5424 specifikaciji](#). U silaznom redoslijedu težine, ovi log nivoi su: hitno (engl. emergency), upozorenje (engl. alert), kritično (engl. critical), greška (engl. error), upozorenje (engl. warning), obavijest (engl. notice), informacija (engl. info) i otklanjanje grešaka (engl. debug).

Dakle, zamislite da bilježimo poruku pomoću `debug` metode:

```
Log::debug('Informativna poruka.');
```

S obzirom na našu konfiguraciju, `syslog` kanal će napisati poruku u sistem log; međutim, budući da poruka o grešci nije `critical` ili iznad, neće biti poslana Slacku. Međutim, ako zabilježimo `emergency` poruku, ona će biti poslana i u sistem log i u Slack jer je `emergency` nivo iznad našeg minimalnog praga nivoa za oba kanala:

```
Log::emergency('Sistem je pao!');
```

#### Pisanje log poruka

Možete pisati informacije u zapisnike pomoću `Log facade`. Kao što je ranije spomenuto, logger pruža osam nivoa zapisivanja definiranih u [RFC 5424 specifikaciji](#): hitno (engl. emergency), upozorenje (engl. alert), kritično (engl. critical), greška (engl. error), upozorenje (engl. warning), obavijest (engl. notice), informacija (engl. info) i otklanjanje grešaka (engl. debug):

```
use Illuminate\Support\Facades\Log;

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

Možete pozvati bilo koju od ovih metoda za zapisivanje poruke za odgovarajući nivo. Prema zadanim postavkama, poruka će biti zapisana na zadani log kanal kako je konfigurirano u vašoj `logging` konfiguracijskoj datoteci:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Support\Facades\Log;
use Illuminate\View\View;

class UserController extends Controller
{
    /**
     * Prikaži profil za određenog korisnika.
     */
    public function show(string $id): View
    {
```



```
Log::info('Showing the user profile for user: {id}', ['id' => $id]);

return view('user.profile', [
    'user' => User::findOrFail($id)
]);
}
}
```

### *Kontekstualne informacije*

Matrica kontekstualnih podataka može se proslijediti log metodama. Ovi kontekstualni podaci bit će formatirani i prikazani s log porukom:

```
use Illuminate\Support\Facades\Log;

Log::info('Korisnik {id} nije se uspio logirati.', ['id' => $user->id]);
```

Povremeno ćete možda htjeti navesti neke kontekstualne informacije koje bi trebale biti uključene u sve naredne log unose na određenom kanalu. Na primjer, možda želite zabilježiti ID zahtjeva koji je povezan sa svakim dolaznim zahtjevom vašoj aplikaciji. Da biste to postigli, možete pozvati `Log` fasade `withContext` metodu:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AssignRequestId
{
    /**
     * Rukovanje dolaznim zahtjevom.
     *
     * @param  \Closure(\Illuminate\Http\Request):
    (\Symfony\Component\HttpFoundation\Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::withContext([
```

```
        'request-id' => $requestId
    ]);

    $response = $next($request);

    $response->headers->set('Request-Id', $requestId);

    return $response;
}
}
```

Ako želite dijeliti kontekstualne informacije preko svih kanala za evidentiranje, možete pozvati `Log::shareContext()` metodu. Ova će metoda pružiti kontekstualne informacije svim kreiranim kanalima i svim kanalima koji se naknadno kreiraju:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Log;
use Illuminate\Support\Str;
use Symfony\Component\HttpFoundation\Response;

class AssignRequestId
{
    /**
     * Rukovanje dolaznim zahtjevom.
     *
     * @param  \Closure(\Illuminate\Http\Request):
    (\Symfony\Component\HttpFoundation\Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        $requestId = (string) Str::uuid();

        Log::shareContext([
            'request-id' => $requestId
        ]);

        // ...

    }
}
```

**NAPOMENA:**

Ako trebate dijeliti log kontekst dok obrađujete poslove u redu čekanja, možete upotrijebiti middleware za posao.

*Zapisivanje na određene kanale*

Ponekad ćete možda poželjeti zapisati poruku na kanal koji nije podrazumijevani kanal vaše aplikacije. Možete koristiti `channel` metodu na `Log` fasadi za dohvaćanje i prijavu na bilo koji kanal definiran u vašoj konfiguracijskoj datoteci:

```
use Illuminate\Support\Facades\Log;

Log::channel('slack')->info('Nešto se desilo!');
```

*Kanali na zahtjev*

Također je moguće kreirati kanal na zahtjev (engl. on-demand logging stack) pružanjem konfiguracije tokom izvođenja bez te konfiguracije u `logging` konfiguracijskoj datoteci vaše aplikacije. Da biste to postigli, možete proslijediti konfiguracijsku matricu `Log` fasade `build` metodi:

```
use Illuminate\Support\Facades\Log;

Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
])->info(' Nešto se desilo!');
```

Možda biste također željeli uključiti on-demand kanal u on-demand stek zapisnik. To se može postići uključivanjem vaše on-demand instance kanala u matricu proslijeđenu `stack` metodi:

```
use Illuminate\Support\Facades\Log;

$channel = Log::build([
    'driver' => 'single',
    'path' => storage_path('logs/custom.log'),
]);

Log::stack(['slack', $channel])->info('Nešto se desilo!');
```

## Monolog prilagođavanje kanala

*Prilagođavanje Monologa za kanale*

Ponekad ćete možda trebati potpunu kontrolu nad načinom na koji je Monolog konfiguriran za postojeći kanal. Na primjer, možda biste željeli konfigurirati prilagođenu Monolog `FormatterInterface` implementaciju za Laravel-ov ugrađeni `single` kanal.

Za početak definirajte `tap` matricu u konfiguraciji kanala. `tap` matrica bi trebala sadržavati popis klasa koje bi trebale imati priliku prilagoditi (ili „kucnuti“) Monolog instancu nakon što je kreirana. Ne postoji konvencionalno mjesto gdje bi te klase trebale biti smještene, tako da slobodno možete stvoriti direktorij unutar svoje aplikacije koji će sadržavati ove klase:

```
'single' => [  
    'driver' => 'single',  
    'tap' => [App\Logging\CustomizeFormatter::class],  
    'path' => storage_path('logs/laravel.log'),  
    'level' => 'debug',  
],
```

Nakon što ste konfigurirali `tap` opciju na svom kanalu, spremni ste definirati klasu koja će prilagoditi vašu Monolog instancu. Ova klasa treba samo jednu metodu: `__invoke`, koja prima `Illuminate\Log\Logger` instancu. `Illuminate\Log\Logger` instanca opunomoćava sve pozive metode do osnovne Monolog instance:

```
<?php  
  
namespace App\Logging;  
  
use Illuminate\Log\Logger;  
use Monolog\Formatter\LineFormatter;  
  
class CustomizeFormatter  
{  
    /**  
     * Prilagodite danu instancu logger-a.  
     */  
    public function __invoke(Logger $logger): void  
    {  
        foreach ($logger->getHandlers() as $handler) {  
            $handler->setFormatter(new LineFormatter(  
                '[%datetime%] %channel%.%level_name%: %message% %context% %extra%'  
            ));  
        }  
    }  
}
```

**NAPOMENA:**

Sve vaše „tap“ klase rješava [kontejner usluge](#), tako da će sve zavisnosti konstruktora koje zahtijevaju automatski biti injektirane.

### *Kreiranje Monolog Handler kanala*

Monolog ima niz dostupnih rukovatelja, a Laravel ne uključuje ugrađeni kanal za svaki od njih. U nekim slučajevima možda ćete poželjeti stvoriti prilagođeni kanal koji je samo instanca određenog Monolog rukovatelja koji nema odgovarajući Laravel log driver. Ovi se kanali mogu jednostavno stvoriti pomoću `monolog` drivera.

Kada koristite `monolog` driver, `handler` konfiguracijska opcija se koristi za određivanje koji će rukovatelj biti instanciran. Po izboru, bilo koji parametri konstruktora koji su potrebni rukovatelju mogu se specificirati pomoću `with` konfiguracijske opcije:

```
'logentries' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\SyslogUdpHandler::class,
    'with' => [
        'host' => 'my.logentries.internal.datahubhost.company.com',
        'port' => '10000',
    ],
],
```

### *Monolog formateri*

Kada koristite `monolog` driver, Monolog LineFormatter će se koristiti kao zadani formater. Međutim, možete prilagoditi vrstu formatera koji se proslijeđuje rukovatelju pomoću opcija konfiguracije `formatter` i `formatter_with`:

```
'browser' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\BrowserConsoleHandler::class,
    'formatter' => Monolog\Formatter\HtmlFormatter::class,
    'formatter_with' => [
        'dateFormat' => 'Y-m-d',
    ],
],
```

Ako koristite Monolog rukovatelj koji je sposoban osigurati vlastiti formater, možete postaviti vrijednost konfiguracijske `formatter` opcije na `default`:

```
'newrelic' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\NewRelicHandler::class,
    'formatter' => 'default',
],
```

### *Monolog procesori*

Monolog također može obraditi poruke prije nego ih zabilježi. Možete izraditi vlastite procesore ili koristiti [postojeće procesore koje nudi Monolog](#).

Ako želite prilagoditi procesore za `monolog` driver, dodajte `processors` vrijednost konfiguracije konfiguraciji vašeg kanala:

```
'memory' => [
    'driver' => 'monolog',
    'handler' => Monolog\Handler\StreamHandler::class,
    'with' => [
        'stream' => 'php://stderr',
    ],
    'processors' => [
        // Jednostavna sintaksa...
        Monolog\Processor\MemoryUsageProcessor::class,

        // Sa opcijama...
        [
            'processor' => Monolog\Processor\PsrLogMessageProcessor::class,
            'with' => ['removeUsedContextFields' => true],
        ],
    ],
],
```

### *Kreiranje prilagođenih kanala pomoću Factories*

Ako želite definirati potpuno prilagođeni kanal u kojem imate punu kontrolu nad instanciranjem Monolog konfiguracijom, možete navesti `custom` tip drivera u vašoj `config/logging.php` konfiguracijskoj datoteci. Vaša konfiguracija treba uključivati `via` opciju koja sadrži naziv tvorničke klase koja će se pozvati za stvaranje Monolog instance:

```
'channels' => [
    'example-custom-channel' => [
        'driver' => 'custom',
        'via' => App\Logging\CreateCustomLogger::class,
    ],
],
```

Nakon što konfigurirate `custom` driver, spremni ste definirati klasu koja će kreirati vašu Monolog instancu. Ova klasa treba samo jednu `__invoke` metodu koja bi trebala vratiti Monolog zapisnik instancu. Metoda će primiti polje konfiguracije kanala kao jedini argument:

```
<?php

namespace App\Logging;
```

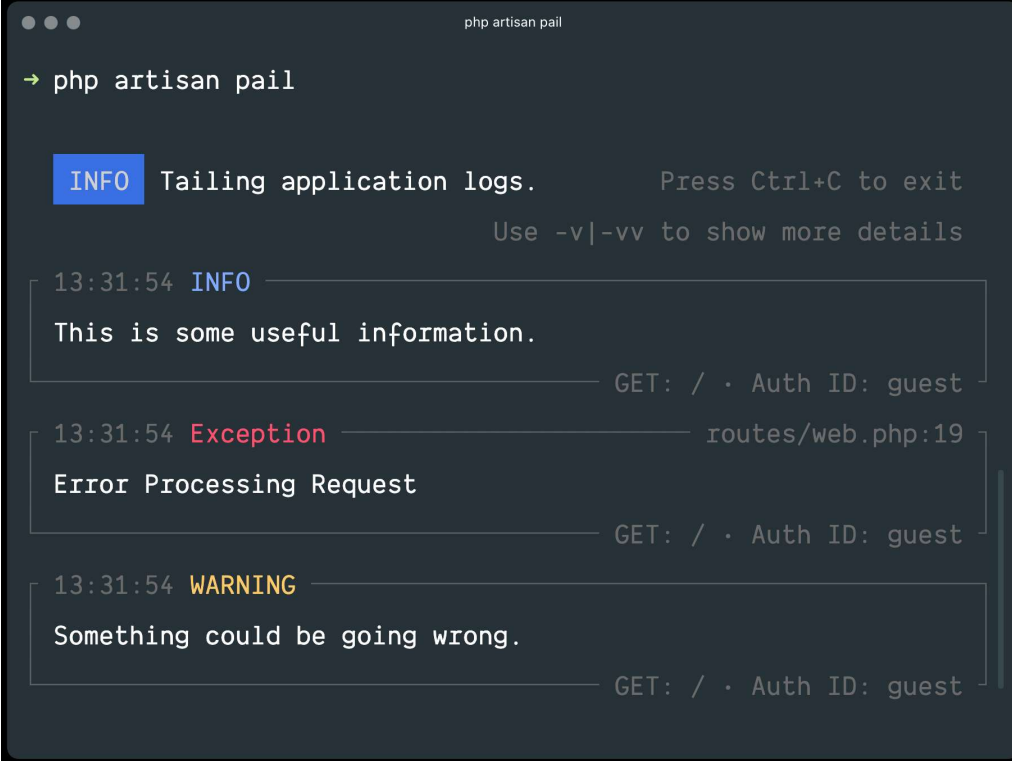
```
use Monolog\Logger;

class CreateCustomLogger
{
    /**
     * Kreiraj prilagođenu Monolog instancu.
     */
    public function __invoke(array $config): Logger
    {
        return new Logger(/* ... */);
    }
}
```

### Praćenje evidentiranih poruka pomoću Pail-a

Često ćete možda morati pratiti zapisnike svoje aplikacije u realnom vremenu. Na primjer, kada otklanjate greške s problemima (engl. debugging) ili kada nadzirete log-ove svoje aplikacije radi traženja određenih vrsta grešaka.

Laravel Pail je paket koji vam omogućava da jednostavno zaronite u vaše Laravel log datoteke aplikacije direktno iz komandnog reda. Za razliku od standardne `tail` naredbe, Pail je dizajniran za rad s bilo kojim log driverom, uključujući Sentry ili Flare. Osim toga, Pail nudi skup korisnih filtera koji vam pomažu da brzo pronađete ono što tražite.



```
php artisan pail

→ php artisan pail

INFO Tailing application logs. Press Ctrl+C to exit
Use -v|-vv to show more details

13:31:54 INFO —————
This is some useful information.
————— GET: / • Auth ID: guest

13:31:54 Exception ————— routes/web.php:19
Error Processing Request
————— GET: / • Auth ID: guest

13:31:54 WARNING —————
Something could be going wrong.
————— GET: / • Auth ID: guest
```

## Instalacija

### UPOZORENJE:

Laravel Pail zahtijeva PHP [8.2 i viši](#) i [PCNTL ekstenziju](#).



Za početak instalirajte Pail u svoj projekt pomoću Composer-a upravitelja paketa:

```
composer require laravel/pail
```

## Korištenje

Da biste započeli s praćenjem log-ova, pokrenite `pail` naredbu:

```
php artisan pail
```

Da biste povećali opširnost izlaza i izbjegli skraćivanje (engl. truncation) (...), koristite `-v` opciju:

```
php artisan pail -v
```

Za maksimalnu opširnost i prikaz tragova iznimaka steka (engl. exception stack traces) upotrijebite `-vv` opciju:

```
php artisan pail -vv
```

Za prekid praćenja log-ova, pritisnite `Ctrl+C` bilo kada.

## Filtriranje zapisa

`--filter`

Možete upotrijebiti `--filter` opciju za filtriranje log-ova prema tipu, datoteci, poruci i sadržaju praćenja steka:

```
php artisan pail --filter="QueryException"  
--message
```

`--message`

Za filtriranje log-ova samo prema njihovoj poruci, možete koristiti `--message` opciju:

```
php artisan pail --message="User created"
```

`--level`

Opcija `--level` se može koristiti za filtriranje log-ova prema njihovom [nivou log-a](#):



```
php artisan pail --level=error
```

`--user`

Za prikaz samo log-ova koji su napisani dok je određen korisnik bio autentificiran, možete unijeti korisnički ID za `--user` opciju:

```
php artisan pail --user=1
```

## Dublje kopanje

### Artisan konzola

#### Uvod

Artisan je interface komandnog reda uključen u Laravel. Artisan postoji u korijenu vaše aplikacije kao `artisan` skripta i pruža brojne korisne naredbe koje vam mogu pomoći dok gradite svoju aplikaciju. Za pregled popisa svih dostupnih Artisan naredbi, možete koristiti `list` naredbu:

**NAPOMENA:**

```
php artisan list
```



Svaka naredba također uključuje „help“ ekran koji prikazuje i opisuje dostupne argumente i opcije naredbe. Za prikazivanje help ekrana ispred naziva naredbe stavite `help`:

```
php artisan help migrate
```

#### Laravel Sail

Ako koristite Laravel Sail kao svoje lokalnu razvojnu okolinu, ne zaboravite koristiti sail komandni red za pozivanje Artisan naredbi. Sail će izvršiti vaše Artisan naredbe unutar Docker kontejnera vaše aplikacije:

```
./vendor/bin/sail artisan list
```

#### Tinker (REPL)

Laravel Tinker moćan je REPL (Read–eval–print loop) za Laravel radnu okolinu, kojeg pokreće [PsySH](#) paket.

#### Instalacija

Sve Laravel aplikacije prema zadanim postavkama uključuju Tinker. Međutim, možete instalirati Tinker koristeći Composer ako ste ga prethodno uklonili iz svoje aplikacije:

```
composer require laravel/tinker
```

**NAPOMENA:**

Tražite brzo ponovno učitavanje, editiranje višelinijjskih kodova i automatsko dovršavanje kod interakcije s vašom Laravel aplikacijom? Provjerite [Tinkerwell](#)!

#### Korištenje

Tinker vam omogućava interakciju s vašom cijelom Laravel aplikacijom u komandnom redu, uključujući vaše Eloquent modele, poslove, događaje i još mnogo toga. Za ulazak u Tinker-ovu okolinu, pokrenite `tinker` Artisan naredbu:

```
php artisan tinker
```

Tinkerovu konfiguracijsku datoteku možete objaviti pomoću `vendor:publish` naredbe:

```
php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"
```

**UPOZORENJE:**

`dispatch` funkcija pomoćnika i `dispatch` metoda na `Dispatchable` klasi ovisi o skupljanju smeća za stavljanje posla u red čekanja (engl. queue). Zato, kada koristite tinker, trebali biste koristiti `Bus::dispatch` ili `Queue::push` za poslove otpremanja.

*Lista dopuštenih naredbi*

Tinker koristi „dopuštenu“ listu kako bi odredio koje se Artisanove naredbe smiju pokretati unutar njegove ljuske. Prema podrazumijevanim postavkama možete pokrenuti naredbe `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `optimize` i `up` naredbe. Ako želite dopustiti više naredbi, možete ih dodati u `commands` matricu u vašoj `tinker.php` konfiguracijskoj datoteci:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

*Klase koje ne bi trebale imati aliase*

U pravilu, Tinker automatski daje aliase klasama dok komunicirate s njima u Tinkeru. Međutim, možda ćete poželjeti nikada ne koristiti aliase za neke klase. To možete postići izlistavanjem klasa u `dont_alias` klasi vaše `tinker.php` konfiguracijske datoteke:

```
'dont_alias' => [
    App\Models\User::class,
],
```

*Pisanje naredbi*

Uz naredbe koje nudi Artisan, možete izraditi vlastite prilagođene naredbe. Naredbe su obično pohranjene u `app/Console/Commands` direktoriju; međutim, slobodni ste odabrati vlastitu lokaciju za pohranjivanje sve dok vaše naredbe može učitati Composer.

*Generiranje naredbi*

Za generiranje nove naredbe, možete koristiti `make:command` Artisan naredbu. Ova naredba će kreiraati novu klasu naredbi u `app/Console/Commands` direktoriju. Ne brinite ako ovaj direktorij ne postoji u vašoj aplikaciji - bit će kreiran kada prvi put pokrenete `make:command` Artisan naredbu:

```
php artisan make:command SendEmails
```

### Struktura naredbi

Nakon generiranja vaše naredbe, trebali biste definirati odgovarajuće vrijednosti za `signature` i `description` svojstva klase. Ova svojstva će se koristiti kada prikazuje vaše naredbe na `list` ekranu. Svojstvo `signature` vam također omogućava da definirate [ulazna očekivanja vaše naredbe](#). Metoda `handle` će biti pozvana kada se vaša naredba izvrši. Svoju naredbenu logiku možete smjestiti u ovu metodu.

Pogledajmo primjer naredbe. Imajte na umu da možemo zatražiti sve ovisnosti koje su nam potrebne kroz naredbe `handle` metode. Laravel [kontejner usluga](#) automatski će ubaciti sve ovisnosti koje su nagovještavanjem tipa (engl. type-hint) u potpisu ove metode:

```
<?php

namespace App\Console\Commands;

use App\Models\User;
use App\Support\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * Naziv i potpis naredbe konzole..
     *
     * @var string
     */
    protected $signature = 'mail:send {user}';

    /**
     * Opis naredbe konzole.
     *
     * @var string
     */
    protected $description = 'Send a marketing email to a user';

    /**
     * Izvrši naredbu konzole.
     */
    public function handle(DripEmailer $drip): void
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```

**NAPOMENA:**

Za veće ponovo korištenje koda, dobra je praksa da vaše konzolne naredbe budu lagane i da se prepuste aplikacijskim servisima da izvrše svoje zadatke. U gornjem primjeru, imajte na umu da ubacujemo klasu usluge za obavljanje „teškog posla“ slanja e-pošte.

*Izlazni kodovi*

Ako se ništa ne vrati iz metode rukovanja i naredba se uspješno izvrši, naredba će izaći s izlaznim kodom 0, što ukazuje na uspjeh. Međutim, metoda rukovanja može opcionalno vratiti cjelobrojni broj za ručno specificiranje izlaznog koda naredbe:

```
$this->error('Nešto nije u redu.');
```

```
return 1;
```

Ako želite „podbaciti“ naredbom iz bilo koje metode unutar naredbe, možete upotrijebiti `fail` metodu. `fail` metoda će odmah prekinuti izvršenje naredbe i vratiti izlazni kod 1:

```
$this->fail('Something went wrong.');
```

*Naredbe anonimnih funkcija (engl. closures)*

Naredbe temeljene na anonimnim funkcijama (engl. closure) pružaju alternativu definiranju konzolnih naredbi kao klasa. Na isti način na koji su anonimne funkcije (engl. closures) ruta alternativa kontrolerima, razmišljajte o naredbama anonimnih funkcija (engl. closure) kao alternativni naredbama klasa.

Iako datoteka `routes/console.php` ne definira HTTP rute, ona definira ulazne točke temeljene na konzoli (rute) u vašu aplikaciju. Unutar ove datoteke možete definirati sve vaše konzolne naredbe temeljene na anonimnim funkcijama (engl. closure) pomoću metode `Artisan::command`. Metoda naredbe prihvaća dva argumenta: potpis naredbe i anonimne funkcije (engl. closure) koje prima argumente i opcije naredbe:

Unutar `commands` metode vaše `app/Console/Kernel.php` datoteke, Laravel učitava `routes/console.php` datoteku:

```
Artisan::command('mail:send {user}', function (string $user) {
    $this->info("Sending email to: {$user}!");
});
```

Anonimna funkcija (engl. closure) se veže za temeljnu instancu naredbe, tako da imate potpuni pristup svim pomoćnim metodama kojima biste obično mogli pristupiti u punoj naredbi klase.

*Zavisnosti nagovještavanjem tipa (engl. type-hint)*

Osim primanja argumenata i opcija vaše naredbe, zatvaranja naredbi također mogu nagovještavanjem tipa (engl. type-hint) dodatne ovisnosti koje biste željeli riješiti izvan [kontejnera usluge](#):

```
use App\Models\User;
use App\Support\DripEmailer;

Artisan::command('mail:send {user}', function (DripEmailer $drip, string $user) {
    $drip->send(User::find($user));
});
```

### Opisi naredbi anonimnih funkcija (engl. closures)

Kada definirate naredbu anonimne funkcije (engl. closure), možete koristiti `purpose` metodu za dodavanje opisa naredbi. Ovaj opis će se prikazati kada pokrenete `php artisan list` ili `php artisan help` naredbe:

```
Artisan::command('mail:send {user}', function (string $user) {
    // ...
})->purpose('Pošalji marketinški email korisniku');
```

### Naredbe koje se mogu izolirati

#### UPOZORENJE:



Da biste koristili ovo svojstvo, vaša aplikacija mora koristiti `memcached`, `redis`, `dynamodb`, `database`, `file` ili `array` keš driver kao zadani keš driver. Osim toga, svi serveri moraju komunicirati s istim centralnim keš serverom.

Ponekad ćete možda poželjeti osigurati da se samo jedna instanca naredbe može izvoditi u isto vrijeme. Da biste to postigli, možete implementirati `Illuminate\Contracts\Console\Isolatable` interface na svoju klasu naredbi:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Contracts\Console\Isolatable;

class SendEmails extends Command implements Isolatable
{
    // ...
}
```

Kada je naredba označena kao `Isolatable`, Laravel će automatski dodati `--isolated` opciju naredbi. Kada se naredba pozove s tom opcijom, Laravel će osigurati da nijedna druga instanca te naredbe nije već pokrenuta. Laravel to postiže pokušavajući nabaviti atomsko zaključavanje koristeći zadani keš

driver vaše aplikacije. Ako se pokreću druge instance naredbe, naredba se neće izvršiti; međutim, naredba će još uvijek izaći s uspješnim status kodom izlaza:

```
php artisan mail:send 1 --isolated
```

Ako želite navesti izlazni statusni kod koji bi naredba trebala vratiti ako se ne može izvršiti, možete dati željeni statusni kod pomoću `isolated` opcije:

```
php artisan mail:send 1 --isolated=12
```

### *Zaključavanje ID*

Prema zadanim postavkama, Laravel će koristiti naziv naredbe za generiranje string ključa koji se koristi za dobivanje atomskog zaključavanja u kešu vaše aplikacije. Međutim, ovaj ključ možete prilagoditi definiranjem `isolatableId` metode u svojoj Artisan klasi naredbi, što vam omogućava integraciju argumenata ili opcija naredbe u ključ:

```
/**
 * Nabavi izolirani ID za naredbu.
 */
public function isolatableId(): string
{
    return $this->argument('user');
}
```

### *Vrijeme kada istječe zaključavanje*

Prema zadanim postavkama, izolacijska zaključavanja istječu nakon završetka naredbe. Ili, ako se naredba prekine i ne može završiti, zaključavanje će isteći nakon jednog sata. Međutim, možete prilagoditi vrijeme isteka zaključavanja definiranjem `isolationLockExpiresAt` metode na svojoj naredbi:

```
use DateTimeInterface;
use DateInterval;

/**
 * Odredite kada istječe zaključavanje izolacije za naredbu.
 */
public function isolationLockExpiresAt(): DateTimeInterface|DateInterval
{
    return now()->addMinutes(5);
}
```

## Definiranje ulaznih očekivanja

Kada pišete konzolne naredbe, uobičajeno je prikupljati unose od korisnika pomoću argumenata ili opcija. Laravel čini vrlo praktičnim definiranje unosa koji očekujete od korisnika pomoću `signature` svojstva vaših naredbi. Svojstvo `signature` vam omogućava da definirate naziv, argumente i opcije za naredbu u jednoj, ekspresivnoj sintaksi sličnoj sintaksi rute.

### Argumenti

Svi argumenti i opcije koje je dao korisnik umotani su u vitičaste zagrade. U sljedećem primjeru, naredba definira jedan potreban argument `user`:

```
/**
 * Ime i potpis konzolne naredbe.
 *
 * @var string
 */
protected $signature = 'mail:send {user}';
```

Također možete učiniti argumente opcionalnim ili definirati zadane vrijednosti za argumente:

```
// Opcionalni argument...
'mail:send {user?}'

// Opcionalni argument s zadonom vrijednošću...
'mail:send {user=foo}'
```

### Opcije

Opcije su, poput argumenata, još jedan oblik korisničkog unosa. Opcije su ispred dvije crtice ( --) kada se daju pomoću reda s naredbom. Postoje dvije vrste opcija: one koje dobivaju vrijednost i one koje ne dobivaju. Opcije koje ne primaju vrijednost služe kao logički prekidač (engl. Boolean switch). Pogledajmo primjer ovog tipa opcije:

```
/**
 * Ime i potpis konzolne naredbe.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue}';
```

U ovom primjeru, `--queue` prekidač se može navesti kada pozivate Artisan naredbe. Ako je `--queue` prekidač prošao, vrijednost opcije bit će `true`. U suprotnom, vrijednost će biti `false`:

```
php artisan mail:send 1 --queue
```



### Opcije s vrijednostima

Zatim, pogledajmo opciju koja očekuje vrijednost. Ako korisnik mora navesti vrijednost za opciju, kao sufiks nazivu opcije trebate dodati `=` znak:

```
/**
 * Ime i potpis naredbe konzole
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue=}';
```

U ovom primjeru, korisnik može proslijediti vrijednost za takvu opciju. Ako opcija nije navedena kada pozivate naredbu, njezina će vrijednost biti `null`:

```
php artisan mail:send 1 --queue=default
```

Možete dodijeliti zadane vrijednosti opcijama navođenjem zadane vrijednosti iza naziva opcije. Ako korisnik ne proslijedi niti jednu vrijednost opcije, koristit će se zadana vrijednost:

```
'mail:send {user} {--queue=default}'
```

### Prečice opcija

Da biste dodijelili prečicu kada definirate opciju, možete je navesti prije naziva opcije i koristiti znak `|` kao graničnik za odvajanje prečice od punog naziva opcije:

```
'mail:send {user} {--Q|queue}'
```

Kada pozivate naredbu na vašem terminalu, prečice opcija trebali bi imati prefiks s jednom crticom i niti jedan `=` znak ne bi trebao biti uključen kada se navodi vrijednost za opciju:

```
php artisan mail:send 1 -Qdefault
```

### Ulazne matrice

Ako želite definirati argumente ili opcije za očekivanje više ulaznih vrijednosti, možete koristiti `*` znak. Prvo, pogledajmo primjer koji specificira takav argument:

```
'mail:send {user*}'
```

Prilikom pozivanja ove metode, `user` argumenti se mogu proslijediti redoslijedom u komandnom redu. Na primjer, sljedeća naredba postaviti će vrijednost `user` na polje s `1` i `2` kao njegove vrijednosti:

```
php artisan mail:send 1 2
```

Ovaj `*` se znak može kombinirati s neobaveznom definicijom argumenta kako bi se omogućilo nula ili više instanci argumenta:

```
'mail:send {user?*}'
```

### Opcijske matrice

Prilikom definiranja opcije koja očekuje višestruke ulazne vrijednosti, svaka vrijednost opcije proslijeđena naredbi trebala bi imati prefiks s nazivom opcije:

```
'mail:send {--id=*}'
```

Takva se naredba može pozvati proslijeđivanjem više `--id` argumenata:

```
php artisan mail:send --id=1 --id=2
```

### Opisi unosa

Možete dodijeliti opise ulaznim argumentima i opcijama odvajanjem naziva argumenta od opisa pomoću dvotočke. Ako vam treba malo dodatnog prostora za definiranje vaše naredbe, slobodno proširite definiciju u više redova:

```
/**
 * Naziv i potpis naredbe konzole.
 *
 * @var string
 */
protected $signature = 'mail:send
                        {user : The ID of the user}
                        [--queue : Whether the job should be queued]';
```

### Upit za nedostajući unos

Ako vaša naredba sadrži potrebne argumente, korisnik će primiti poruku o grešci ako oni nisu navedeni. Alternativno, možete konfigurirati svoju naredbu da automatski pita korisnika kada nedostaju potrebni argumenti implementacijom `PromptsForMissingInput` interface-a:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Contracts\Console\PromptsForMissingInput;

class SendEmails extends Command implements PromptsForMissingInput
{
    /**
```

```
    * Naziv i potpis naredbe konzole.
    *
    * @var string
    */
    protected $signature = 'mail:send {user}';

    // ...
}
```

Ako Laravel treba prikupiti traženi argument od korisnika, automatski će od korisnika zatražiti argument inteligentnim formuliranjem pitanja korištenjem naziva ili opisa argumenta. Ako želite prilagoditi pitanje koje se koristi za prikupljanje potrebnih argumenata, možete implementirati metodu `promptForMissingArgumentsUsing`, vraćajući matricu pitanja ključnih imena argumenata:

```
/**
 * Pitaj za ulazne argumente koji nedostaju pomoću vraćenih pitanja.
 *
 * @return array
 */
protected function promptForMissingArgumentsUsing()
{
    return [
        'user' => 'Koji korisnički ID treba primiti mail?',
    ];
}
```

Također možete dati tekst rezerviranog mjesta korištenjem tuple (matrica fiksne veličine i poznatih tipova podataka) koja sadrži pitanje i rezervirano mjesto:

```
return [
    'user' => ['Koji ID korisnika treba primiti mail?', 'Npr. 123'],
];
```

Ako želite potpunu kontrolu nad upitom, možete omogućiti anonimne funkcije (engl. closure) koje bi trebalo upitati korisnika i vratiti njegov odgovor:

```
use App\Models\User;
use function Laravel\Prompts\search;

// ...

return [
    'user' => fn () => search(
```

```

        label: 'Traži korisnika:',
        placeholder: 'Npr. Taylor Otwell',
        options: fn ($value) => strlen($value) > 0
            ? User::where('name', 'like', "%{$value}%")->pluck('name', 'id')->all()
            : []
    ),
];

```

**NAPOMENA:**

Sveobuhvatna dokumentacija Laravel Prompts uključuje dodatne informacije o dostupnim prompt-ovima i njihovoj upotrebi.

Ako želite potaknuti (engl. prompt) korisnika da odabere ili unese opcije, možete uključiti upite u `handle` metodu svoje naredbe. Međutim, ako želite samo potaknuti (engl. prompt) korisnika kada su mu također automatski potaknuti (engl. prompt) argumenti koji nedostaju, tada možete implementirati `afterPromptingForMissingArguments` metodu:

```

use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use function Laravel\Prompts\confirm;

// ...

/**
 * Izvrši radnje nakon što se korisniku zatraže argumenti koji nedostaju.
 *
 * @param \Symfony\Component\Console\Input\InputInterface $input
 * @param \Symfony\Component\Console\Output\OutputInterface $output
 * @return void
 */
protected function afterPromptingForMissingArguments(InputInterface $input,
OutputInterface $output)
{
    $input->setOption('queue', confirm(
        label: 'Želite li staviti email u red (engl. queue)?',
        default: $this->option('queue')
    ));
}

```

## Komandni I/O

### Dohvaćanje unosa

Dok se vaša naredba izvršava, vjerojatno ćete morati pristupiti vrijednostima za argumente i opcije koje prihvaća vaša naredba. Da biste to učinili, možete koristiti `argument` i `option` metode. Ako argument ili opcija ne postoji, `null` će biti vraćeno:

```
/**
 * Izvrši konzolnu naredbu.
 */
public function handle(): void
{
    $userId = $this->argument('user');
}
```

Ako trebate dohvatiti sve argumente kao `array`, pozovite `arguments` metodu:

```
$arguments = $this->arguments();
```

Opcije se mogu dohvatiti jednako lako kao i argumenti korištenjem `option` metode. Da biste dohvatili sve opcije kao matricu, pozovite `options` metodu:

```
// Dohvati određene opcije.....
$queueName = $this->option('queue');

// Dohati sve opcije kao matricu...
$options = $this->options();
```

### Traženje unosa

#### NAPOMENA:



Laravel Prompts je PHP paket za dodavanje sjajnih formi jednostavnih za korištenje u vaše aplikacije komandnog reda, sa svojstvima sličnim browseru, uključujući rezerviranog mjesta za tekst i validacijsku provjeru.

Osim prikazivanja izlaza, također možete zatražiti od korisnika unos podataka tokom izvođenja vaše naredbe. Metoda `ask` će korisnika spremiti za prihvatanje naredbe (engl. prompt) na zadano pitanje, prihvatiti njihov unos a zatim vratiti korisnikov unos natrag u vašu naredbu:

```
/**
 * Izvrši naredbu konzole.
 */
public function handle(): void
{
    $name = $this->ask('Koje je vaše ime?');
```

```
// ...  
}
```

`ask` metoda također prihvaća opcionalni drugi argument koji specificira zadanu vrijednost koja bi trebala biti vraćena ako nije dostavljen korisnički unos:

```
$name = $this->ask('Koje je tvoje ime?', 'Taylor');
```

`secret` metoda je slična onoj `ask`, ali korisnikov unos neće biti vidljiv dok upisuje u konzolu. Ova je metoda korisna kada se traže osjetljivi podaci kao što su lozinke:

```
$password = $this->secret('Koja je šifra?');
```

### *Traženje potvrde*

Ako trebate od korisnika zatražiti jednostavnu potvrdu "da ili ne", možete koristiti ovu `confirm` metodu. Prema zadanim postavkama, ova će metoda vratiti `false`. Međutim, ako korisnik unese `y` ili `yes` kao odgovor na upit, metoda će se vratiti `true`.

```
if ($this->confirm('Da li želite nastaviti?')) {  
    // ...  
}
```

A

ko je potrebno, možete navesti da znak spremnosti za prihvatanje naredbe (engl. prompt) za potvrdu treba vratiti `true` prema zadanim postavkama s proslijeđivanjem `true` kao drugog argumenta `confirm` metodi:

```
if ($this->confirm('Da li želite nastaviti?', true)) {  
    // ...  
}
```

### *Automatsko dovršavanje*

Metoda `anticipate` se može koristiti za automatsko dovršavanje mogućih izbora. Korisnik i dalje može dati bilo koji odgovor, bez obzira na savjete za automatsko dovršavanje:

```
$name = $this->anticipate('Koje je vaše ime?', ['Taylor', 'Dayle']);
```

Alternativno, možete proslijediti anonimnu funkciju (engl. closure) kao drugi argument `anticipate` metode. Anonimna funkcija (engl. closure) će biti pozvano svaki puta kada korisnik upiše ulazni znak. Anonimna funkcija (engl. closure) bi trebala prihvatiti string parametar koji sadrži korisnički unos do sada i vratiti matricu opcija za automatsko dovršavanje:

```
$name = $this->anticipate('What is your address?', function (string $input) {  
    // Vraća automatski popunjene opcije
```

```
});
```

### *Pitanja s višestrukim izborom*

Ako korisniku trebate dati unaprijed definirani skup izbora prilikom postavljanja pitanja, možete koristiti `choice` metodu. Možete postaviti da se indeks matrice zadane vrijednosti vraća ako nije odabrana niti jedna opcija prosljeđivanjem indeksa kao trećeg argumenta metodi:

```
$name = $this->choice(  
    'Koje je vaše ime?',  
    ['Taylor', 'Dayle'],  
    $defaultIndex  
);
```

Pored toga, `choice` metoda izbora prihvća opcionalni četvrti i peti argument za određivanje maksimalnog broja pokušaja da se izabere validan odgovor i da li je višestruki odabir dozvoljen:

```
$name = $this->choice(  
    'Koje je vaše ime?',  
    ['Taylor', 'Dayle'],  
    $defaultIndex,  
    $maxAttempts = null,  
    $allowMultipleSelections = false  
);
```

### *Zapisivanje izlaza*

Za slanje izlaza na konzolu, možete koristiti metode `line`, `info`, `comment`, `question`, `warn` i `error`. Svaka od ovih metoda će koristiti odgovarajuće ANSI boje za svoju svrhu. Na primjer, prikažimo neke općenite informacije korisniku. Obično će `info` se metoda prikazati na konzoli kao tekst zelene boje:

```
/**  
 * Izvrši naredbu konzole.  
 */  
public function handle(): void  
{  
    // ...  
  
    $this->info('Ova naredba je uspješna!');  
}
```

Za prikaz poruke o grešci upotrijebite `error` metodu. Tekst poruke o grešci obično se prikazuje crveno:

```
$this->error('Nešto nije uredu!');
```

Možete koristiti `line` metodu za prikaz običnog, neobojenog teksta:

```
$this->line(' Prikaži ovo na ekranu');
```

Možete koristiti `newLine` metodu za prikaz praznog reda:

```
// Napiši jedan prazan red...
$this->newLine();

// Napiši tri prazna reda...
$this->newLine(3);
```

### Tablice

`table` metoda olakšava ispravno formatiranje više redova/stupaca podataka. Sve što trebate učiniti je dati nazive stupaca i podatke za tablicu i Laravel će automatski izračunati odgovarajuću širinu i visinu tablice za vas:

```
use App\Models\User;

$this->table(
    ['Name', 'Email'],
    User::all(['name', 'email'])->toArray()
);
```

### Trake koje prikazuju napredovanje

Za dugotrajne zadatke može biti korisno prikazati traku napretka (engl. progress bar, traka linije toka) koja obavještava korisnike koliko je zadatak dovršen. Koristeći `withProgressBar` metodu, Laravel će prikazati traku s napretkom i unaprijediti svoj napredak za svaku iteraciju preko za zadane [iterable](#)<sup>53</sup> vrijednost:

```
use App\Models\User;

$users = $this->withProgressBar(User::all(), function (User $user) {
    $this->performTask($user);
});
```

Ponekad ćete možda trebati više ručne kontrole nad napredovanjem trake koja prikazuje napredak. Prvo definirajte ukupan broj koraka kroz koje će proces iterirati. Zatim pomaknite traku koja pokazuje napredak nakon obrade svake stavke:

---

<sup>53</sup> Iterable je bilo koja vrijednost koja se može proći kroz petlju `foreach()`. Pseudotip koji se može ponavljati i koristiti kao tip podataka za argumente funkcije i povratne vrijednosti funkcije. Ključna riječ `iterable` može se koristiti kao tip podataka argumenta funkcije ili kao povratni tip funkcije.



```
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}

$bar->finish();
```

**NAPOMENA:**

Za naprednije opcije pogledajte [dokumentaciju Symfony Progress Bar komponente](#).

### Registriranje naredbi

Prema zadanim postavkama, Laravel automatski registrira sve naredbe unutar direktorija `app/Console/Commands`. Međutim, možete uputiti Laravel da skenira druge direktorije za Artisan naredbe pomoću metode `withCommands` u datoteci `bootstrap/app.php` vaše aplikacije:

```
->withCommands([
    __DIR__.'../app/Domain/Orders/Commands',
])
```

Ako je potrebno, također možete ručno registrirati naredbe davanjem naziva klase naredbe metodi `withCommands`:

```
use App\Domain\Orders\Commands\SendEmails;

->withCommands([
    SendEmails::class,
])
```

Kada se Artisan podigne, sve naredbe u vašoj aplikaciji razriješit će [kontejner usluga](#) i registrirati u Artisanu.

### Programsko izvršavanje naredbi

Ponekad ćete možda poželjeti izvršiti naredbu Artisan izvan interface-a komandne linije (CLI-ja). Na primjer, možda želite izvršiti `Artisan` naredbu iz rute ili kontrolera. Možete koristiti `call` metodu na Artisan fasadi da biste to postigli. `call` metoda prihvaća ili naziv potpisa naredbe ili naziv klase kao prvi argument, a matricu parametara naredbe kao drugi argument. Izlazni kod će biti vraćen:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    $exitCode = Artisan::call('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    // ...
});
```

Alternativno, možete proslijediti cijelu Artisan naredbu metodi `call` kao string:

```
Artisan::call('mail:send 1 --queue=default');
```

### Prosljeđivanje vrijednosti matrice

Ako vaša naredba definira opciju koja prihvaća matricu, možete proslijediti matricu vrijednosti toj opciji:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

### Prosljeđivanje Boolean vrijednosti

Ako trebate navesti vrijednost opcije koja ne prihvaća string vrijednosti, kao što je `--force` zastavica u `migrate:refresh` naredbi, trebate proslijediti `true` ili `false` kao vrijednost opcije:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

### Stavljanje u redoslijed (engl. *queuing*) Artisan naredbi

Koristeći `queue` metodu na `Artisan` fasadi, možete čak staviti Artisanove naredbe u red čekanja tako da se obrađuju u pozadini vaši queue radni slojevi. Prije korištenja ove metode provjerite jeste li konfigurirali svoj red čekanja i izvodite li reda čekanja slušatelj (engl. *queue listener*):

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function (string $user) {
    Artisan::queue('mail:send', [
        'user' => $user, '--queue' => 'default'
    ]);

    // ...
});
```

Koristeći `onConnection` i `onQueue` metode, možete odrediti vezu ili red čekanja na koji bi se Artisan naredba trebala poslati:

```
Artisan::queue('mail:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

### Pozivanje naredbi iz drugih naredbi

Ponekad ćete možda htjeti pozvati druge naredbe iz postojeće Artisan naredbe. To možete učiniti pomoću `call` metode. Ova `call` metoda prihvaća naziv naredbe i matricu argumenata/opcija naredbe:

```
/**
 * Izvrši konzolnu naredbu.
 */
public function handle(): void
{
    $this->call('mail:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    // ...
}
```

Ako želite pozvati drugu naredbu konzole i suzbiti sav njen izlaz, možete upotrijebiti `callSilently` metodu. `callSilently` metoda ima isti potpis kao `call` metoda:

```
$this->callSilently('mail:send', [
    'user' => 1, '--queue' => 'default'
]);
```

### Rukovanje signalima

Kao što možda znate, operativni sistemi dopuštaju slanje signala pokrenutim procesima. Na primjer, `SIGTERM` signal je kako operativni sistemi traže od programa da prekine rad. Ako želite slušati signale u svojim naredbama konzole Artisan i izvršiti kod kada se pojave, možete koristiti `trap` metodu:

```
/**
 * Izvrši konzolnu naredbu.
 */
public function handle(): void
{
    $this->trap(SIGTERM, fn () => $this->shouldKeepRunning = false);

    while ($this->shouldKeepRunning) {
        // ...
    }
}
```

Za slušanje više signala odjednom, možete dati matricu signala `trap` metodi:

```
$this->trap([SIGTERM, SIGQUIT], function (int $signal) {
    $this->shouldKeepRunning = false;

    dump($signal); // SIGTERM / SIGQUIT
});
```

### Stub prilagođavanje

Artisan konzolne `make` naredbe koriste se za stvaranje raznih klasa, kao što su kontroleri, poslovi (engl. jobs), migracije i testovi. Te se klase generiraju korištenjem „stub“ datoteka koje se popunjavaju vrijednostima na temelju vašeg unosa. Međutim, možda ćete htjeti napraviti male promjene u datotekama koje je generirao Artisan. Da biste to postigli, možete koristiti `stub:publish` naredbu za objavljivanje najčešćih dopuna u vašoj aplikaciji kako biste ih mogli prilagoditi:

```
php artisan stub:publish
```

Objavljeni stub-ovi bit će smješteni unutar `stubs` direktorija u korijenu vaše aplikacije. Sve promjene koje napravite na ovim stub-ovima odrazit će se kada generirate njihove odgovarajuće klase koristeći Artisanove `make` naredbe.

### Događaji

Artisan šalje tri događaja prilikom izvođenja naredbi:

```
Illuminate\Console\Events\ArtisanStarting,
Illuminate\Console\Events\CommandStarting i
```

`Illuminate\Console\Events\CommandFinished`. `ArtisanStarting` događaj se šalje odmah kada se Artisan pokrene. Zatim `CommandStarting` se događaj šalje neposredno prije pokretanja naredbe. Konačno, `CommandFinished` događaj se šalje kada naredba završi s izvršavanjem.

## Emitiranje (engl. Broadcasting)

### Uvod

U mnogim modernim web aplikacijama, WebSockets se koriste za implementaciju korisničkih interface-a koja se ažuriraju uživo u realnom vremenu. Kada se neki podaci ažuriraju na serveru, poruka se obično šalje pomoću WebSocket veze kojom upravlja klijent. WebSockets pružaju efikasniju alternativu stalnom ispitivanju servera vaše aplikacije za promjene podataka koje bi se trebali odraziti na vašem korisničkom interface-u.

Na primjer, zamislite da vaša aplikacija može izvesti korisničke podatke u CSV datoteku i poslati im ih e-poštom. Međutim, izrada ove CSV datoteke traje nekoliko minuta pa se odlučite za izradu i slanje CSV-a unutar posla na čekanju. Kada je CSV kreiran i poslan korisniku, možemo koristiti emitiranje događaja za slanje `App\Events\UserDataExported` događaja koji je primio JavaScript naše aplikacije. Nakon što se događaj primi, korisniku možemo prikazati poruku da mu je njegov CSV poslan e-poštom, a da on uopće ne mora osvježavati stranicu.

Kako bi vam pomogao u izgradnji ovih tipova svojstava, Laravel olakšava „emitiranje“ vaših Laravel događaja na strani servera pomoću WebSocket veze. Emitiranje vaših Laravel događaja omogućava vam dijeljenje istih naziva događaja i podataka između vaše Laravel aplikacije na strani servera i JavaScript aplikacije na strani klijenta.

Temeljni koncepti iza emitiranja su jednostavni: klijenti se povezuju na imenovane kanale na frontend-u, dok vaša Laravel aplikacija emitira događaje na te kanale na backend-u. Ti događaji mogu sadržavati sve dodatne podatke koje želite učiniti dostupnima frontend-u.

### Podržani driveri

Prema zadanim postavkama Laravel uključuje dva upravljačka programa za emitiranje na strani poslužitelja koje možete izabrati: [Pusher Channels](#) i [Ably](#). Međutim, paketi koje pokreće zajednica kao što je [soketi](#) obezbjeđuju dodatne drivere emitiranja koji ne zahtijevaju komercijalne pružatelje emitiranja (engl. broadcasting providers).

#### NAPOMENA:



Prije nego što se upustite u emitiranje događaja, provjerite jeste li pročitali Laravelovu dokumentaciju o događajima i slušateljima (engl. events and listeners).

### Instalacija na strani servera

Da bismo započeli s korištenjem Laravelovog emitiranja događaja, moramo napraviti neke konfiguracije unutar Laravel aplikacije kao i instalirati nekoliko paketa.

Emitiranje događaja postiže upravljački program za emitiranje na strani servera koji emitira vaše Laravel događaje tako da ih Laravel Echo (JavaScript biblioteka) može primiti unutar browser klijenta. Ne brinite - proći ćemo kroz svaki dio postupka instalacije korak po korak.

### Konfiguracija

Sva konfiguracija emitiranja događaja vaše aplikacije pohranjuje se u konfiguracijskoj datoteci `config/broadcasting.php`. Laravel podržava nekoliko driver-a za emitiranje van okvira: [Pusher](#)

[Channels](#), Redis i log driver za lokalni razvoj i otklanjanje grešaka. Dodatno, `null` driver koji vam omogućava da potpuno onemogućite emitiranje tokom testiranja. Primjer konfiguracije uključen je za svaki od ovih upravljačkih programa u `config/broadcasting.php` konfiguracijskoj datoteci.

#### *Emitovanje davatelja usluga*

Prije emitiranja bilo kojeg događaja, najprije ćete morati registrirati `App\Providers\BroadcastServiceProvider`. U novim Laravel aplikacijama trebate samo ukloniti komentare ovog davatelja usluga u `providers` matrici vaše `config/app.php` konfiguracijske datoteke. Ovaj `BroadcastServiceProvider` sadrži kod potreban za registraciju emitiranja autorizacijskih ruta i funkcija koja se prosljeđujete kao argument drugim funkcijama (eng. callbacks).

#### *Konfiguracija reda čekanja*

Također ćete morati konfigurirati i pokrenuti radnik reda čekanja (engl. queue worker). Emitiranje svih događaja vrši se kroz poslove na čekanju tako da na vrijeme odgovora vaše aplikacije ne utječu ozbiljno događaji koji se emituju.

#### **UPOZORENJE:**



Laravel Echo `reverb` emiter (engl. broadcaster) zahtijeva `laravel-echo v1.16.0+`.

#### *Reverb*

Prilikom pokretanja `install:broadcasting` naredbe, od vas će se tražiti da instalirate Laravel Reverb. Naravno, Reverb možete instalirati i ručno pomoću Composer-a upravitelja paketima:

```
composer require laravel/reverb
```

Nakon što je paket instaliran, možete pokrenuti Reverb-ovu instalacijsku naredbu da objelodanite konfiguraciju, dodate Reverb-ove potrebne varijable okoline (engl. environment variables) i omogućite emitiranje događaja u svojoj aplikaciji:

```
php artisan reverb:install
```

Detaljne upute o instalaciji i korištenju Reverba možete pronaći u [Reverb dokumentaciji](#).

#### *Pusher kanali*

Laravel Echo je JavaScript biblioteka koja olakšava pretplatu na kanale i slušanje događaja koje emitira vaš upravljački program (engl. driver) za emitiranje na strani servera. Echo također koristi paket `pusher-js` NPM za implementaciju Pusher protokola za WebSocket pretplate, kanale i poruke.

`install:broadcasting` Artisan naredba automatski instalira pakete `laravel-echo` i `pusher-js` za vas; međutim, ove pakete možete instalirati i ručno pomoću NPM-a:

```
npm install --save-dev laravel-echo pusher-js
```

Nakon što je Echo instaliran, spremni ste za kreiranje nove instance Echo u JavaScriptu svoje aplikacije. Naredba `install:broadcasting` stvara konfiguracijsku datoteku Echo na adresi `resources/js/echo.js`; međutim, zadana konfiguracija u ovoj datoteci namijenjena je za Laravel Reverb. Možete kopirati konfiguraciju u nastavku za prijenos vaše konfiguracije na Pusher:

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
  broadcaster: 'pusher',
  key: import.meta.env.VITE_PUSHER_APP_KEY,
  cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
  forceTLS: true
});
```

Zatim biste trebali definirati odgovarajuće vrijednosti za varijable okruženja Pusher u `.env` datoteci svoje aplikacije. Ako ove varijable već ne postoje u vašoj `.env` datoteci, trebali biste ih dodati:

```
PUSHER_APP_ID="vaš-pusher-app-id"
PUSHER_APP_KEY="vaš-pusher-key"
PUSHER_APP_SECRET="vaš-pusher-secret"
PUSHER_HOST=
PUSHER_PORT=443
PUSHER_SCHEME="https"
PUSHER_APP_CLUSTER="mt1"

VITE_APP_NAME="${APP_NAME}"
VITE_PUSHER_APP_KEY="${PUSHER_APP_KEY}"
VITE_PUSHER_HOST="${PUSHER_HOST}"
VITE_PUSHER_PORT="${PUSHER_PORT}"
VITE_PUSHER_SCHEME="${PUSHER_SCHEME}"
VITE_PUSHER_APP_CLUSTER="${PUSHER_APP_CLUSTER}"
```

Nakon što prilagodite konfiguraciju Echo-a prema potrebama svoje aplikacije, možete sastaviti materijale svoje aplikacije:

`config/broadcasting.php` datoteka pusher konfiguracije također također vam omogućava da odredite dodatne opcije Channels, kao što je klaster.

Zatim ćete morati promijeniti driver za emitiranje u pusher u vašoj `.env` datoteci:

```
npm run build
```



**NAPOMENA:** Kako biste saznali više o sastavljanju JavaScript materijala vaše aplikacije, pogledajte dokumentaciju na Viteu.



Korištenje instance postojećeg klijenta

Ako već imate unaprijed konfiguriranu instancu klijenta Pusher Channels koju želite da Echo koristi, možete je proslijediti Echo-u putem `client` konfiguracije opcije:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

const options = {
  broadcaster: 'pusher',
  key: 'your-pusher-channels-key'
}

window.Echo = new Echo({
  ...options,
  client: new Pusher(options.key, options)
});
```

#### *Pusher alternative otvorenog koda*

[soketi](#) pruža Pusher kompatibilan WebSocket server za Laravel, omogućavajući vam da iskoristite punu snagu Laravel emitiranja bez komercijalnog WebSocket provider-a. Za više informacija o instaliranju i korištenju paketa otvorenog koda za emitiranje, pogledajte našu dokumentaciju o [alternativama otvorenog koda](#).

#### *Ably*

**NAPOMENA:**



Dokumentacija u nastavku govori o tome kako koristiti Ably u „Pusher compatibility“ načinu rada. Međutim, Ably tim preporučuje i održava emitiranje i Echo klijenta koji može iskoristiti prednosti jedinstvenih mogućnosti koje nudi Ably. Za više informacija o korištenju driver-a koje održava Ably, pogledajte [Ably Laravel dokumentaciju emitiranja](#).

Laravel Echo je JavaScript biblioteka koja olakšava pretplatu na kanale i slušanje događaja koje emitira vaš upravljački program za emitiranje na strani poslužitelja. Echo također koristi paket pusher-js NPM za implementaciju Pusher protokola za WebSocket pretplate, kanale i poruke.

`install:broadcasting` Artisan naredba automatski instalira pakete `laravel-echo` i `pusher-js` za vas; međutim, ove pakete možete instalirati i ručno putem NPM-a:

```
npm install --save-dev laravel-echo pusher-js
```

Prije nego nastavite, trebali biste omogućiti podršku za protokol Pusher u postavkama aplikacije Ably. Ovu značajku možete omogućiti unutar dijela "Protocol Adapter Settings" ("Postavke adaptera protokola") na nadzornoj ploči (engl. dashboard) postavki vaše aplikacije Ably.

Nakon što je Echo instaliran, spremni ste za kreiranje nove instance Echo u JavaScriptu svoje aplikacije. Naredba `install:broadcasting` kreira konfiguracijsku datoteku Echo na adresi `resources/js/echo.js`; međutim, zadana konfiguracija u ovoj datoteci namijenjena je za Laravel Reverb. Možete kopirati konfiguraciju u nastavku da svoju konfiguraciju prebacite na Ably:

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
  broadcaster: 'pusher',
  key: import.meta.env.VITE_ABLY_PUBLIC_KEY,
  wsHost: 'realtime-pusher.ably.io',
  wsPort: 443,
  disableStats: true,
  encrypted: true,
});
```

Možda ste primijetili da se naša konfiguracija Ably Echo poziva na varijablu okoline `VITE_ABLY_PUBLIC_KEY`. Vrijednost ove varijable trebala bi biti vaš javni ključ Ablyja. Vaš javni ključ je dio vašeg ključa Ably koji se nalazi prije znaka `:`.

Nakon što prilagodite Echo konfiguraciju prema svojim potrebama, možete sastaviti materijale svoje aplikacije:

```
npm run dev
```

#### NAPOMENA:



Kako biste saznali više o sastavljanju JavaScript sredstava vaše aplikacije, pogledajte dokumentaciju o [Vite](#).

#### Instalacija na strani klijenta

##### *Reverb*

Laravel Echo je JavaScript biblioteka koja olakšava pretplatu na kanale i slušanje događaja koje emitira vaš upravljački program za emitiranje na strani servera. Echo možete instalirati putem NPM upravitelja paketa (engl. package manager). U ovom primjeru također ćemo instalirati paket `pusher-js` budući da Reverb koristi protokol Pusher za WebSocket pretplate, kanale i poruke:

```
npm install --save-dev laravel-echo pusher-js
```

Nakon što je Echo instaliran, spremni ste za stvaranje nove instance Echo u JavaScriptu svoje aplikacije. Odlično mjesto za to je na dnu datoteke `resources/js/bootstrap.js` koja je uključena u okvir Laravel. Prema zadanim postavkama, primjer konfiguracije Echo već je uključen u ovu datoteku - jednostavno je trebate odkomentirati i ažurirati `broadcaster` opciju konfiguracije na `reverb`:

```
import Echo from 'laravel-echo';

import Pusher from 'pusher-js';
window.Pusher = Pusher;

window.Echo = new Echo({
  broadcaster: 'reverb',
  key: import.meta.env.VITE_REVERB_APP_KEY,
  wsHost: import.meta.env.VITE_REVERB_HOST,
  wsPort: import.meta.env.VITE_REVERB_PORT,
  wssPort: import.meta.env.VITE_REVERB_PORT,
  forceTLS: (import.meta.env.VITE_REVERB_SCHEME ?? 'https') === 'https',
  enabledTransports: ['ws', 'wss'],
});
```

Zatim biste trebali sastaviti sredstva svoje aplikacije:

```
npm run build
```

### *Pusher kanali*

[Laravel Echo](#) je JavaScript biblioteka koja olakšava pretplatu na kanale i slušanje događaja koje emitira vaš upravljački program za emitiranje na strani servera. Echo možete instalirati putem NPM upravitelja paketa. U ovom primjeru također ćemo instalirati `pusher-js` paket jer ćemo koristiti emiter Pusher Channels:

```
npm install --save-dev laravel-echo pusher-js
```

Nakon što je Echo instaliran, spremni ste za kreiranje nove Echo instance u JavaScriptu svoje aplikacije. Odlično mjesto za to je na dnu datoteke `resources/js/bootstrap.js` koja je uključena u okvir Laravel. Prema zadanim postavkama, primjer Echo konfiguracije već je uključen u ovu datoteku - jednostavno trebate ukloniti komentare:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
  broadcaster: 'pusher',
```

```
key: import.meta.env.VITE_PUSHER_APP_KEY,
cluster: import.meta.env.VITE_PUSHER_APP_CLUSTER,
forceTLS: true
});
```

Nakon što uklonite komentare i prilagodite Echo konfiguraciju prema svojim potrebama, možete kompajlirati vaša aplikacijske materijale (engl. asset):

```
npm run build
```

#### NAPOMENA:



Da biste saznali više o kompajliranju JavaScript sredstava vaše aplikacije, pogledajte dokumentaciju o [Vite](#).

#### Korištenje postojećih instanci klijenta

Ako već imate unaprijed konfiguriranu instancu klijenta Pusher Channels koju želite da Echo koristi, možete je proslijediti Echo-u pomoću `client` konfiguracijske opcije:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

const options = {
  broadcaster: 'pusher',
  key: 'your-pusher-channels-key'
}

window.Echo = new Echo({
  ...options,
  client: new Pusher(options.key, options)
});
```

#### Ably

#### NAPOMENA:



Dokumentacija u nastavku govori o tome kako koristiti Ably u načinu rada „Pusher compatibility“. Međutim, Ably tim preporučuje i održava emitera i Echo klijenta koji može iskoristiti prednosti jedinstvenih mogućnosti koje nudi Ably. Za više informacija o korištenju drivera koje održava Ably, [pogledajte dokumentaciju Ably-jevog Laravel emitera](#).

[Laravel Echo](#) je JavaScript biblioteka koja olakšava pretplatu na kanale i slušanje događaja koje emitira vaš driver za emitiranje na strani poslužitelja. Echo možete instalirati pomoću NPM upravitelja paketa. U ovom primjeru također ćemo instalirati `pusher-js` paket.

Možda se pitate zašto bismo instalirali `pusher-js` JavaScript biblioteku iako koristimo Ably za emitiranje naših događaja. Srećom, Ably uključuje Pusher kompatibilni način rada koji nam omogućava korištenje Pusher protokola kada osluškujemo događaje u našoj aplikaciji na strani klijenta:

```
npm install --save-dev laravel-echo pusher-js
```

**Prije nego nastavite, trebali biste omogućiti podršku za Pusher protokol u postavkama Ably aplikacije. Ovu svojstvo možete omogućiti unutar dijela „Postavke adaptera protokola“ na nadzornoj tabli postavki vaše Ably aplikacije.**

Nakon što je Echo instaliran, spremni ste za kreiranje nove Echo instance u JavaScriptu vaše aplikacije. Odlično mjesto za to je na dnu datoteke `resources/js/bootstrap.js` koja je uključena u Laravel radnu okolinu. Prema zadanim postavkama, primjer konfiguracije Echo već je uključen u ovu datoteku; međutim, zadana konfiguracija u `bootstrap.js` datoteci namijenjena je Pusher-u. Možete kopirati konfiguraciju u nastavku da svoju konfiguraciju prebacite u Ably:

```
import Echo from 'laravel-echo';
import Pusher from 'pusher-js';

window.Pusher = Pusher;

window.Echo = new Echo({
  broadcaster: 'pusher',
  key: import.meta.env.VITE_ABLY_PUBLIC_KEY,
  wsHost: 'realtime-pusher.ably.io',
  wsPort: 443,
  disableStats: true,
  encrypted: true,
});
```

Primijetite da se naša konfiguracija Ably Echo odnosi na `VITE_ABLY_PUBLIC_KEY` varijablu okoline. Vrijednost ove varijable trebala bi biti vaš Ably javni ključ. Vaš javni ključ je dio vašeg Ably ključa koji se nalazi prije `:` znaka.

Nakon što uklonite komentare i prilagodite Echo konfiguraciju prema svojim potrebama, možete kompajlirati materijale vaše aplikacije:

```
npm run dev
```

#### NAPOMENA:



Da biste saznali više o kompajliranju JavaScript materijala vaše aplikacije, pogledajte dokumentaciju o [Vite](#).

#### Pregled koncepta

Laravelovo emitiranje događaja omogućava vam da emitirate svoje Laravel događaje na strani servera ka svojoj JavaScript aplikaciji na strani klijenta koristeći pristup WebSocketsu koji se temelji na

upravljačkom programu. Trenutno se Laravel isporučuje s [Pusher Channels](#) i [Ably](#) drajverima. Događaji se mogu jednostavno konzumirati na strani klijenta pomoću [Laravel Echo](#) JavaScript paketa.

Događaji se emitiraju preko „kanala“, koji mogu biti javni ili privatni. Svaki posjetitelj vaše aplikacije može se pretplatiti na javni kanal bez ikakve provjere autentičnosti ili autorizacije; međutim, da bi se pretplatilo na privatni kanal, korisnik mora biti autentificiran i ovlašten za slušanje na tom kanalu.

### Korištenje primjera aplikacije

Prije nego što zaronimo u svaku komponentu emitiranja događaja, uzmimo pregled na visokoj razini koristeći trgovinu e-trgovine kao primjer.

U našoj aplikaciji, pretpostavimo da imamo stranicu koja korisnicima omogućuje pregled statusa otpreme za njihove narudžbe. Pretpostavimo također da se `OrderShipmentStatusUpdated` događaj aktivira kada aplikacija obradi ažuriranje statusa otpreme:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

### `ShouldBroadcast` Interface

Kada korisnik gleda neku od svojih naloga (engl. orders), ne želimo da mora osvježavati stranicu kako bi vidio ažuriranja statusa. Umjesto toga, želimo emitirati (engl. broadcast) ažuriranja aplikacije kako se stvaraju. Dakle, trebamo označiti `OrderShipmentStatusUpdated` događaj interface-om `ShouldBroadcast`. Ovo će uputiti Laravel da emitira događaj kada se pokrene:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;
}
```

Interface `ShouldBroadcast` zahtijeva naš događaj za definiranje `broadcastOn` metode. Ova metoda je odgovorna za vraćanje kanala na kojima bi se događaj trebao emitirati. Prazan stub ove metode već je definiran na generiranim klasama događaja, tako da samo trebamo ispuniti njegove detalje. Želimo da samo kreator naloga može vidjeti ažuriranja statusa, zato ćemo događaj emitirati na privatnom kanalu koji je vezan uz nalog:

```
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Get the channel the event should broadcast on.
 */
public function broadcastOn(): Channel
{
    return new PrivateChannel('orders.'.$this->order->id);
}
```

Ako želite da se događaj emitira na više kanala, umjesto toga možete vratiti `array`:

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Get the channels the event should broadcast on.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(): array
{
    return [
        new PrivateChannel('orders.'.$this->order->id),
        // ...
    ];
}
```

### *Autoriziranje kanala*

Zapamtite, korisnici moraju biti ovlašteni za slušanje na privatnim kanalima. Možemo definirati naša pravila autorizacije kanala u `routes/channels.php` datoteci naše aplikacije. U ovom primjeru moramo provjeriti je li svaki korisnik koji pokušava slušati na privatnom `orders.1` kanalu zapravo kreator naloga:

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
```

```
return $user->id === Order::findOrNew($orderId)->user_id;
});
```

Metoda `channel` prihvaća dva argumenta: naziv kanala i funkciju koja se proslijeđujete kao argument drugim funkcijama (eng. callback). koji vraća `true` ili `false` pokazuje da li je korisnik ovlašten slušati na kanalu.

Svi autorizacijski callback-ovi primaju trenutno provjerenog korisnika kao svoj prvi argument i sve dodatne parametre zamjenskih znakova (engl. wildcards) kao svoje sljedeće argumente. U ovom primjeru koristimo `{orderId}` rezervirano mjesto kako bismo označili da je "ID" dio naziva kanala zamjenski znak.

### *Slušanje prijenosa događaja*

Zatim, sve što preostaje je slušati događaj u našoj JavaScript aplikaciji. To možemo učiniti koristeći Laravel Echo. Prvo ćemo upotrijebiti privatnu metodu za pretplatu na privatni kanal. Zatim možemo koristiti metodu slušanja za slušanje `OrderShipmentStatusUpdated` događaja. Prema zadanim postavkama, sva javna svojstva događaja bit će uključena u događaj emitiranja:

```
Echo.private(`orders.${orderId}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order);
    });
```

### *Definiranje događaja emitiranja*

Da biste obavijestili Laravel da se dati događaj treba emitirati, morate implementirati `Illuminate\Contracts\Broadcasting\ShouldBroadcast` interface na klasu događaja. Ovaj interface već je uvezeno u sve klase događaja koje je generirala radna okolina tako da ga možete jednostavno dodati bilo kojem od svojih događaja.

Interface `ShouldBroadcast` zahtijeva implementaciju jedne metode: `broadcastOn`. Metoda `broadcastOn` bi trebala vratiti kanal ili matricu kanala na kojima bi se događaj trebao emitirati. Kanali bi trebali biti primjerci `Channel`, `PrivateChannel` ili `PresenceChannel`. Instance `Channel` predstavljaju javne kanale na koje se svaki korisnik može pretplatiti, dok `PrivateChannels` i `PresenceChannels` predstavljaju privatne kanale koji zahtijevaju autorizaciju kanala:



```
<?php

namespace App\Events;

use App\Models\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast
{
    use SerializesModels;

    /**
     * Kreiraj novu instancu događaja.
     */
    public function __construct(
        public User $user,
    ) {}

    /**
     * Nabavi kanale na kojima bi se događaj trebao emitirati.
     *
     * @return array<int, \Illuminate\Broadcasting\Channel>
     */
    public function broadcastOn(): array
    {
        return [
            new PrivateChannel('user.'.$this->user->id),
        ];
    }
}
```

Nakon implementacije `ShouldBroadcast` interface-a, trebate samo pokrenuti događaj kao što biste inače. Nakon što se događaj pokrene, posao u redu čekanja automatski će emitirati događaj koristeći vaš navedeni upravljački program za emitiranje.

### Ime emitiranja

Prema zadanim postavkama, Laravel će emitirati događaj koristeći naziv klase događaja. Međutim, naziv emitiranja možete prilagoditi definiranjem `broadcastAs` metode na događaju:

```
/**
 * Ime prijenosa događaja.
 */
public function broadcastAs(): string
{
    return 'server.created';
}
```

Ako naziv emitiranja prilagodite metodom `broadcastAs`, trebali biste registrirati svog slušatelja s vodećim `.` znakom. Ovo će uputiti Echo da ne doda prostor imena aplikacije ispred događaja:

```
.listen('server.created', function (e) {
    ....
});
```

### Emitirani podaci (engl. Broadcast data)

Kada se događaj emitira, sva njegova `public` svojstva automatski se serijaliziraju i emitiraju kao sadržaj događaja, što vam omogućuje pristup svim njegovim javnim podacima iz vaše JavaScript aplikacije. Tako, na primjer, ako vaš događaj ima jedno javno `$user` svojstvo koje sadrži model Eloquent, skup korisničkih informacija emitiranja (engl. payload broadcast) događaja bila bi:

```
{
  "user": {
    "id": 1,
    "name": "Zdravko Čolić"
    ...
  }
}
```

Međutim, ako želite imati finiju kontrolu nad sadržajem emitiranja, možete dodati `broadcastWith` metodu svom događaju. Ova bi metoda trebala vratiti matricu podataka koje želite emitirati kao skup korisničkih informacija događaja (engl. event payload):

```
/**
 * Dobij podatke za emitiranje.
 *
 * @return array<string, mixed>
 */
```

```
public function broadcastWith(): array
{
    return ['id' => $this->user->id];
}
```

#### *Red čekanja za emitiranje (engl. Broadcast Queue)*

Prema zadanim postavkama, svaki događaj emitiranja stavlja se u zadani red čekanja za zadanu vezu reda čekanja navedenu u vašoj `queue.php` konfiguracijskoj datoteci. Možete prilagoditi vezu reda čekanja (engl. queue connection) i naziv koji koristi emiter (engl. broadcaster) definiranjem `connection` i `queue` svojstava na vašoj klasi događaja:

```
/**
 * Naziv reda čekanja (engl. queue)konekcije za korištenje kada emitiramo događaj.
 *
 * @var string
 */
public $connection = 'redis';

/**
 * Naziv reda čekanja (engl. queue) u koji treba smjestiti */
* posao emitiranja (engl. broadcasting job).
 *
 * @var string
 */
public $queue = 'default';
```

Alternativno, možete prilagoditi naziv reda čekanja (engl. queue name) definiranjem `broadcastQueue` metode na vašem događaju:

```
/**
 * Naziv reda čekanja (engl. queue ) u koji treba smjestiti */
* posao emitiranja (engl. broadcasting job). */
public function broadcastQueue(): string
{
    return 'default';
}
```

Ako želite emitirati svoj događaj koristeći sync red čekanja (engl. queue) umjesto zadanog driver-a reda čekanja, možete implementirati `ShouldBroadcastNow` interface umjesto `ShouldBroadcast`:

```
<?php

use Illuminate\Contracts\Broadcasting\ShouldBroadcastNow;

class OrderShipmentStatusUpdated implements ShouldBroadcastNow
{
    // ...
}
```

### *Uslovi emitiranja*

Ponekad želite emitirati svoj događaj samo ako je zadani uslov istinit. Ove uvjete možete definirati dodavanjem `broadcastWhen` metode svojoj klasi događaja:

```
/**
 * Odredi treba li ovaj događaj emitirati.
 */
public function broadcastWhen(): bool
{
    return $this->order->value > 100;
}
```

### *Transakcije emitiranja i baze podataka*

Kada se događaji emitiranja šalju unutar transakcija baze podataka, oni se mogu obraditi u redu čekanja (engl. queue) prije nego što se transakcija baze podataka izvrši (engl. committed.). Kada se to dogodi, sva ažuriranja koja ste izvršili na modelima ili zapisima baze podataka tokom transakcije baze podataka možda se još neće reflektirati u bazi podataka. Osim toga, bilo koji modeli ili zapisi baze podataka kreirani unutar transakcije možda neće postojati u bazi podataka. Ako vaš događaj ovisi o ovim modelima, mogu se pojaviti neočekivane greške kada se obradi posao koji emitira događaj.

Ako je konfiguracijska opcija vaše veze s redom čekanja `after_commit` postavljena na `false`, još uvijek možete naznačiti da se određeni događaj emitiranja treba poslati nakon što su sve transakcije otvorene baze podataka izvršene implementacijom interface-a `ShouldDispatchAfterCommit` na klasu događaja:

```
<?php

namespace App\Events;

use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Contracts\Events\ShouldDispatchAfterCommit;
use Illuminate\Queue\SerializesModels;

class ServerCreated implements ShouldBroadcast, ShouldDispatchAfterCommit
{
}
```

```
use SerializesModels;
}
```

**NAPOMENA:**

Da biste saznali više o zaobilazanju ovih problema, pregledajte dokumentaciju koja se odnosi na poslove čekanja i transakcije baze podataka .

*Autorizacija kanala*

Privatni kanali zahtijevaju da autorizirate da trenutno autentificirani korisnik može stvarno slušati na kanalu. To se postiže slanjem HTTP zahtjeva vašoj Laravel aplikaciji s nazivom kanala i dopuštanjem vašoj aplikaciji da odredi može li korisnik slušati na tom kanalu. Kada koristite [Laravel Echo](#), HTTP zahtjev za autorizaciju pretplata na privatne kanale bit će napravljen automatski.

Kada je emitiranje omogućeno, Laravel automatski registrira `/broadcasting/auth` rutu za obradu zahtjeva za autorizaciju. Ruta `/broadcasting/auth` se automatski postavlja unutar `web` middleware grupe.

*Definiranje autorizacijskih ruta*

Zatim moramo definirati logiku koja će zapravo odrediti može li trenutno autentificirani korisnik slušati dati kanal. To se radi u `routes/channels.php` datoteci koju je stvorila `install:broadcasting` Artisan naredba. U ovoj datoteci možete koristiti `Broadcast::channel` metodu za registraciju funkcija koja se proslijeđuju kao argument drugim funkcijama (eng. callback) autorizacije kanala:

```
use App\Models\User;

Broadcast::channel('orders.{orderId}', function (User $user, int $orderId) {
    return $user->id === Order::findOrCreate($orderId)->user_id;
});
```

Metoda `channel` prihvaća dva argumenta: naziv kanala i callback koji vraća `true` ili `false` pokazuje da li je korisnik ovlašten slušati na kanalu.

Svi autorizacijski callback-ovi primaju trenutno provjerenog korisnika kao svoj prvi argument i sve dodatne parametre zamjenskih znakova kao svoje sljedeće argumente. U ovom primjeru koristimo `{orderId}` rezervirano mjesto kako bismo označili da je "ID" dio naziva kanala zamjenski znak (engl. wildcard).

Možete vidjeti listu aplikacijskih autoriziranih callback-ova emitiranja povratnih poziva autorizacije pomoću `channel:list` naredbe Artisan:

```
php artisan channel:list
```

*Vežanje modela povratnog poziva autorizacije*

Baš kao i HTTP rute, rute kanala također mogu iskoristiti prednost implicitnog i eksplicitnog [uvezivanja modela rute \(engl. route model binding\)](#). Na primjer, umjesto primanja stringa ili numeričkog ID-a naloga, možete zatražiti stvarnu `Order` instancu modela:

```
use App\Models\Order;
use App\Models\User;

Broadcast::channel('orders.{order}', function (User $user, Order $order) {
    return $user->id === $order->user_id;
});
```

**UPOZORENJE:**

Za razliku od uvezivanja (engl. binding) modela HTTP rute, uvezivanje modela kanala ne podržava automatsko [određivanje dosega implicitnog uvezivanja modela](#). Međutim, to je rijetko problem jer se većini kanala može odrediti opseg na temelju jedinstvenog, primarnog ključa jednog modela.

*Autorizacija Callback autentifikacije*

Privatni kanali i kanali za emitiranje prisutnosti autentificiraju tekućeg korisnika pomoću zadane zaštite autentifikacije vaše aplikacije. Ako korisnik nije autentificiran, autorizacija kanala automatski se odbija i autorizani callback nikada se ne izvršava. Međutim, možete dodijeliti više prilagođenih čuvara koji bi trebali potvrditi autentičnost dolaznog zahtjeva ako je potrebno:

```
Broadcast::channel('channel', function () {
    // ...
}, ['guards' => ['web', 'admin']]);
```

*Definiranje klase kanala*

Ako vaša aplikacija koristi mnogo različitih kanala, vaša `routes/channels.php` i datoteka mogla postati glomazna. Dakle, umjesto korištenja closures za autorizaciju kanala, možete koristiti klase kanala. Za generiranje klase kanala upotrijebite `make:channel` Artisan naredbu. Ova naredba će postaviti novu klasu kanala u `App/Broadcasting` direktorij.

```
php artisan make:channel OrderChannel
```

Zatim registrirajte svoj kanal u svojoj `routes/channels.php` datoteci:

```
use App\Broadcasting\OrderChannel;

Broadcast::channel('orders.{order}', OrderChannel::class);
```

Konačno, možete postaviti logiku autorizacije za vaš kanal u `join` metodu klase kanala. Ova `join` će metoda sadržavati istu logiku koju biste obično postavili u zatvaranje autorizacije kanala. Također možete iskoristiti prednosti uvezivanja modela kanala:

```
<?php

namespace App\Broadcasting;

use App\Models\Order;
use App\Models\User;

class OrderChannel
{
    /**
     * Kreiraj novu instancu kanala.
     */
    public function __construct() {}

    /**
     * Autentificirajte korisnički pristup kanalu.
     */
    public function join(User $user, Order $order): array|bool
    {
        return $user->id === $order->user_id;
    }
}
```

#### NAPOMENA:



Kao i mnoge druge klase u Laravelu, klase kanala automatski će razriješiti [konteiner usluge](#). Dakle, možete nagovijestiti tip (engl. type-hint) sve zavisnosti (engl. dependency) koje zahtijeva vaš kanal u njegovom konstruktoru.

#### Emitiranje događaja

Nakon što ste definirali događaj i označili ga `ShouldBroadcast` interface-om, samo trebate pokrenuti događaj koristeći metodu slanja događaja. Dispečer događaja (engl. event dispatcher) će primijetiti da je događaj označen `ShouldBroadcast` interfaceom i staviti će ga u red čekanja (engl. queue) za emitiranje:

```
use App\Events\OrderShipmentStatusUpdated;

OrderShipmentStatusUpdated::dispatch($order);
```

### Only za druge

Prilikom izrade aplikacije koja koristi emitiranje događaja, možda ćete povremeno trebati emitirati događaj svim pretplatnicima određenog kanala osim tekućem korisniku. To možete postići pomoću broadcast pomoćnika i `toOthers` metode:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->toOthers();
```

Da biste bolje razumjeli kada biste trebali upotrijebiti `toOthers` metodu, zamislimo aplikaciju s popisom zadataka u kojoj korisnik može stvoriti novi zadatak unosom naziva zadatka. Za izradu zadatka, vaša aplikacija može uputiti zahtjev na `/task` URL koji emitira stvaranje zadatka i vraća JSON prikaz novog zadatka. Kada vaša JavaScript aplikacija primi odgovor od krajnje točke, može direktno umetnuti novi zadatak u svoj popis zadataka ovako:

```
axios.post('/task', task)
  .then((response) => {
    this.tasks.push(response.data);
  });
```

Međutim, zapamtite da također emitiramo izradu zadatka. Ako vaša JavaScript aplikacija također osluškuje ovaj događaj kako bi dodala zadatke na popis zadataka, imat ćete duplicirane zadatke na popisu: jedan s krajnje točke i jedan s emitiranja. To možete riješiti korištenjem `toOthers` metode da naložite emiteru da ne emitira događaj trenutnom korisniku.

#### UPOZORENJE:



Vaš događaj mora koristiti `Illuminate\Broadcasting\InteractsWithSockets` svojstvo (engl. trait) kako bi pozvao `toOthers` metodu.

### Konfiguracija

Kada inicijalizirate Laravel Echo instancu, ID utora (engl. socket) se dodjeljuje konekciji. Ako koristite globalnu `Axios` instancu za upućivanje HTTP zahtjeva iz vaše JavaScript aplikacije, ID utor će automatski biti priložen svakom odlaznom zahtjevu kao `X-Socket-ID` zaglavlje. Zatim, kada pozovete `toOthers` metodu, Laravel će izdvojiti ID utor iz zaglavlja i dati instrukciju emiteru da ne emitira ni na jednu vezu s tim ID-om utora.

Ako ne koristite globalnu instancu Axios, morat ćete ručno konfigurirati svoju JavaScript aplikaciju za slanje `X-Socket-ID` zaglavlja sa svim odlaznim zahtjevima. Možete dohvatiti ID utora pomoću `Echo.socketId` metode:

```
var socketId = Echo.socketId();
```



### Prilagođavanje konekcije

Ako vaša aplikacija komunicira s višestrukim vezama za emitiranje i želite emitirati događaj koristeći emitera koji nije vaš zadani, možete odrediti na koju ćete konekciju proslijediti događaj pomoću `via` metode:

```
use App\Events\OrderShipmentStatusUpdated;

broadcast(new OrderShipmentStatusUpdated($update))->via('pusher');
```

Alternativno, možete navesti emitiranu vezu događaja pozivanjem `broadcastVia` metode unutar konstruktora događaja. Međutim, prije nego što to učinite, trebali biste osigurati da klasa događaja koristi `InteractsWithBroadcasting` svojstvo (engl. trait):

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithBroadcasting;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Queue\SerializesModels;

class OrderShipmentStatusUpdated implements ShouldBroadcast
{
    use InteractsWithBroadcasting;

    /**
     * Kreiraj novu instancu događaja.
     */
    public function __construct()
    {
        $this->broadcastVia('pusher');
    }
}
```

### Anonimni događaji

Ponekad ćete možda htjeti emitirati jednostavan događaj na frontend svoje aplikacije bez kreiranja posvećene klase događaja. Kako bi se to prilagodilo, `Broadcast` fasada vam omogućava emitiranje "anonimnih događaja":

```
Broadcast::on('orders.' . $order->id)->send();
```

Prethodni primjer će emitirati sljedeći događaj:

```
{
  "event": "AnonymousEvent",
  "data": "[]",
  "channel": "orders.1"
}
```

Pomoću metoda `as` i `with` možete prilagoditi naziv i podatke događaja:

```
Broadcast::on('orders.'.$order->id)
    ->as('OrderPlaced')
    ->with($order)
    ->send();
```

Broadcast je fasada fasada (pristupna tačka) koja omogućava emitiranje događaja u Laravelu putem različitih drajvera, kao što su Pusher, Redis, itd.

Primjer iznad će emitirati događaj kao ovaj:

```
{
  "event": "OrderPlaced",
  "data": "{ id: 1, total: 100 }",
  "channel": "orders.1"
}
```

Ako želite emitirati anonimni događaj na privatnom kanalu ili kanalu prisutnosti (engl. presence channel), možete koristiti metode `private` i `presence`:

```
Broadcast::private('orders.'.$order->id)->send();
Broadcast::presence('channels.'.$channel->id)->send();
```

**Op. prev.:** Kanali prisutnosti su posebna vrsta kanala u Laravelu koji omogućavaju praćenje identiteta korisnika koji su trenutno povezani s određenim kanalom. Oni proširuju funkcionalnost privatnih kanala tako što, pored toga što ograničavaju pristup samo ovlaštenim korisnicima, omogućavaju da saznate tko je točno povezan s kanalom u realnom vremenu. Ključna razlika između privatnih kanala i kanala prisutnosti je u tome što kanali prisutnosti omogućavaju praćenje i emitiranje informacija o korisnicima koji su povezani s kanalom, dok privatni kanali samo ograničavaju pristup emitiranim događajima bez dodatnih informacija o prisutnim korisnicima.

Emitiranje anonimnog događaja pomoću `send` metode šalje događaj u red čekanja vaše aplikacije na obradu. Međutim, ako želite odmah emitirati događaj, možete koristiti metodu `sendNow`:

```
Broadcast::on('orders.'.$order->id)->sendNow();
```

Za emitiranje događaja svim pretplatnicima kanala osim trenutno autentificiranom korisniku, možete pozvati metodu `toOthers`:

```
Broadcast::on('orders.'.$order->id)
    ->toOthers()
    ->send();
```

### *Primanje emitiranja*

#### *Slušanje događaja*

Nakon što ste [instalirali i instancirali Laravel Echo](#), spremni ste za početak osluškivanja događaja koji se emitiraju iz vaše Laravel aplikacije. Prvo upotrijebite `channel` metodu za dohvaćanje instance kanala, zatim pozovite `listen` metodu za slušanje određenog događaja:

```
Echo.channel(`orders.${this.order.id}`)
    .listen('OrderShipmentStatusUpdated', (e) => {
        console.log(e.order.name);
    });
```

Ako želite slušati događaje na privatnom kanalu, upotrijebite ovu `private` metodu. Možete nastaviti s ulančanim pozivima metode `listen` za slušanje više događaja na jednom kanalu:

```
Echo.private(`orders.${this.order.id}`)
    .listen(/* ... */)
    .listen(/* ... */)
    .listen(/* ... */);
```

**Op. prev.:** `Echo.private()` označava povezivanje na privatni kanal koristeći Echo. Sintaksa ``orders.${this.order.id}`` koristi template string u JavaScript-u, gde se dinamički ubacuje `order.id` (ID naloga) u ime kanala. To znači da Echo kreira privatni kanal pod nazivom npr. `orders.123`, ako je `order.id` 123.

Privatni kanal je zaštićen, što znači da samo ovlašćeni korisnici mogu slušati događaje na njemu. Metoda `listen()` se koristi za slušanje događaja koji su emitovani na kanalu. Svaki `listen()` poziva određeni događaj na kanalu i obavlja neku akciju kada se taj događaj dogodi. U ovom slučaju, postoje tri poziva metode `listen()`, što znači da aplikacija sluša tri različita događaja na privatnom kanalu `orders.<ID>`.

### *Zaustavljanje slušanja za događaje*

Ako želite prestati slušati određeni događaj bez [napuštanja kanala](#), možete upotrijebiti `stopListening` metodu:

```
Echo.private(`orders.${this.order.id}`)
    .stopListening('OrderShipmentStatusUpdated')
```

### Napuštanje kanala

Da biste napustili kanal, možete pozvati `leaveChannel` metodu na svojoj Echo instanci:

```
Echo.leaveChannel(`orders.${this.order.id}`);
```

Ako želite napustiti kanal i također povezane privatne kanale i kanale prisutnosti, možete pozvati `leave` metodu:

```
Echo.leave(`orders.${this.order.id}`);
```

### Imenski prostori (engl. Namespaces)

Možda ste primijetili u prethodnim primjerima da nismo specificirali puni `App\Events` imenski prostor za klase događaja. To je zato što će Echo automatski pretpostaviti da se događaji nalaze u `App\Events` imenskom prostoru. Međutim, možete konfigurirati korijenski imenski prostor kada instancirate Echo prosljeđivanjem `namespace` konfiguracijske opcije:

```
window.Echo = new Echo({
  broadcaster: 'pusher',
  // ...
  namespace: 'App.Other.Namespace'
});
```

Alternativno, klasama događaja možete dodati prefiks `.` kada se na njih pretplaćujete koristeći Echo. To će vam omogućiti da uvijek navedete potpuno kvalificirano ime klase:

```
Echo.channel('orders')
  .listen('.Namespace\\Event\\Class', (e) => {
    // ...
  });
```

### Kanali prisutnosti

Kanali prisutnosti grade se na sigurnosti privatnih kanala dok izlažu dodatnu mogućnost svijesti o tome tko je pretplaćen na kanal. To olakšava izgradnju snažnih svojstava suradničke aplikacije kao što je obavještanje korisnika kada drugi korisnik gleda istu stranicu ili izlista prisutne u chat sobi.

### Autoriziranje kanala prisutnosti

Svi kanali prisutnosti također su privatni kanali; zato korisnici moraju biti [ovlašteni za pristup njima](#). Međutim, kada definirate callback-ove autorizacije za kanale prisutnosti, nećete vratiti `true` ako je korisnik ovlašten pridružiti se kanalu. Umjesto toga, trebali biste vratiti matricu podataka o korisniku.

Podaci koje vraća autoriziran callback bit će dostupni slušateljima događaja kanala prisutnosti u vašoj JavaScript aplikaciji. Ako korisnik nije ovlašten pridružiti se kanalu prisutnosti, vratit će se `false` ili `null`:

```
use App\Models\User;

Broadcast::channel('chat.{roomId}', function (User $user, int $roomId) {
    if ($user->canJoinRoom($roomId)) {
        return ['id' => $user->id, 'name' => $user->name];
    }
});
```

#### *Pridruživanje kanalima prisutnosti*

Da biste se pridružili kanalu prisutnosti, možete koristiti Echo-ovu `join` metodu. `join` metoda će vratiti `PresenceChannel` implementaciju koja vam, uz izlaganje `listen` metode, omogućuje pretplatu na `here`, `joining` i `leaving` događaje.

```
Echo.join(`chat.${roomId}`)
    .here((users) => {
        // ...
    })
    .joining((user) => {
        console.log(user.name);
    })
    .leaving((user) => {
        console.log(user.name);
    })
    .error((error) => {
        console.error(error);
    });
```

`here` callback će se izvršiti odmah nakon uspješnog pridruživanja kanalu i primit će matricu koji sadrži korisničke podatke za sve ostale korisnike koji su trenutno pretplaćeni na kanal. `joining` metoda će se izvršiti kada se novi korisnik pridruži kanalu, dok će se metoda `leaving` izvršiti kada korisnik napusti kanal. `error` metoda će se izvršiti kada krajnja točka provjere autentičnosti vrati HTTP statusni kod različit od 200 ili ako postoji problem prilikom analize vraćenog JSON-a.

#### *Emitiranje na kanalima prisutnosti*

Kanali prisutnosti mogu primiti događaje baš kao javni ili privatni kanali. Koristeći primjer chat sobe, možda ćemo htjeti emitirati `NewMessage` događaje na sobu kanala prisutnosti. Da bismo to učinili, vratit ćemo instancu `PresenceChannel` iz događaja `broadcastOn` metode:

```
/**
 * Dohvati kanale na kojima bi događaj trebao biti emitiran.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
```

```
public function broadcastOn(): array
{
    return [
        new PresenceChannel('chat.'.$this->message->room_id),
    ];
}
```

Kao i kod drugih događaja, možete koristiti pomoćnik `broadcast` i `toOthers` metodu za isključivanje tekućeg korisnika iz primanja emitiranja:

```
broadcast(new NewMessage($message));

broadcast(new NewMessage($message))->toOthers();
```

Kao što je tipično za druge tipove događaja, možete slušati događaje poslane kanalima prisutnosti pomoću Echo-ove `listen` metode:

```
Echo.join('chat.${roomId}')
    .here(/* ... */)
    .joining(/* ... */)
    .leaving(/* ... */)
    .listen('NewMessage', (e) => {
        // ...
    });
```

### Model emitiranja (Broadcasting)

#### UPOZORENJE:



Prije nego što pročitate sljedeću dokumentaciju o emitiranju modela, preporučujemo da se upoznate s općim konceptima Laravelovih usluga emitiranja modela kao i kako ručno stvoriti i slušati događaje emitiranja.

Uobičajeno je emitirati događaje kada se kreiraju, ažuriraju ili brišu [Eloquent modeli](#) vaše aplikacije. Naravno, to se lako može postići ručnim [definiranjem prilagođenih događaja za promjene stanja modela Eloquent](#) i označavanjem tih događaja ShouldBroadcast interface.

Međutim, ako ove događaje ne koristite ni u koje druge svrhe u svojoj aplikaciji, može biti nezgodno stvarati klase događaja samo u svrhu njihovog emitiranja. Da biste to ispravili, Laravel vam omogućuje da naznačite da bi model Eloquent trebao automatski emitirati svoje promjene stanja.

Za početak, vaš Eloquent model trebao bi koristiti `Illuminate\Database\Eloquent\BroadcastsEvents` svojstvo. Osim toga, model bi trebao definirati `broadcastOn` metodu koja će vratiti niz kanala na kojima bi se događaji modela trebali emitirati:

```
<?php

namespace App\Models;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Database\Eloquent\BroadcastsEvents;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Post extends Model
{
    use BroadcastsEvents, HasFactory;

    /**
     * Dohvati korisnika kojem post pripada.
     */
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }

    /**
     * Dohvati kanale na kojima bi se događaji modela trebali emitirati.
     *
     * @return array<int,
    \Illuminate\Broadcasting\Channel|\Illuminate\Database\Eloquent\Model>
     */
    public function broadcastOn(string $event): array
    {
        return [$this, $this->user];
    }
}
```

Nakon što vaš model uključi ovu značajku i definira svoje kanale emitiranja, početak će automatski emitirati događaje kada se instanca modela stvori, ažurira, izbriše, baci u smeće ili vrati.

Osim toga, možda ste primijetili da `broadcastOn` metoda prima `$event` string argument. Ovaj argument sadrži vrstu događaja koji se dogodio na modelu i imat će vrijednost `created`, `updated`, `deleted`, `trashed` ili `restored`. Provjerom vrijednosti ove varijable možete odrediti na koje kanale (ako ih ima) model treba emitirati za određeni događaj:

```
/**
 * Dohvati kanale na kojima bi se događaji modela trebali emitirati.
 *
 * @return array<string, array<int,
\Illuminate\Broadcasting\Channel|\Illuminate\Database\Eloquent\Model>>
 */
public function broadcastOn(string $event): array
{
    return match ($event) {
        'deleted' => [],
        default => [$this, $this->user],
    };
}
```

### *Prilagođavanje modela emitiranja događaja kreiranja (engl. Customizing Model Broadcasting Event Creation)*

Povremeno biste mogli prilagoditi način na koji Laravel stvara događaj emitiranja temeljnog modela. To možete postići definiranjem `newBroadcastableEvent` metode na vašem Eloquent modelu. Ova metoda bi trebala vratiti `Illuminate\Database\Eloquent\BroadcastableModelEventOccurred` instancu:

```
use Illuminate\Database\Eloquent\BroadcastableModelEventOccurred;

/**
 * Kreiranje novog događaja modela emitiranja za model.
 */
protected function newBroadcastableEvent(string $event):
BroadcastableModelEventOccurred
{
    return (new BroadcastableModelEventOccurred(
        $this, $event
    ))->dontBroadcastToCurrentUser();
}
```

### *Model konvencija emitiranja*

#### *Konvencija kanala*

Kao što ste možda primijetili, `broadcastOn` metoda u gornjem primjeru modela nije vratila `Channel` instancu. Umjesto toga, Eloquent modeli su vraćeni izravno. Ako je instanca modela Eloquent vraćena metodom vašeg modela `broadcastOn` (ili je sadržana u polju koje je vratila metoda), Laravel će automatski instancirati instancu privatnog kanala za model koristeći naziv klase modela i identifikator primarnog ključa kao naziv kanala.



Dakle, `App\Models\User` model s `id` bi `1` se pretvorio u `Illuminate\Broadcasting\PrivateChannel` instancu s imenom od `App\Models\User.1`. Naravno, uz vraćanje instanci modela Eloquent iz `broadcastOn` metode vašeg modela, možete vratiti potpune `Channel` instance kako biste imali potpunu kontrolu nad nazivima kanala modela:

```
use Illuminate\Broadcasting\PrivateChannel;

/**
 * Dohvati kanale na kojima bi se događaji modela trebali emitirati.
 *
 * @return array<int, \Illuminate\Broadcasting\Channel>
 */
public function broadcastOn(string $event): array
{
    return [
        new PrivateChannel('user.' . $this->id)
    ];
}
```

Ako planirate eksplicitno vratiti instancu kanala iz `broadcastOn` metode vašeg modela, možete prenijeti instancu modela Eloquent konstruktoru kanala. Pritom će Laravel upotrijebiti konvencije modela kanala o kojima se govorilo gore za pretvaranje modela Eloquent u niz naziva kanala:

```
return [new Channel($this->user)];
```

Ako trebate odrediti naziv kanala modela, možete pozvati metodu `broadcastChannel` na bilo kojoj instanci modela. Na primjer, ova metoda vraća niz `App\Models\User.1` za `App\Models\User` model s `id 1`:

```
$user->broadcastChannel()
```

### Konvencija događaja

Budući da događaji emitiranja modela nisu povezani sa "stvarnim" događajem unutar `App\Events` direktorija vaše aplikacije, dodjeljuju im se naziv i nosivost na temelju konvencija. Laravelova je konvencija emitirati događaj koristeći naziv klase modela (ne uključujući prostor imena) i naziv događaja modela koji je pokrenuo emitiranje.

Tako bi, na primjer, ažuriranje modela `App\Models\Post` emitiralo događaj vašoj aplikaciji na strani klijenta kao `PostUpdated` sa sljedećim sadržajem:

```
{
  "model": {
    "id": 1,
    "title": "Moj prvi post "
    ...
  }
}
```

```
    },  
    ...  
    "socket": "someSocketId",  
}
```

Brisanje modela `App\Models\User` bi emitiralo događaj pod nazivom `UserDeleted`.

Ako želite, možete definirati prilagođeni naziv emitiranja i nosivost dodavanjem metode `broadcastAs` i `broadcastWith` svom modelu. Ove metode primaju naziv događaja/operacije modela koji se odvija, što vam omogućuje da prilagodite naziv događaja i nosivost za svaku operaciju modela. Ako `null` se vrati iz `broadcastAs` metode, Laravel će koristiti konvencije naziva događaja modela emitiranja o kojima se govorilo gore prilikom emitiranja događaja:

```
/**  
 * Naziv emitiranja modela događaja.  
 */  
public function broadcastAs(string $event): string|null  
{  
    return match ($event) {  
        'created' => 'post.created',  
        default => null,  
    };  
}  
  
/**  
 * Dohvatite podatke za emitiranje za model.  
 *  
 * @return array<string, mixed>  
 */  
public function broadcastWith(string $event): array  
{  
    return match ($event) {  
        'created' => ['title' => $this->title],  
        default => ['model' => $this],  
    };  
}
```

### *Slušanje modela emitiranja*

Nakon što ste dodali `BroadcastsEvents` svojstvo svom modelu i definirali `broadcastOn` metodu svog modela, spremni ste za početak osluškivanja emitiranih događaja modela unutar vaše aplikacije na strani klijenta. Prije nego što počnete, možda ćete htjeti konzultirati cjelovitu dokumentaciju o slušanju događaja.

Prvo upotrijebite `private` metodu za dohvaćanje instance kanala, zatim pozovite `listen` metodu za slušanje određenog događaja. Tipično, naziv kanala koji se daje `private` metodi trebao bi odgovarati konvencijama Laravelovog modela emitiranja.

Nakon što dobijete instancu kanala, možete koristiti `listen` metodu za slušanje određenog događaja. Budući da događaji emitiranja modela nisu povezani sa "stvarnim" događajem unutar `App\Events` direktorija vaše aplikacije, ime događaja mora imati prefiks da `.bi` se naznačilo da ne pripada određenom prostoru imena. Svaki događaj emitiranja modela ima `model` svojstvo koje sadrži sva svojstva modela koja se mogu emitirati:

```
Echo.private(`App.Models.User.${this.user.id}`)
    .listen('.PostUpdated', (e) => {
        console.log(e.model);
    });
```

### Klijentski događaji

#### NAPOMENA:



Kada koristite [Pusher kanale](#), morate omogućiti opciju "Klijentski događaji" u odlomku "Postavke aplikacije" [na nadzornoj ploči vaše aplikacije](#) kako biste slali klijentske događaje.

Ponekad ćete možda poželjeti emitirati događaj drugim povezanim klijentima, a da uopće ne dodirnete svoju Laravel aplikaciju. To može biti posebno korisno za stvari poput obavijesti o "tipkanju", gdje želite upozoriti korisnike svoje aplikacije da drugi korisnik upisuje poruku na određenom zaslonu.

Za emitiranje događaja klijenta, možete koristiti Echo-ovu `whisper` metodu:

```
Echo.private(`chat.${roomId}`)
    .whisper('typing', {
        name: this.user.name
    });
```

Za slušanje događaja klijenta, možete koristiti metodu `listenForWhisper`:

```
Echo.private(`chat.${roomId}`)
    .listenForWhisper('typing', (e) => {
        console.log(e.name);
    });
```

### Obavijesti

Spajanjem emitiranja događaja s obavijestima, vaša JavaScript aplikacija može primati nove obavijesti čim se pojave bez potrebe za osvježavanjem stranice. Prije nego počnete, svakako pročitajte dokumentaciju o korištenju kanala obavijesti o emitiranju.

Nakon što ste konfigurirali obavijest za korištenje emitiranog kanala, možete slušati emitirane događaje pomoću Echo-ove `notification` metode. Zapamtite, naziv kanala treba odgovarati nazivu klase entiteta koji prima obavijesti:

```
Echo.private(`App.Models.User.${userId}`)  
  .notification((notification) => {  
    console.log(notification.type);  
  });
```

U ovom primjeru, sve obavijesti poslane `App\Models\User` instancama putem `broadcast` kanala primile bi povratni poziv. Povratni poziv autorizacije kanala za `App.Models.User.{id}` kanal uključen je u datoteku vaše aplikacije `routes/channels.php`.

## Keš

### Uvod

Neki od zadataka dohvaćanja ili obrade podataka koje izvodi vaša aplikacija mogu biti intenzivni za procesor ili zahtijevati nekoliko sekundi. Kada je to slučaj, uobičajeno je keširati dohvaćene podatke neko vrijeme kako bi se mogli brzo dohvatiti na sljedećim zahtjevima za istim podacima. Keširani podaci obično se pohranjuju u vrlo brzu pohranu podataka kao što je Memcached ili Redis .

Srećom, Laravel pruža ekspresivan, objedinjeni API za razne pozadine keša, omogućujući vam da iskoristite njihovu munjevito brzu pretragu podataka i ubrzate svoju web aplikaciju.

### Konfiguracija

Konfiguracijska datoteka keša vaše aplikacije nalazi se na `config/cache.php`. U ovoj datoteci možete navesti koju pohranu keša želite koristiti prema zadanim postavkama u vašoj aplikaciji. Laravel podržava popularne pozadine za keširanje kao što su Memcached , Redis , DynamoDB i relacijske baze podataka izvan kutije. Dodatno, dostupan je upravljački program za keš temeljen na datoteci, arraya "null" upravljački programi za keš pružaju prikladne pozadine keša za vaše automatizirane testove.

Konfiguracijska datoteka keša također sadrži niz drugih opcija koje možete pregledati. Prema zadanim postavkama, Laravel je konfiguriran da koristi `database` upravljački program za keš,, koji pohranjuje serijalizirane, predmemorirane objekte u bazi podataka vaše aplikacije.

### *Preduvjeti za driver*

#### *Baza podataka*

Kada koristite `database` upravljački program za keš, trebat će vam tablica baze podataka koja će sadržavati podatke iz predmemorije. Obično je to uključeno u Laravelovu zadanu `0001_01_01_000001_create_cache_table.php` migraciju baze podataka ; međutim, ako vaša aplikacija ne sadrži ovu migraciju, možete upotrijebiti `make:cache-table` naredbu Artisan da je napravite:

```
php artisan make:cache-table
```

```
php artisan migrate
```

### *Memcached*

Za korištenje Memcached upravljačkog programa potrebno je instalirati Memcached PECL paket . Možete navesti sve svoje Memcached poslužitelje u `config/cache.php` konfiguracijskoj datoteci. Ova datoteka već sadrži `memcached.servers` unos za početak:

```
'memcached' => [  
    // ...  
  
    'servers' => [  
        [  
            'host' => env('MEMCACHED_HOST', '127.0.0.1'),  
            'port' => env('MEMCACHED_PORT', 11211),
```

```
        'weight' => 100,  
    ],  
],  
],
```

Ako je potrebno, možete postaviti `host` opciju na stazu UNIX utičnice. Ako to učinite, `port` opcija bi trebala biti postavljena na 0:

```
'memcached' => [  
    // ...  
  
    'servers' => [  
        [  
            'host' => '/var/run/memcached/memcached.sock',  
            'port' => 0,  
            'weight' => 100  
        ],  
    ],  
],
```

### Redis

Prije korištenja Redis keša s Laravelom, morat ćete ili instalirati PhpRedis PHP ekstenziju putem PECL-a ili instalirati `predis/predis` paket (~2.0) putem Composer-a. Laravel Sail već uključuje ovo proširenje. Osim toga, službene Laravel platforme za implementaciju kao što su Laravel Forge i Laravel Vapor imaju proširenje PhpRedis instalirano prema zadanim postavkama.

Za više informacija o konfiguraciji Redisa, pogledajte stranicu dokumentacije za Laravel .

### DynamoDB

Prije korištenja DynamoDB dražvera za keš, morate stvoriti DynamoDB tablicu za pohranjivanje svih podataka u keš. Tipično, ova bi se tablica trebala zvati cache. Međutim, trebali biste dati naziv tablici na temelju vrijednosti `stores.dynamodb.table` konfiguracijske vrijednosti unutar cachekonfiguracijske datoteke. Naziv tablice također se može postaviti preko `DYNAMODB_CACHE_TABLE` varijable okoline.

Ova tablica također treba imati particijski ključ niza s nazivom koji odgovara vrijednosti konfiguracijske stavke unutar konfiguracijske datoteke `stores.dynamodb.attributes.key` vaše aplikacije . `cache` Prema zadanim postavkama, particijski ključ trebao bi imati naziv `key`.

Zatim instalirajte AWS SDK kako bi vaša Laravel aplikacija mogla komunicirati s DynamoDB-om:

```
composer require aws/aws-sdk-php
```

Osim toga, trebali biste osigurati da su navedene vrijednosti za opcije konfiguracije pohrane DynamoDB keš. Obično ove opcije, kao što su `AWS_ACCESS_KEY_ID` i `AWS_SECRET_ACCESS_KEY`, trebaju biti definirane u `.env` konfiguracijskoj datoteci vaše aplikacije:

```
'dynamodb' => [  
    'driver' => 'dynamodb',  
    'key' => env('AWS_ACCESS_KEY_ID'),  
    'secret' => env('AWS_SECRET_ACCESS_KEY'),  
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),  
    'table' => env('DYNAMODB_CACHE_TABLE', 'cache'),  
    'endpoint' => env('DYNAMODB_ENDPOINT'),  
],
```

## Upotreba keša

### *Dobivanje keš instance*

Da biste dobili instancu pohrane predmemorije, možete koristiti fasadu Cache, što je ono što ćemo koristiti u ovoj dokumentaciji. `Cache` fasada pruža praktičan, sažet pristup temeljnim implementacijama ugovora Laravel predmemorije:

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Support\Facades\Cache;  
  
class UserController extends Controller  
{  
    /**  
     * Prikažite popis svih korisnika aplikacije.  
     */  
    public function index(): array  
    {  
        $value = Cache::get('key');  
  
        return [  
            // ...  
        ];  
    }  
}
```

### *Pristup višestrukim keš memorijama*

Koristeći Cache fasadu, možete pristupiti raznim memorijama keša pomoću `store` metode. Ključ proslijeđen metodi `store` trebao bi odgovarati jednoj od trgovina navedenih u `stores` konfiguracijskoj matrici u vašoj `cache` konfiguracijskoj datoteci:

```
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minuta
```

### *Dohvaćanje stavki iz keša*

`Cache` fasade `get` metoda koristi se za dohvaćanje stavki iz keša. Ako stavka ne postoji u kešu, `null` će biti vraćen. Ako želite, možete proslijediti drugi argument `get` metodi specificirajući zadanu vrijednost koju želite vratiti ako stavka ne postoji:

```
$value = Cache::get('key');

$value = Cache::get('key', 'default');
```

Možete čak proslijediti closure kao zadanu vrijednost. Rezultat closure bit će vraćen ako navedena stavka ne postoji u kešu. Proslijeđivanje closure omogućava vam odgodu dohvaćanja zadanih vrijednosti iz baze podataka ili druge vanjske usluge:

```
$value = Cache::get('key', function () {
    return DB::table('/...')->get();
});
```

### *Utvrđivanje postojanja predmeta*

Metoda `has` se može koristiti za određivanje postoji li stavka u kešu. Ova metoda će također vratiti `false` ako stavka postoji, ali je njena vrijednost `null`:

```
if (Cache::has('key')) {
    // ...
}
```

### *Povećanje/smanjenje vrijednosti*

`increment` i `decrement` metode mogu se koristiti za prilagodbu vrijednosti cjelobrojnih stavki u kešu. Obje ove metode prihvaćaju opcionalni drugi argument koji označava količinu za koji treba povećati ili smanjiti vrijednost stavke:

```
// Inicijaliziraj vrijednost ako ne postoji...
Cache::add('key', 0, now()->addHours(4));

// Increment or decrement the value...
Cache::increment('key');
```



```
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

### *Dohvaćanje i pohranjivanje*

Ponekad ćete možda poželjeti dohvatiti stavku iz keša, ali i pohraniti zadanu vrijednost ako tražena stavka ne postoji. Na primjer, možda želite dohvatiti sve korisnike iz predmemorije ili, ako ne postoje, dohvatiti ih iz baze podataka i dodati ih u keš. To možete učiniti pomoću `Cache::remember` metode:

```
$value = Cache::remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

Ako stavka ne postoji u kešu, zatvaranje proslijeđeno metodi `remember` će se izvršiti i njegov rezultat će biti smješten u keš.

Možete upotrijebiti ovu `rememberForever` metodu da dohvatite stavku iz keša ili je zauvijek pohranite ako ne postoji:

```
$value = Cache::rememberForever('users', function () {
    return DB::table('users')->get();
});
```

### *Dohvati i izbriši*

Ako trebate dohvatiti stavku iz keša i zatim izbrisati stavku, možete koristiti ovu `pull` metodu. Kao i `get` metoda, `null` će vratiti se ako stavka ne postoji u kešu:

```
$value = Cache::pull('key');

$value = Cache::pull('key', 'default');
```

### *Pohranjivanje stavki u keš*

Možete koristiti `put` metodu na `Cache` fasadi za spremanje stavki u keš:

```
Cache::put('key', 'value', $seconds = 10);
```

Ako se vrijeme pohrane ne proslijedi `put` metodi, stavka će biti pohranjena na neodređeno vrijeme:

```
Cache::put('key', 'value');
```

Umjesto proslijeđivanja broja sekundi kao cijelog broja, možete proslijediti i `DateTime` instancu koja predstavlja željeno vrijeme isteka keširane stavke:

```
Cache::put('key', 'value', now()->addMinutes(10));
```

#### *Spremi ako nije prisutan*

`add` metoda će samo dodati stavku u keš ako već ne postoji u memoriji keša. Metoda će se vratiti `true` ako je stavka stvarno dodana u keš. U suprotnom, metoda će vratiti `false`. Metoda `add` je atomska operacija:

```
Cache::add('key', 'value', $seconds);
```

#### *Pohranjivanje predmeta zauvijek*

`forever` metoda se može koristiti za trajno pohranjivanje stavke u keš. Budući da ove stavke neće isteći, moraju se ručno ukloniti iz keša pomoću `forget` metode:

```
Cache::forever('key', 'value');
```

#### **NAPOMENA:**



Ako koristite upravljački program Memcached, stavke koje su pohranjene "zauvijek" mogu se ukloniti kada keš dosegne ograničenje veličine.

#### *Uklanjanje stavki iz keša*

Stavke iz keša možete ukloniti koristeći `forget` metodu:

```
Cache::forget('key');
```

Također možete ukloniti stavke tako da navedete nulu ili negativan broj sekundi isteka:

```
Cache::put('key', 'value', 0);
```

```
Cache::put('key', 'value', -5);
```

Cijeli keš možete izbrisati na sljedeći `flush` metodu:

```
Cache::flush();
```

#### **UPOZORENJE:**



Čišćenje keša ne poštuje vaš konfigurirani keš "prefiks" i uklonit će sve unose iz keša. Ovo pažljivo razmotrite kada čistite keš koju dijele druge aplikacije.

### Keš pomoćnik

Osim korištenja `Cache` fasade, također možete koristiti globalnu `cache` funkciju za dohvaćanje i pohranjivanje podataka pomoću keša. Kada `cache` funkcija se pozove s jednim string argumentom, vratit će vrijednost zadanog ključa:

```
$value = cache('key');
```

Ako funkciji date matricu parova ključ/vrijednost i vrijeme isteka, ona će pohraniti vrijednosti u keš za određeno vrijeme:

```
cache(['key' => 'value'], $seconds);

cache(['key' => 'value'], now()->addMinutes(10));
```

Kada se pozove `cache` funkcija bez ikakvih argumenata, vraća instancu implementacije `Illuminate\Contracts\Cache\Factory`, što vam omogućava pozivanje drugih keš metoda:

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

**NAPOMENA:** Kada testirate poziv globalne `cache` funkcije, možete koristiti `Cache::shouldReceive` metodu baš kao da testirate fasadu .



### Atomska zaključavanja

Da biste koristili ovo svojstvo, vaša aplikacija mora koristiti `memcached`, `redis`, `dynamodb`, `database`, `file` ili `array` keš driver kao zadani upravljački program vaše aplikacije. Osim toga, svi serveri moraju komunicirati s istim centralnim keš serverom.

#### Upravljanje zaključavanjima

Atomska zaključavanja dopuštaju manipulaciju distribuiranih brava bez brige o utrci za resursima (engl. race conditions)<sup>54</sup>. Na primjer, Laravel Forge koristi atomsko zaključavanje kako bi osigurao da se na serveru istovremeno izvršava samo jedan udaljeni zadatak. Možete izraditi i upravljati bravama pomoću `Cache::lock` metode:

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);
```

<sup>54</sup> Race conditions je pojam koji opisuje situaciju u kojoj dva ili više procesa pokušavaju simultano pristupiti ili izmijeniti isti resurs, što može dovesti do neočekivanog ponašanja i pogrešnih rezultata, jer izvršenje tih operacija zavisi od redoslijeda pristupa resursu. Problem nastaje kada redoslijed izvršavanja tih operacija nije pravilno sinkroniziran.

```
if ($lock->get()) {  
    // Zaključavanje postignuto na 10 sekundi...  
  
    $lock->release();  
}
```

`get` metoda također prihvaća closure. Nakon što se closure izvrši, Laravel će automatski osloboditi zaključavanje:

```
Cache::lock('foo', 10)->get(function () {  
    // Zaključavanje postignuto na 10 sekundi i automatski oslobađa...  
});
```

Ako zaključavanje nije dostupno u trenutku kada ga zatražite, možete uputiti Laravel da pričeka određeni broj sekundi. Ako se zaključavanje ne može dobiti unutar navedenog vremenskog ograničenja, `Illuminate\Contracts\Cache\LockTimeoutException` bit će bačeno:

```
use Illuminate\Contracts\Cache\LockTimeoutException;  
  
$lock = Cache::lock('foo', 10);  
  
try {  
    $lock->block(5);  
  
    // Zaključavanje je postignuto nakon čekanja od najviše 5 sekundi...  
} catch (LockTimeoutException $e) {  
    // Nije moguće zaključati...  
} finally {  
    $lock->release();  
}
```

Primjer iznad može se pojednostaviti prosljeđivanjem closure `block` metode. Kada se closure proslijedi ovoj metodi, Laravel će pokušati preuzeti zaključavanje na navedeni broj sekundi i automatski će otpustiti zaključavanje nakon što se closure izvrši:

```
Cache::lock('foo', 10)->block(5, function () {  
    // Zaključavanje je postignuto nakon čekanja od najviše 5 sekundi...  
});
```

*Upravljanje zaključavanjem procesa*

Ponekad ćete možda poželjeti usvojiti zaključavanje u jednom procesu i otpustiti ga u drugom procesu. Na primjer, možete usvojiti zaključavanje tokom web zahtjeva i željeti osloboditi zaključavanje na kraju poslova u redu čekanja (engl. *queued job*) koji je pokrenut tim zahtjevom. U ovom scenariju, trebali biste proslijediti "vlasnički token" s dosegom zaključavanja poslu u redu čekanja tako da posao može re-instancirati zaključavanje koristeći dani token.

U donjem primjeru otpremit ćemo posao u redu čekanja ako je zaključavanje uspješno usvojeno. Osim toga, proslijedit ćemo vlasnički token brave poslu u redu čekanja pomoću `owner` metode zaključavanja:

```
$podcast = Podcast::find($id);

$lock = Cache::lock('processing', 120);

if ($lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

`$podcast = Podcast::find($id);` linija koda pronalazi podkast iz baze podataka na osnovu njegovog ID-a. Koristi Eloquent model `Podcast` kako bi dohvatila specifični unos s tim ID-em. `$id` predstavlja identifikator podkasta koji se obrađuje.

`$lock = Cache::lock('processing', 120);` kreira atomsko zaključavanje sa ključem `'processing'`, koji traje 120 sekundi (2 minute). `Cache::lock()` omogućava Laravelu da kreira jedinstveni ključ za zaključavanje (`'processing'`), čime osigurava da samo jedan proces može dobiti ovaj ključ u jednom trenutku. Ako neki drugi proces pokuša dobiti isti ključ dok je zaključan, on će biti odbijen dok ključ ne istekne ili se oslobodi.

Metoda `get()` pokušava da ostvari zaključavanje. Ako uspješno ostvari zaključavanje, ulazi u `if` blok i nastavlja izvršavanje. Ako zaključavanje nije dostupno (ako ga je neki drugi proces već preuzeo), kod unutar `if` bloka se ne izvršava. Ako je zaključavanje uspješno ostvareno, pokreće se posao `ProcessPodcast`.

Linija `ProcessPodcast::dispatch($podcast, $lock->owner());` koristi Laravel Queues (redove) da "dispećuje" (odašilje) zadatak za procesiranje podkasta u pozadinskom procesu. To znači da Laravel dodaje ovaj zadatak u red za izvršavanje i izvrši ga u pozadini, van glavnog toka aplikacije. `$lock->owner()` vraća jedinstveni identifikator (vlasnika) trenutnog zaključavanja. Ovo može biti korisno za praćenje tko je ostvario zaključavanje ili za otklanjanje grešaka.

Ovaj kod omogućava sigurno procesiranje podkasta na sledeći način: Proces pokušava dobiti zaključavanje s ključem `'processing'`. Ako neki drugi proces već procesira isti podkast, zaključavanje neće biti dostupno, i samim tim ovaj proces neće moći započeti. Ako proces uspješno ostvari zaključavanje, zadatak za procesiranje podkasta (`ProcessPodcast`) se dispećuje, a zaključavanje traje do 120 sekundi (ili dok proces ne završi). Na taj način se sprečava da više procesa istovremeno pokrene procesiranje istog podkasta, čime se izbegavaju potencijalne greške usljed paralelnih operacija (npr. dupliranje zapisa u bazi, preklapanje podataka, itd.).

Unutar našeg aplikacijskog `ProcessPodcast` posla, možemo vratiti i osloboditi zaključavanje pomoću vlasničkog tokena:

```
Cache::restoreLock('processing', $this->owner)->release();
```

Ako želite osloboditi zaključavanje bez poštovanja prema trenutnom vlasniku, možete upotrijebiti metodu `forceRelease`:

```
Cache::lock('processing')->forceRelease();
```

Dodavanje prilagođenih driver-a za keš

#### *Pisanje driver-a*

Da bismo stvorili naš prilagođeni upravljački program za predmemoriju, prvo moramo implementirati `Illuminate\Contracts\Cache\Store` ugovor. Dakle, implementacija MongoDB keša mogla bi izgledati otprilike ovako:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

Samo trebamo implementirati svaku od ovih metoda pomoću MongoDB veze. Za primjer kako implementirati svaku od ovih metoda, pogledajte `Illuminate\Cache\MemcachedStore` [izvorni kod](#) [Laravel radne okoline](#). Nakon što je naša implementacija dovršena, možemo završiti prilagođenu registraciju upravljačkog programa pozivanjem metode `Cache` fasade `extend`:

```
Cache::extend('mongo', function (Application $app) {
    return Cache::repository(new MongoStore);
});
```

**NAPOMENA:**

Ako se pitate gdje staviti svoj prilagođeni kod upravljačkog programa za keš, možete stvoriti prostor `Extensions` imena unutar svog `app` direktorija. Međutim, imajte na umu da Laravel nema krutu strukturu aplikacije i slobodno možete organizirati svoju aplikaciju prema svojim željama.

*Registracija driver-a*

Za registraciju prilagođenog keš driver-a s Laravelom, koristit ćemo `extend` metodu na `Cache` fasadi. Budući da drugi pružatelji usluga mogu pokušati čitati keširane vrijednosti unutar svoje `boot` metode, registrirat ćemo naš prilagođeni driver unutar `booting` callback-a. Korištenjem `booting` callback-a možemo osigurati da je prilagođeni driver registriran neposredno prije nego što se `boot` metoda pozove na davateljima usluga naše aplikacije, ali nakon što registerse metoda pozove na svim davateljima usluga. Registrirat ćemo naš `booting` callback unutar `register` metode naše aplikacije `App\Providers\AppServiceProvider` klase:

```
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Contracts\Foundation\Application;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Registriraj bilo koju aplikacijsku uslugu.
     */
    public function register(): void
    {
        $this->app->booting(function () {
            Cache::extend('mongo', function (Application $app) {
                return Cache::repository(new MongoStore);
            });
        });
    }

    /**
     * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
     */
    public function boot(): void
    {
        // ...
    }
}
```

```
}  
}
```

Prvi argument proslijeđen `extend` metodi je naziv driver-a. To će odgovarati vašoj `driver` opciji u `config/cache.php` konfiguracijskoj datoteci. Drugi argument je closure koje bi trebalo vratiti `Illuminate\Cache\Repository` instancu. Closure će biti proslijeđen `$app` instanci, koja je instanca [kontejnera usluga](#).

Nakon što je vaše proširenje registrirano, ažurirajte `CACHE_STORE` varijablu okoline ili `default` opciju u konfiguracijskoj datoteci vaše aplikacije `config/cache.php` na naziv vašeg proširenja.

### Događaji

Da biste izvršili kod kod svake operacije keša, možete slušati različite događaje koje šalje keš:

Naziv događaja
<code>Illuminate\Cache\Events\CacheHit</code>
<code>Illuminate\Cache\Events\CacheMissed</code>
<code>Illuminate\Cache\Events\KeyForgotten</code>
<code>Illuminate\Cache\Events\KeyWritten</code>

Da biste povećali izvedbu, možete onemogućiti događaje predmemorije postavljanjem eventskonfiguracijske opcije na za određenu pohranu predmemorije u konfiguracijskoj datoteci falsevaše aplikacije :`config/cache.php`

```
'database' => [  
    'driver' => 'database',  
    // ...  
    'events' => false,  
],
```



## Eloquent

### Početak rada

#### Uvod

Laravel uključuje Eloquent, objektno-relacijski mapper (ORM) koji čini ugodnu interakciju s vašom bazom podataka. Kada koristite Eloquent, svaka tablica baze podataka ima odgovarajući „Model“ koji se koristi za interakciju s tom tablicom. Osim dohvaćanja zapisa iz tablice baze podataka, Eloquent modeli vam također omogućuju umetanje, ažuriranje i brisanje zapisa iz tablice.

#### NAPOMENA:



Prije početka, svakako konfigurirajte vezu s bazom podataka u konfiguracijskoj datoteci vaše aplikacije `config/database.php`. Za više informacija o konfiguraciji vaše baze podataka, pogledajte dokumentaciju o konfiguraciji baze podataka.

#### Laravel Bootcamp

Ako ste novi u Laravelu, slobodno uskočite u Laravel Bootcamp. Laravel Bootcamp će vas provesti kroz izradu vaše prve Laravel aplikacije koristeći Eloquent. To je sjajan način za obilazak svega što Laravel i Eloquent imaju za ponuditi.

#### Generiranje klasa modela

Za početak, kreirajmo model Eloquent. Modeli obično žive u `app\Models` direktoriju i proširuju `Illuminate\Database\Eloquent\Model` klasu. Možete koristiti `make:model` naredbu Artisan za generiranje novog modela:

```
php artisan make:model Flight
```

Ako želite generirati migraciju baze podataka kada generirate model, možete upotrijebiti opciju `migration` ili `-m`:

```
php artisan make:model Flight -migration
```

Prilikom generiranja modela možete generirati razne druge vrste klasa, kao što su tvornice (engl. factories), seeder-i, politike (engl. policies), kontroleri i zahtjevi za formama. Osim toga, ove se opcije mogu kombinirati za stvaranje više klasa odjednom:

```
# Generira model i FlightFactory klasu...
```

```
php artisan make:model Flight --factory
```

```
php artisan make:model Flight -f
```

```
# Generira model i FlightSeeder klasu...
```

```
php artisan make:model Flight --seed
```

```
php artisan make:model Flight -s
```

```
# Generira model i FlightController klasu...
```

```
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generira model, FlightController resurs klasu i formu request klasa...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Generira model i FlightPolicy klasu...
php artisan make:model Flight --policy

# Generira model i migraciju, factory, seeder i kontroler...
php artisan make:model Flight -mfsc

# Kratica za generir. modela, migracije, factory, seeder, policy-ja, kontrolera
forme request-ova...
php artisan make:model Flight --all
php artisan make:model Flight -a

# Generira pivot model...
php artisan make:model Member --pivot
php artisan make:model Member -p
```

### *Provjera modela*

Ponekad može biti teško odrediti sve dostupne attribute i odnose modela samo letimičnim pregledom njegovog koda. Umjesto toga, isprobajte `model:show` naredbu Artisan, koja pruža praktičan pregled svih atributa i odnosa modela:

```
php artisan model:show Flight
```

### *Konvencija Eloquent modela*

Modeli generirani naredbom `make:model` bit će smješteni u `app/Models` direktorij. Ispitajmo osnovnu klasu modela i raspravimo neke od ključnih konvencija Eloquent-a:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    // ...
}
```

### Nazivi tablica

Nakon što ste pogledali gornji primjer, možda ste primijetili da Eloquentu nismo rekli koja tablica baze podataka odgovara našem `Flight` modelu. Prema konvenciji, "snake case", ime klase u množini koristit će se kao naziv tablice osim ako nije izričito naveden drugi naziv. Dakle, u ovom slučaju, Eloquent će pretpostaviti da `Flight` model pohranjuje zapise u `flights` tablicu, dok `AirTrafficController` model bi trebao pohraniti zapise u `air_traffic_controllers` tablicu.

Ako odgovarajuća tablica baze podataka vašeg modela ne odgovara ovoj konvenciji, možete ručno navesti naziv tablice modela definiranjem `table` svojstva na modelu:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Tablica povezana s modelom.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

### Primarni ključevi

Eloquent će također pretpostaviti da odgovarajuća tablica baze podataka svakog modela ima stupac primarnog ključa pod nazivom `id`. Ako je potrebno, možete definirati zaštićeno `$primaryKey` svojstvo na svom modelu kako biste naveli drugi stupac koji služi kao primarni ključ vašeg modela:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

```
}
```

Osim toga, Eloquent pretpostavlja da je primarni ključ inkrementirajuća cjelobrojna vrijednost, što znači da će Eloquent automatski pretvoriti primarni ključ u cijeli broj. Ako želite koristiti neinkrementirajući ili nenumerički primarni ključ, morate definirati javno `$incrementing` svojstvo na vašem modelu koje je postavljeno na `false`:

```
<?php

class Flight extends Model
{
    /**
     * Označava povećava li se ID modela automatski.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

Ako primarni ključ vašeg modela nije cjelobrojni broj, trebali biste definirati zaštićeno `$keyType` svojstvo na svom modelu. Ovo svojstvo treba imati vrijednost string:

```
<?php

class Flight extends Model
{
    /**
     * Tip podataka je primarni ključ ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

#### *„Kompozitni“ primarni ključevi*

Eloquent zahtijeva da svaki model ima barem jedan jedinstveni identifikacijski "ID" koji može poslužiti kao primarni ključ. "Kompozitne" primarne ključeve ne podržavaju modeli Eloquent. Međutim, slobodno možete dodati jedinstvene indekse s više stupaca tablicama baze podataka uz jedinstveni identifikacijski primarni ključ tablice.

### UUID i ULID ključevi

Umjesto korištenja auto-inkrementirajućih cjelobrojnih brojeva kao primarnih ključeva vašeg Eloquent modela, možete odabrati korištenje UUID-ova. UUID-ovi su univerzalno jedinstveni alfanumerički identifikatori koji imaju 36 znakova.

Ako želite da model koristi UUID ključ umjesto autoinkrementirajućeg ključa cijelog broja, možete koristiti značajku `Illuminate\Database\Eloquent\Concerns\HasUuids` na modelu. Naravno, trebali biste osigurati da model ima UUID ekvivalentni stupac primarnog ključa :

```
use Illuminate\Database\Eloquent\Concerns\HasUuids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUuids;

    // ...
}

$article = Article::create(['title' => 'Putovanje po Evropi']);

$article->id; // "8f8e8478-9035-4d23-b9a7-62f4d2612ce5"
```

Prema zadanim postavkama, `HasUuids` svojstvo će generirati "naručene" UUID-ove za vaše modele. Ti su UUID-ovi učinkovitiji za pohranu indeksirane baze podataka jer se mogu sortirati leksikografski.

Možete nadjačati proces generiranja UUID-a za određeni model definiranjem `newUniqueId` metode na modelu. Osim toga, možete odrediti koji bi stupci trebali primiti UUID-ove definiranjem `uniqueIds` metode na modelu:

```
use Ramsey\Uuid\Uuid;

/**
 * Generira novi UUID za model.
 */
public function newUniqueId(): string
{
    return (string) Uuid::uuid4();
}

/**
 * Dobijte stupce koji bi trebali dobiti jedinstveni identifikator.
 *
 * @return array<int, string>
```

```
*/  
public function uniqueIds(): array  
{  
    return ['id', 'discount_code'];  
}
```

Ako želite, možete odabrati korištenje "ULID-ova" umjesto UUID-ova. ULID-ovi su slični UUID-ovima; međutim, dugi su samo 26 znakova. Poput uređenih UUID-ova, ULID-ove je moguće leksikografski sortirati za učinkovito indeksiranje baze podataka. Da biste koristili ULID-ove, trebali biste koristiti `Illuminate\Database\Eloquent\Concerns\HasUlid`s svojstvo na svom modelu. Također biste trebali osigurati da model ima stupac primarnog ključa ekvivalentan ULID-u :

```
use Illuminate\Database\Eloquent\Concerns\HasUlid;  
use Illuminate\Database\Eloquent\Model;  
  
class Article extends Model  
{  
    use HasUlid;  
  
    // ...  
}  
  
$article = Article::create(['title' => 'Putovanje u Aziju']);  
  
$article->id; // "01gd4d3tgrrfqeda94gdbtdk5c"
```

#### *Vremenske oznake (engl. Timestamps)*

Prema zadanim postavkama, Eloquent očekuje postojanje `created_at` i `updated_at` stupaca u odgovarajućoj tablici baze podataka vašeg modela. Eloquent će automatski postaviti vrijednosti ovih stupaca kada se modeli kreiraju ili ažuriraju. Ako ne želite da ovim stupcima automatski upravlja Eloquent, trebali biste definirati `$timestamp` svojstvo na svom modelu s vrijednošću `false`:

```
<?php  
namespace App\Models;  
use Illuminate\Database\Eloquent\Model;  
class Flight extends Model  
{  
    /**  
     * Ukazuje treba li model imati vremensku oznaku (engl. Timestamp).  
     *  
     * @var bool  
     */
```

```
public $timestamps = false;
}
```

Ako trebate prilagoditi format vremenskih oznaka svog modela, postavite `$dateFormat` svojstvo na svoj model. Ovo svojstvo određuje kako se atributi datuma pohranjuju u bazi podataka kao i njihov format kada je model serijaliziran u polje ili JSON:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

Ako trebate prilagoditi nazive stupaca koji se koriste za pohranjivanje vremenskih oznaka, možete definirati `CREATED_AT` i `UPDATED_AT` konstante na svom modelu:

```
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

Ako želite izvršiti operacije modela bez `updated_at` izmjene vremenske oznake modela, možete raditi na modelu unutar zatvaranja danog `withoutTimestamps` metodi:

```
Model::withoutTimestamps(fn () => $post->increment('reads'));
```

### *Veze s bazom podataka*

Prema zadanim postavkama, svi će modeli Eloquent koristiti zadanu vezu s bazom podataka koja je konfigurirana za vašu aplikaciju. Ako želite navesti drugu vezu koja bi se trebala koristiti prilikom interakcije s određenim modelom, trebali biste definirati `$connection` svojstvo na modelu:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Veza s bazom podataka koju bi trebao koristiti model.
     *
     * @var string
     */
    protected $connection = 'mysql';
}
```

#### *Zadane vrijednosti atributa*

Prema zadanim postavkama, novoinstancirana instanca modela neće sadržavati vrijednosti atributa. Ako želite definirati zadane vrijednosti za neke od atributa vašeg modela, možete definirati `$attributes` svojstvo na svom modelu. Vrijednosti atributa smještene u `$attributes` polje trebaju biti u svom sirovom formatu koji se može "pohraniti" kao da su upravo pročitane iz baze podataka:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Zadane vrijednosti modela za attribute.
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```



### Konfiguriranje Eloquent Strictness

Laravel nudi nekoliko metoda koje vam omogućuju da konfigurirate Eloquentovo ponašanje i "striktnost" u raznim situacijama.

Prvo, `preventLazyLoading` metoda prihvaća izborni `boolean` argument koji pokazuje treba li spriječiti odlagano učitavanje. Na primjer, možda biste željeli onemogućiti odlagano učitavanje samo u neproizvodnim okruženjima kako bi vaše proizvodno okruženje nastavilo funkcionirati normalno čak i ako je odnos odlaganog učitavanja slučajno prisutan u proizvodnom kodu. Obično se ova metoda treba pozvati u `boot` metodi vaše aplikacije `AppServiceProvider`:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Pokrenite (engl. Bootstrap) bilo koju aplikacijsku uslugu.
 */
public function boot(): void
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

Također, možete uputiti Laravel da izbací iznimku kada pokušava ispuniti atribut koji se ne može ispuniti pozivanjem metode `preventSilentlyDiscardingAttributes`. To može pomoći u sprječavanju neočekivanih grešaka tokom lokalnog razvoja kod pokušaja postavljanja atributa koji nije dodan u polje modela `fillable`:

```
Model::preventSilentlyDiscardingAttributes(! $this->app->isProduction());
```

### Dohvaćanje modela

Nakon što ste izradili model i njegovu pridruženu tablicu baze podataka, spremni ste za početak dohvaćanja podataka iz vaše baze podataka. Svaki Eloquentov model možete zamisliti kao snažan alat za izradu upita (engl. query builder) koji vam omogućuje tečno postavljanje upita (engl. query) tablici baze podataka povezanoj s modelom. Metoda modela `all` će dohvatiti sve zapise iz tablice pridružene baze podataka modela:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

### Izgradnja upita (engl. Building Queries)

Eloquent `all` metoda će vratiti sve rezultate u tablici modela. Međutim, budući da svaki Eloquent model služi kao alat za sastavljanje upita, možete dodati dodatna ograničenja upitima i zatim pozvati `get` metodu za dohvaćanje rezultata:

```
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->take(10)
    ->get();
```

**NAPOMENA:**

Budući da su Eloquent modeli graditelji upita, trebali biste pregledati sve metode koje nudi Laravelov alat za izradu upita . Možete koristiti bilo koju od ovih metoda kada pišete svoje Eloquent upite.

*Osvježavanje modela*

Ako već imate instancu Eloquent modela koji je dohvaćen iz baze podataka, možete "osvježiti" model pomoću metoda `fresh` i `refresh`. Metoda `fresh` će ponovno dohvatiti model iz baze podataka. Neće utjecati na postojeću instancu modela:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

Metoda `refresh` će ponovno hidrirati postojeći model korištenjem svježih podataka iz baze podataka. Osim toga, osvježiti će se i svi njegovi učitani odnosi:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

*Kolekcije*

Kao što smo vidjeli, Eloquent metode vole `all` i `get` dohvaćaju više zapisa iz baze podataka. Međutim, ove metode ne vraćaju običan PHP niz. Umjesto toga, `Illuminate\Database\Eloquent\Collection` vraća se instanca.

Klasa Eloquent `Collection` proširuje osnovnu klasu Laravla `Illuminate\Support\Collection`, koja pruža razne korisne metode za interakciju sa zbirkama podataka. Na primjer, `reject` metoda se može upotrijebiti za uklanjanje modela iz zbirke na temelju rezultata pozvanog zatvaranja:

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
```

```
});

$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
});
```

Uz metode koje pruža Laravelova osnovna klasa zbirke, klasa zbirke Eloquent pruža nekoliko dodatnih metoda koje su posebno namijenjene za interakciju s kolekcijama Eloquent modela.

Budući da sve Laravelove kolekcije implementiraju PHP-ove interface koji se mogu ponavljati, možete prelaziti preko kolekcija kao da su matrica:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

### *Ulančavanje rezultata*

Vašoj aplikaciji može ponestati memorije ako pokušate učitati desetke tisuća Eloquentovih zapisa putem `all` ili `get` metoda. Umjesto korištenja ovih metoda, `chunk` metoda se može koristiti za učinkovitiju obradu velikog broja modela.

Metoda `chunk` će dohvatiti podskup Eloquent modela, prosljeđujući ih zatvaraču za obradu. Budući da se u isto vrijeme dohvaća samo trenutni dio Eloquent modela, `chunk` metoda će omogućiti znatno smanjenu upotrebu memorije pri radu s velikim brojem modela:

```
use App\Models\Flight;
use Illuminate\Database\Eloquent\Collection;

Flight::chunk(200, function (Collection $flights) {
    foreach ($flights as $flight) {
        // ...
    }
});
```

Prvi argument prosljeđen `chunk` metodi je broj zapisa koje želite primiti po "djelu". Zatvaranje prosljeđeno kao drugi argument bit će pozvano za svaki komad koji je dohvaćen iz baze podataka. Izvršit će se upit baze podataka kako bi se dohvatilo svaki dio zapisa prosljeđen zatvaranju.

Ako filtrirate rezultate metode `chunk` na temelju stupca koji ćete također ažurirati dok ponavljate rezultate, trebali biste koristiti metodu `chunkById`. Korištenje `chunk` metode u ovim scenarijima moglo

bi dovesti do neočekivanih i nedosljednih rezultata. Interno, `chunkById` metoda će uvijek dohvatiti modele sa `id` stupcem većim od zadnjeg modela u prethodnom dijelu:

```
Flight::where('departed', true)
    ->chunkById(200, function (Collection $flights) {
        $flights->each->update(['departed' => false]);
    }, $column = 'id');
```

#### *Ulančavanje korištenjem Lazy kolekcija*

Metoda radi `lazy` slično `chunk` metodi u smislu da, iza scene, izvršava upit u dijelovima. Međutim, umjesto prosljeđivanja svake karike direktno u povratni poziv kakav jest, `lazy` metoda vraća izravnatu `LazyCollection` Eloquent modele, što vam omogućava interakciju s rezultatima kao jedan tok:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    // ...
}
```

Ako filtrirate rezultate metode `lazy` na temelju stupca koji ćete također ažurirati dok ponavljate rezultate, trebali biste koristiti metodu `lazyById`. Interno, `lazyById` metoda će uvijek dohvatiti modele sa `id` stupcem većim od zadnjeg modela u prethodnom dijelu:

```
Flight::where('departed', true)
    ->lazyById(200, $column = 'id')
    ->each->update(['departed' => false]);
```

Rezultate možete filtrirati prema silaznom redoslijedu `id` koristeći `lazyByIdDesc` metodu.

#### *Kursori*

Slično `lazy` metodi, `cursor` metoda se može koristiti za značajno smanjenje potrošnje memorije vaše aplikacije prilikom ponavljanja kroz desetke tisuća zapisa modela Eloquent.

Metoda `cursor` će izvršiti samo jedan upit baze podataka; međutim, pojedinačni Eloquent modeli neće biti hidratizirani sve dok se stvarno ne ponove. Zato se samo jedan Eloquentov model čuva u memoriji u bilo kojem trenutku tokom ponavljanja preko kursora.

#### **UPOZORENJE:**



Budući da `cursor` metoda drži samo jedan Eloquentov model u memoriji odjednom, ne može brzo (eager) učitavati odnose. Ako trebate eager load odnose, razmislite o korištenju lazy metode umjesto toga.

Interno, `cursor` metoda koristi PHP generatore za implementaciju ove funkcije:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    // ...
}
```

Vraća `cursor` instancu `Illuminate\Support\LazyCollection`. Lazy kolekcije omogućuju vam korištenje mnogih metoda prikupljanja dostupnih na tipičnim Laravel zbirkama dok istovremeno učitate samo jedan model u memoriju:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Iako `cursor` metoda koristi puno manje memorije od uobičajenog upita (držanjem samo jednog modela Eloquent u memoriji odjednom), svejedno će joj na kraju ponestati memorije. To je zbog toga što PHP-ov PDO upravljački program interno kešira sve neobrađene rezultate upita u svom buffer-u. Ako imate posla s vrlo velikim brojem Eloquentovih zapisa, razmislite o korištenju lazy metode umjesto toga.

### *Napredni podupiti*

#### *Odabir podupita (engl. Subquery Selects)*

Eloquent također nudi naprednu podršku za podupit, koja vam omogućuje izvlačenje informacija iz povezanih tablica u jednom upitu. Na primjer, zamislimo da imamo tablicu leta destinations i tablicu flightsdo odredišta. Tablica flights sadrži arrived\_at stupac koji pokazuje kada je let stigao na odredište.

Koristeći funkciju podupita koja je dostupna alatima za izradu upita select i addSelect metodama, možemo odabrati sve destinations i naziv leta koji je zadnji stigao na to odredište pomoću jednog upita:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
])
```

```
->limit(1)
])->get();
```

### *Redoslijed podupita*

Osim toga, query builder-ova `orderBy` u funkcija podržava podupite. Nastavljajući koristiti naš primjer leta, možemo koristiti ovu funkciju za sortiranje svih odredišta na temelju toga kada je zadnji let stigao na to odredište. Ponovo, ovo se može učiniti tokom izvršavanja jednog upita baze podataka:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

### *Dohvaćanje pojedinačnih modela/agregata*

Osim dohvaćanja svih zapisa koji odgovaraju danom upitu, možete također dohvaćati pojedinačne zapise pomoću metoda `find`, `first` ili `firstWhere`. Umjesto vraćanja kolekcije modela, ove metode vraćaju jednu instancu modela:

```
use App\Models\Flight;

// Dohvaćanje modela po primarnom ključu...
$flight = Flight::find(1);

// Dohvati prvi model koji odgovara ograničenjima upita...
$flight = Flight::where('active', 1)->first();

// Alternativa dohvaćanju prvog modela koji odgovara ograničenjima upita...
$flight = Flight::firstWhere('active', 1);
```

Ponekad ćete možda htjeti izvršiti neku drugu radnju ako nema rezultata. Metode `findOr` i `firstOr` vratit će jednu instancu modela ili, ako se ne pronađu rezultati, izvršiti zadano zatvaranje. Vrijednost koju vraća zatvaranje smatrat će se rezultatom metode:

```
$flight = Flight::findOr(1, function () {
    // ...
});

$flight = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

### *Not Found Iznimke*

Ponekad ćete možda htjeti baciti iznimku ako model nije pronađen. Ovo je osobito korisno u rutama ili kontrolerima. Metode `findOrFail` i `firstOrFail` će dohvatiti prvi rezultat upita; međutim, ako se ne pronađe rezultat, `Illuminate\Database\Eloquent\ModelNotFoundException` bit će izbačeno:

```
$flight = Flight::findOrFail(1);
```

```
$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

Ako `ModelNotFoundException` nije uhvaćen, HTTP odgovor 404 automatski se šalje natrag klijentu:

```
use App\Models\Flight;
```

```
Route::get('/api/flights/{id}', function (string $id) {  
    return Flight::findOrFail($id);  
});
```

### *Dohvaćanje ili kreiranje modela*

Metoda `firstOrCreate` će pokušati locirati zapis baze podataka pomoću zadanih parova stupaca/vrijednosti. Ako se model ne može pronaći u bazi podataka, umetnut će se zapis s atributima koji proizlaze iz spajanja prvog argumenta polja s izbornim argumentom drugog polja:

Metoda `firstOrCreate`, poput `firstOrCreate`, pokušat će locirati zapis u bazi podataka koji odgovara zadanim atributima. Međutim, ako se model ne pronađe, vratit će se nova instanca modela. Imajte na umu da model koji je vratio `firstOrCreate` još nije sačuvan u bazi podataka. Morat ćete ručno pozvati `save` metodu da biste je zadržali:

```
use App\Models\Flight;
```

```
// Dohvatite let po imenu ili ga stvorite ako ne postoji...
```

```
$flight = Flight::firstOrCreate([  
    'name' => 'London za Paris'  
]);
```

```
// Dohvatite let po imenu ili ga kreiraj s atributima naziv, kašnjenje i vrijeme dolaska...
```

```
$flight = Flight::firstOrCreate(  
    ['name' => 'London za Paris'],  
    ['delayed' => 1, 'arrival_time' => '11:30']  
);
```

// Dohvatite let po imenu ili instanciraj novu instancu leta...

```
$flight = Flight::firstOrCreate([
    'name' => 'London za Paris'
]);

// Dohvati let po imenu ili instanciraj atributima imena, kašnjenja i vremena
dolaska...
$flight = Flight::firstOrCreate(
    ['name' => 'Tokio za Sidnej'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

### *Dohvaćanje agregata*

Prilikom interakcije s Eloquent modelima, također možete koristiti `count`, `sum`, `max` i druge agregatne metode koje nudi Laravel alat za izradu upita . Kao što možete očekivati, ove metode vraćaju skalarnu vrijednost umjesto instance modela Eloquent:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

### *Umetanje i ažuriranje modela*

#### *Umetci (engl. Inserts)*

Naravno, kada koristimo Eloquent, ne trebamo samo dohvaćati modele iz baze podataka. Također moramo umetnuti nove zapise. Srećom, Eloquent to čini jednostavnim. Da biste umetnuli novi zapis u bazu podataka, trebali biste instancirati novu instancu modela i postaviti attribute na modelu. Zatim pozovite `save` metodu na instanci modela:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Pohrani novi let u bazu podataka.
     */
    public function store(Request $request): RedirectResponse
```



```
{  
    // Validiraj zahtjev...  
  
    $flight = new Flight;  
  
    $flight->name = $request->name;  
  
    $flight->save();  
  
    return redirect('/flights');  
}  
}
```

U ovom primjeru dodjeljujemo `name` polje iz dolaznog HTTP zahtjeva atributu `name` instance `App\Models\Flight` modela. Kada pozovemo `save` metodu, zapis će biti umetnut u bazu podataka. Model `created_at` i `updated_at` vremenske oznake automatski će se postaviti kada se `save` metoda pozove, tako da nema potrebe za njihovim ručnim postavljanjem.

Alternativno, možete koristiti `create` metodu za „spremanje“ novog modela pomoću jedne PHP naredbe. Umetnuta instanca modela bit će vam vraćena metodom `create`:

```
use App\Models\Flight;  
  
$flight = Flight::create([  
    'name' => 'London za Paris',  
]);
```

Međutim, prije korištenja `create` metode morat ćete navesti svojstvo `fillable` ili `guarded` na svojoj klasi modela. Ova su svojstva potrebna jer su svi Eloquent modeli prema zadanim postavkama zaštićeni od ranjivosti masovne dodjele. Da biste saznali više o masovnoj dodjeli, pogledajte dokumentaciju o masovnoj dodjeli.

### Ažuriranja

Metoda `save` se također može koristiti za ažuriranje modela koji već postoje u bazi podataka. Da biste ažurirali model, trebali biste ga dohvatiti i postaviti sve atribute koje želite ažurirati. Zatim biste trebali pozvati metodu `save` modela. Opet, `updated_at` vremenska oznaka (engl. timestamp) će se automatski ažurirati, tako da nema potrebe za ručnim postavljanjem njene vrijednosti:

```
use App\Models\Flight;  
  
$flight = Flight::find(1);  
  
$flight->name = 'Paris za London';
```

```
$flight->save();
```

Povremeno ćete možda morati ažurirati postojeći model ili izraditi novi model ako ne postoji odgovarajući model. Kao i `firstOrCreate` metoda, `updateOrCreate` metoda zadržava model, tako da nema potrebe za ručnim pozivanjem `save` metode.

U donjem primjeru, ako postoji let s `departure` lokacijom Oakland i `destination` lokacijom San Diego, njegovi stupci `price` i `discounted` bit će ažurirani. Ako takav let ne postoji, kreirat će se novi let koji ima atribut koji proizlaze iz spajanja prvog niza argumenata s drugim nizom argumenata:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

### Masovna ažuriranja (engl. Mass Updates)

Ažuriranja se također mogu izvesti prema modelima koji odgovaraju danom upitu. U ovom primjeru, svi letovi koji jesu `active` i imaju `destination` od San Diego bit će označeni kao odgođeni:

```
Flight::where('active', 1)
    ->where('destination', 'San Diego')
    ->update(['delayed' => 1]);
```

### UPOZORENJE:



Prilikom izdavanja masovnog ažuriranja putem Eloquent, događaji `saving`, `saved`, `updating` i `updated` model neće se pokretati za ažurirane modele. To je zato što se modeli nikada zapravo ne dohvaćaju prilikom izdavanja masovnog ažuriranja.

### Ispitivanje promjena atributa

Eloquent pruža metode `isDirty`, `isClean`, i `wasChanged` za ispitivanje unutrašnjeg stanja vašeg modela i utvrđivanje kako su se njegovi atributi promijenili u odnosu na vrijeme kada je model izvorno dohvaćen.

Metoda `isDirty` utvrđuje je li neki od atributa modela promijenjen otkad je model dohvaćen. Metodi možete proslijediti određeni naziv atributa ili niz atributa `isDirty` kako biste utvrdili je li neki od atributa "prljav". Metoda `isClean` će odrediti je li atribut ostao nepromijenjen otkad je model dohvaćen. Ova metoda također prihvaća izborni argument atributa:

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);
```

```
$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

Metoda `wasChanged` utvrđuje jesu li neki atributi promijenjeni kada je model posljednji put spremljen unutar trenutnog ciklusa zahtjeva. Ako je potrebno, možete proslijediti naziv atributa da vidite je li određeni atribut promijenjen:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

Metoda `getOriginal` vraća niz koji sadrži originalne attribute modela bez obzira na bilo kakve promjene na modelu otkako je dohvaćen. Ako je potrebno, možete proslijediti određeni naziv atributa da biste dobili izvornu vrijednost određenog atributa:

```
$user = User::find(1);
```

```
$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Matrica s originalnim atributima...
```

### Masovno pridruživanje

Možete koristiti `create` metodu za „spremanje“ novog modela pomoću jedne PHP naredbe. Umetnuta instanca modela bit će vam vraćena metodom:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London za Pariz',
]);
```

Međutim, prije korištenja `create` metode morat ćete navesti svojstvo `fillable` ili `guarded` na svojoj klasi modela. Ova su svojstva potrebna jer su svi Eloquent modeli prema zadanim postavkama zaštićeni od ranjivosti masovne dodjele.

Ranjivost masovnog dodjeljivanja događa se kada korisnik proslijedi neočekivano polje HTTP zahtjeva i to polje promijeni stupac u vašoj bazi podataka koji niste očekivali. Na primjer, zlonamjerni korisnik može poslati parametar `is_admin` pomoću HTTP zahtjeva, koji se zatim prosljeđuje metodi vašeg modela `create`, dopuštajući korisniku da eskalira administratoru.

Dakle, da biste započeli, trebali biste definirati koje atribute modela želite učiniti masovno dodjeljivim. To možete učiniti pomoću `$fillable` svojstva na modelu. Na primjer, učinimo `name` atribut mase našeg Flightmodela dodijeljivim:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Atributi koji se mogu masovno dodijeliti.
     *
     * @var array
     */
```

```
protected $fillable = ['name'];
}
```

Nakon što odredite koji se atributi mogu masovno dodjeljivati, možete koristiti `create` metodu za umetanje novog zapisa u bazu podataka. Metoda `create` vraća novostvorenu instancu modela:

```
$flight = Flight::create(['name' => 'London za Pariz']);
```

Ako već imate instancu modela, možete koristiti metodu `fill` da je popunite matricom atributa:

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

#### *Masovno dodjeljivanje i JSON stupci*

Prilikom dodjele JSON stupca, ključ za masovnu dodjelu svakog stupca mora biti naveden u nizu vašeg modela `$fillable`. Radi sigurnosti, Laravel ne podržava ažuriranje ugniježđenih JSON atributa prilikom korištenja svojstva `guarded`:

```
/**
 * Atributi koji su masovno dodjeljeni.
 *
 * @var array
 */
protected $fillable = [
    'options->enabled',
];
```

#### *Dopušteno masovno dodjeljivanje*

Ako želite učiniti sve svoje atribute masovno dodijeljivim, možete definirati `$guarded` svojstvo modela kao prazno polje. Ako odlučite ukloniti zaštitu svog modela, trebali biste posebno paziti da uvijek ručno izradite nizove proslijeđene Eloquentovim `fill`, `create` i `update` metodama:

```
/**
 * Atributi koji se ne mogu masovno dodijeliti.
 *
 * @var array
 */
protected $guarded = [];
```

### Iznimke masovnog dodjeljivanja

Prema zadanim postavkama, atributi koji nisu uključeni u `$fillable` matricu tiho se odbacuju prilikom izvođenja operacija masovne dodjele. U produkciji je ovo očekivano ponašanje; međutim, tokom lokalnog razvoja može dovesti do zabune o tome zašto promjene modela ne stupaju na snagu.

Ako želite, možete uputiti Laravel da izbací iznimku kada pokušava ispuniti atribut koji se ne može ispuniti pozivanjem `preventSilentlyDiscardingAttributes` metode. Obično se ova metoda treba pozvati u `boot` metodi vaše aplikacijske `AppServiceProvider` klase:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu (engl. services).
 */
public function boot(): void
{
    Model::preventSilentlyDiscardingAttributes($this->app->isLocal());
}
```

### Upsert-i

Eloquentova `upsert` metoda može se koristiti za ažuriranje ili stvaranje zapisa u jednoj, atomskoj operaciji. Prvi argument metode sastoji se od vrijednosti za umetanje ili ažuriranje, dok drugi argument navodi stupce koji jedinstveno identificiraju zapise unutar pridružene tablice. Treći i posljednji argument metode je niz stupaca koje treba ažurirati ako odgovarajući zapis već postoji u bazi podataka. Metoda `upsert` će automatski postaviti vremenske oznake `created_at` i `updated_at` ako su vremenske oznake omogućene na modelu:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], uniqueBy: ['departure', 'destination'], update: ['price']);
```

#### UPOZORENJE:



Sve baze podataka osim SQL Servera zahtijevaju da stupci u drugom argumentu metode `upsert` imaju "primarni" ili "jedinstveni" indeks. Osim toga, upravljački programi baze podataka MariaDB i MySQL ignoriraju drugi argument metode `upsert` i uvijek koriste "primarne" i "jedinstvene" indekse tablice za otkrivanje postojećih zapisa.

### Brisanje modela

Da biste izbrisali model, možete pozvati `delete` metodu na instanci modela:

```
use App\Models\Flight;

$flight = Flight::find(1);
```

```
$flight->delete();
```

Možete pozvati `truncate` metodu za brisanje svih zapisa baze podataka povezanih s modelom. Operacija `truncate` će također poništiti sve ID-ove koji se automatski povećavaju u tablici pridruženoj modelu:

```
Flight::truncate();
```

#### *Brisanje postojećeg modela pomoću primarnog ključa*

U gornjem primjeru dohvaćamo model iz baze podataka prije pozivanja `delete` metode. Međutim, ako znate primarni ključ modela, možete izbrisati model bez izričitog dohvaćanja pozivanjem metode `destroy`. Osim prihvatanja jednog primarnog ključa, `destroy` metoda će prihvatiti više primarnih ključeva, niz primarnih ključeva ili kolekciju primarnih ključeva:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```

Ako koristite modele mekog brisanja, možete trajno izbrisati modele putem `forceDestroy` metode:

```
Flight::forceDestroy(1);
```

#### **UPOZORENJE:**



Metoda `destroy` učitava svaki model zasebno i poziva `delete` metodu tako da se događaji `deleting` i `deleted` ispravno šalju za svaki model.

#### *Brisanje modela korištenjem upita*

Naravno, možete izraditi Eloquent upit za brisanje svih modela koji odgovaraju kriterijima vašeg upita. U ovom primjeru ćemo izbrisati sve letove koji su označeni kao neaktivni. Kao i masovna ažuriranja, masovna brisanja neće otpremiti događaje modela za modele koji se brišu:

```
$deleted = Flight::where('active', 0)->delete();
```

#### **UPOZORENJE:**



Prilikom izvršavanja naredbe masovnog brisanja putem Eloquenta, događaji modela `deleting` i `deleted` neće se slati za izbrisane modele. To je zato što se modeli nikada zapravo ne dohvaćaju prilikom izvođenja naredbe `delete`.

*Meko brisanje (engl. Soft Deleting)*

Osim stvarnog uklanjanja zapisa iz vaše baze podataka, Eloquent također može „soft delete“ modele. Kada se modeli meko izbrišu, oni se zapravo ne uklanjaju iz vaše baze podataka. Umjesto toga, `deleted_at` na modelu se postavlja atribut koji označava datum i vrijeme kada je model "obrisan". Da biste omogućili meko brisanje za model, dodajte svojstvo `Illuminate\Database\Eloquent\SoftDeletes` modelu:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

**NAPOMENA:**

`SoftDeletes` trait (svojstvo) će automatski baciti `deleted_at` atribut na instancu `DateTime` / `Carbon` za vas.

Također biste trebali dodati `deleted_at` stupac u svoju tablicu baze podataka. Alat za izgradnju Laravel sheme sadrži pomoćnu metodu za stvaranje ovog stupca:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Sada, kada pozovete `delete` metodu na modelu, `deleted_at` stupac će biti postavljen na trenutni datum i vrijeme. Međutim, zapis baze podataka modela ostat će u tablici. Prilikom postavljanja upita modelu koji koristi meko brisanje, meko izbrisani modeli automatski će biti isključeni iz svih rezultata upita.

Da biste utvrdili je li određena instanca modela meko izbrisana, možete upotrijebiti metodu `trashed`:



**NAPOMENA:**

```
if ($flight->trashed()) {
    // ...
}
```

*Vraćanje meko izbrisanih modela*

Ponekad ćete možda poželjeti "poništiti brisanje" meko izbrisanog modela. Za vraćanje meko izbrisanog modela, možete pozvati `restore` metodu na instanci modela. Metoda `restore` će postaviti stupac modela `deleted_at` na `null`:

```
$flight->restore();
```

Također možete koristiti `restore` metodu u upitu za vraćanje više modela. Opet, kao i druge "masovne" operacija, ovo neće otpremiti nijedan događaj modela za modele koji se vraćaju:

```
Flight::withTrashed()
    ->where('airline_id', 1)
    ->restore();
```

Metoda `restore` se također može koristiti pri izgradnji upita odnosa :

```
$flight->history()->restore();
```

*Trajno brisanje modela*

Ponekad ćete možda morati doista ukloniti model iz svoje baze podataka. Možete koristiti `forceDelete` metodu za trajno uklanjanje meko izbrisanog modela iz tablice baze podataka:

```
$flight->forceDelete();
```

Također možete koristiti `forceDelete` metodu kada gradite `Eloquent` upite odnosa:

```
$flight->history()->forceDelete();
```

*Upit meko obrisanim modelima**Uključivanje meko obrisanih modela*

Kao što je gore navedeno, meko izbrisani modeli automatski će biti isključeni iz rezultata upita. Međutim, možete natjerati meko izbrisane modele da budu uključeni u rezultate upita pozivanjem `withTrashed` metode na upitu:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
    ->where('account_id', 1)
```

```
->get();
```

`withTrashed` metoda se također može pozvati prilikom izgradnje upita odnosa:

```
$flight->history()->withTrashed()->get();
```

Dohvaćanje samo meko izbrisanih modela

Metoda `onlyTrashed` će dohvatiti samo meko izbrisane modele:

```
$flights = Flight::onlyTrashed()
    ->where('airline_id', 1)
    ->get();
```

### *Modeli rezidbe (engl. Pruning Models)*

Ponekad ćete možda htjeti povremeno brisati modele koji više nisu potrebni. Da biste to postigli, možete dodati `Illuminate\Database\Eloquent\Prunable` svojstvo ili `Illuminate\Database\Eloquent\MassPrunable` modelima koje želite povremeno orezati. Nakon dodavanja jedne od svojstava modelu, implementirajte `prunable` metodu koja vraća Eloquent alat za izradu upita koji rješava modele koji više nisu potrebni:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Kada označavate modele kao `Prunable`, također možete definirati `pruning` metodu na modelu. Ova metoda će biti pozvana prije brisanja modela. Ova metoda može biti korisna za brisanje svih dodatnih resursa povezanih s modelom, kao što su pohranjene datoteke, prije nego što se model trajno ukloni iz baze podataka:

```
/**
 * Pripremi model za pruning.
 */
protected function pruning(): void
{
    // ...
}
```

Nakon konfiguriranja modela koji se može orezivati, trebali biste zakazati `model:prune` naredbu Artisan u datoteci svoje aplikacije `routes/console.php`. Slobodni ste odabrati odgovarajući interval u kojem se ova naredba treba pokrenuti:

```
use Illuminate\Support\Facades\Schedule;

Schedule::command('model:prune')->daily();
```

Iza kulisa, `model:prune` naredba će automatski detektirati „Prunable“ modele unutar direktorija vaše aplikacije `app/Models`. Ako su vaši modeli na drugoj lokaciji, možete upotrijebiti opciju `--model` za navođenje naziva klase modela:

```
Schedule::command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

Ako želite isključiti određene modele iz skraćivanja dok skraćujete sve druge otkrivene modele, možete koristiti opciju `--except`:

```
Schedule::command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

Možete testirati svoj `prunable` upit izvršavanjem `model:prune` naredbe s `--pretend` opcijom. Prilikom pretvaranja, `model:prune` naredba će jednostavno izvijestiti koliko bi zapisa bilo obrezano da se naredba stvarno pokrene:

```
php artisan model:prune --pretend
```

**UPOZORENJE:**

Modeli mekog brisanja bit će trajno izbrisani (`forceDelete`) ako odgovaraju upitu koji se može rezati.

*Masovna rezidba (engl. Mass Pruning)*

Kada su modeli označeni `Illuminate\Database\Eloquent\MassPrunable` svojstvom (trait), modeli se brišu iz baze podataka pomoću upita za masovno brisanje. Zato se pruning metoda neće pozivati, niti će se deleting i deleted događaji modela otpremati. To je zato što se modeli nikada zapravo ne dohvaćaju prije brisanja, čime je proces rezidbe mnogo učinkovitiji:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

*Repliciranje modela*

Možete stvoriti nesprenjenu kopiju postojeće instance modela pomoću `replicate` metode. Ova je metoda osobito korisna kada imate instance modela koje dijele mnogo istih atributa:

```
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);
```

```
]);  
  
$billing = $shipping->replicate()->fill([  
    'type' => 'billing'  
]);  
  
$billing->save();
```

Da biste isključili jedan ili više atributa iz repliciranja na novi model, možete proslijediti polje metodi `replicate`:

```
$flight = Flight::create([  
    'destination' => 'LAX',  
    'origin' => 'LHR',  
    'last_flown' => '2020-03-04 11:00:00',  
    'last_pilot_id' => 747,  
]);  
  
$flight = $flight->replicate([  
    'last_flown',  
    'last_pilot_id'  
]);
```

### *Doseg upita*

#### *Globalni dosezi*

Globalni doseg omogućuje dodavanje ograničenja svim upitima za određeni model. Laravelova vlastita funkcija mekog brisanja koristi globalne dosege za dohvaćanje samo "neizbrisanih" modela iz baze podataka. Pisanje vlastitih globalnih opsega može pružiti prikladan, jednostavan način da osigurate da svaki upit za određeni model prima određena ograničenja.

#### *Generiranje dosega*

Za generiranje novog globalnog dosega, možete pozvati naredbu Artisan, koja će generirati opseg smjestiti u direktorij `make:scope` vaše aplikacije `:app/Models/Scopes`

```
php artisan make:scope AncientScope
```

#### Pisanje globalnih opsega

Pisanje globalnog opsega je jednostavno. Prvo upotrijebite `make:scope` naredbu za generiranje klase koja implementira `Illuminate\Database\Eloquent\Scope` interface. Interface `Scope` zahtijeva

implementaciju jedne metode: `apply`. Metoda `apply` može dodati `where` ograničenja ili druge vrste klauzula upitu prema potrebi:

```
<?php

namespace App\Models\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Primijeni doseg za Eloquent Eloquent query builder.
     */
    public function apply(Builder $builder, Model $model): void
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

**NAPOMENA:**

Ako vaš globalni opseg dodaje stupce u klauzulu odabira upita, trebali biste koristiti metodu `addSelect` umjesto `select`. To će spriječiti nenamjernu zamjenu postojeće klauzule odabira upita.

Primjena globalnih opsega

Da biste dodijelili globalni opseg modelu, možete jednostavno postaviti atribut `ScopedBy` na model:

```
<?php

namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Attributes\ScopedBy;

#[ScopedBy([AncientScope::class])]
class User extends Model
{
    //
}
```

Ili, možete ručno registrirati globalni doseg nadjačavanjem metode modela `booted` i pozivanjem metode modela `addGlobalScope`. Metoda `addGlobalScope` prihvaća instancu vašeg opsega kao jedini argument:

```
<?php

namespace App\Models;

use App\Models\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * "booted" metoda modela.
     */
    protected static function booted(): void
    {
        static::addGlobalScope(new AncientScope);
    }
}
```

Nakon dodavanja dosega iz gornjeg primjera u `App\Models\Usermodel`, poziv metode `User::all()` će izvršiti sljedeći SQL upit:

```
select * from `users` where `created_at` < 0021-02-23 00:00:00
```

### Anonimni globalni dosezi

Eloquent vam također omogućuje definiranje globalnih dosega pomoću anonimnih funkcija (engl. closures), što je posebno korisno za jednostavne dosege koji ne zahtijevaju posebnu vlastitu klasu. Kada definirate globalni doseg pomoću closure, trebali biste navesti naziv dosega po vlastitom izboru kao prvi argument `addGlobalScope` metode:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * "booted" metoda modela.
     */
}
```

```
*/
protected static function booted(): void
{
    static::addGlobalScope('ancient', function (Builder $builder) {
        $builder->where('created_at', '<', now()->subYears(2000));
    });
}
}
```

### Uklanjanje globalnih dosega

Ako želite ukloniti globalni doseg za određeni upit, možete koristiti metodu `withoutGlobalScope`. Ova metoda prihvaća naziv klase globalnog dosega kao jedini argument:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Ili, ako ste definirali globalni opseg korištenjem closure, trebali biste proslijediti naziv niza koji ste dodijelili globalnom opsegu:

```
User::withoutGlobalScope('ancient')->get();
```

Ako želite ukloniti nekoliko ili čak sve globalne dosege upita, možete koristiti metodu `withoutGlobalScopes`:

// Ukloni sve globalne dosege...

```
User::withoutGlobalScopes()->get();
```

// Ukloni neke globalne dosege...

```
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

### Lokalni doseg

Lokalni doseg vam omogućuje da definirate uobičajene skupove ograničenja upita koje možete jednostavno ponovno koristiti u svojoj aplikaciji. Na primjer, možda ćete morati često dohvaćati sve korisnike koji se smatraju "popularnim". Da biste definirali doseg, dodajte metodi Eloquent modela ispred `scope`.

Dosezi bi uvijek trebali vraćati istu instancu alata za izradu upita ili `void`:

```
<?php

namespace App\Models;
```



```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Doseg upita da uključi samo popularne korisnike.
     */
    public function scopePopular(Builder $query): void
    {
        $query->where('votes', '>', 100);
    }

    /**
     * Doseg upita da uključi samo aktivne korisnike.
     */
    public function scopeActive(Builder $query): void
    {
        $query->where('active', 1);
    }
}
```

#### *Korištenje lokalnog dosega*

Nakon što je doseg definiran, možete pozvati metode dosega prilikom postavljanja upita modelu. Međutim, ne biste trebali uključiti `scope` prefiks kada pozivate metodu. Možete čak i lančano vezati pozive na različite dosege:

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Kombiniranje više dosega modela Eloquent pomoću `or` operatora upita može zahtijevati upotrebu zatvaranja za postizanje ispravnog logičkog grupiranja:

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

Međutim, budući da to može biti glomazno, Laravel pruža `orWhere` metodu "višeg reda" koja vam omogućuje tečno ulančavanje dosega bez upotrebe anonimnih funkcija (engl. closures):

```
$users = User::popular()->orWhere->active()->get();
```

### *Dinamički doseg*

Ponekad ćete možda htjeti definirati opseg koji prihvaća parametre. Da biste započeli, samo dodajte svoje dodatne parametre u potpis metode opsega. Parametri opsega trebaju biti definirani nakon `$query` parametra:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Doseg upita tako da uključuje samo korisnike određenog tipa.
     */
    public function scopeOfType(Builder $query, string $type): void
    {
        $query->where('type', $type);
    }
}
```

Nakon što su očekivani argumenti dodani potpisu vaše metode dosega, možete proslijediti argumente prilikom pozivanja dosega:

```
$users = User::ofType('admin')->get();
```

### *Usporedba modela*

Ponekad ćete možda trebati utvrditi jesu li dva modela "ista" ili ne. Metode `is` i `isNot` mogu se koristiti za brzu provjeru imaju li dva modela isti primarni ključ, tablicu i vezu s bazom podataka ili ne:

```
if ($post->is($anotherPost)) {
    // ...
}

if ($post->isNot($anotherPost)) {
    // ...
}
```

Metode `is` i `isNot` također su dostupne kada se koriste `belongsTo`, `hasOne`, `morphTo` i `morphOne` odnosi. Ova je metoda osobito korisna kada želite usporediti povezani model bez postavljanja upita za dohvaćanje tog modela:

```
if ($post->author()->is($user)) {
    // ...
}
```

### Događaji

#### NAPOMENA:



Želite li svoje Eloquent događaje emitirati direktno u svoju aplikaciju na strani klijenta? Pogledajte Laravel model emitiranja događaja .

Eloquent modeli šalju nekoliko događaja, omogućujući vam da se uključite u sljedeće trenutke u životnom ciklusu modela: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleting`, `forceDeleted`, `restoring`, `restored`, i `replicating`.

Događaj `retrieved` će se poslati kada se postojeći model dohvati iz baze podataka. Kada se prvi put spremi novi model, događaji `creating` i `created` će se poslati. Događaji `updating` / `updated` će se poslati kada se postojeći model modificira i pozove `save` metoda. `saving` / `saved` događaji će se poslati kada se model kreira ili ažurira - čak i ako atributi modela nisu promijenjeni. Nazivi događaja koji završavaju sa `-ing` šalju se prije nego što se zadrže bilo kakve promjene u modelu, dok se događaji koji završavaju sa `-ed` otpremaju nakon što se zadrže promjene na modelu.

Da biste počeli slušati događaje modela, definirajte `$dispatchesEvents` svojstvo na svom Eloquent modelu. Ovo svojstvo preslikava različite točke životnog ciklusa modela Eloquent u vaše vlastite klase događaja . Svaka klasa događaja modela trebala bi očekivati da primi instancu pogođenog modela kroz svoj konstruktor:

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * Mapa događaja za model.
     *
     * @var array<string, string>
     */
    protected $dispatchesEvents = [
```

```

        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}

```

Nakon definiranja i mapiranja vaših Eloquent događaja, možete koristiti slušatelje događaja za obradu događaja.

#### UPOZORENJE:



Prilikom izdavanja upita za masovno ažuriranje ili brisanje putem Eloquenta, događaji `saved`, `updated`, `deleting` i `deleted` model neće se slati za zahvaćene modele. To je zato što se modeli nikada zapravo ne dohvaćaju prilikom izvođenja masovnih ažuriranja ili brisanja.

#### Korištenje Closures

Umjesto upotrebe prilagođenih klasa događaja, možete registrirati anonimne funkcije (engl. closure) koja se izvršavaju kada se pošalju različiti događaji modela. Obično biste trebali registrirati ove anonimne funkcije (engl. closure) u `booted` metodi vašeg modela:

```

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Metoda "podizanja" (engl. "booted") modela
     */
    protected static function booted(): void
    {
        static::created(function (User $user) {
            // ...
        });
    }
}

```

Ako je potrebno, možete upotrijebiti anonimne slušatelje događaja u redu čekanja (engl. queueable anonymous event listeners) prilikom registracije događaja modela. Ovo će uputiti Laravel da izvrši slušatelja događaja modela u pozadini koristeći red čekanja vaše aplikacije (engl. queue):

```
use function Illuminate\Events\queueable;

static::created(queueable(function (User $user) {
    // ...
}));
```

### *Promatrači*

#### *Definiranje promatrača*

Ako slušate mnoge događaje na određenom modelu, možete koristiti promatrače za grupiranje svih slušatelja u jednu klasu. Observer klase imaju nazive metoda koji odražavaju Eloquent događaje koje želite slušati. Svaka od ovih metoda prima pogođeni model kao jedini argument. Naredba `make:observer` Artisan je najlakši način za stvaranje nove observer klase:

```
php artisan make:observer UserObserver --model=User
```

Ova naredba će postaviti novog promatrača u vaš `app/Observers` direktorij. Ako ovaj direktorij ne postoji, Artisan će ga kreirati za vas. Vaš novi observer izgledat će ovako:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Rukuj User-om da "kreira" događaj.
     */
    public function created(User $user): void
    {
        // ...
    }

    /**
     * Rukuj User-om da "ažurira" događaj.
     */
    public function updated(User $user): void
    {
        // ...
    }

    /**
```

```
    * Rukuj User-om "obrisan" događaj.
    */
    public function deleted(User $user): void
    {
        // ...
    }

    /**
     * Rukuj User-om da "obnavi" događaj.
     */
    public function restored(User $user): void
    {
        // ...
    }

    /**
     * Rukuj User-om na "forceDeleted" događaj.
     */
    public function forceDeleted(User $user): void
    {
        // ...
    }
}
```

Da biste registrirali promatrača, možete postaviti `ObservedBy` atribut na odgovarajući model:

```
use App\Observers\UserObserver;
use Illuminate\Database\Eloquent\Attributes\ObservedBy;

#[ObservedBy([UserObserver::class])]
class User extends Authenticatable
{
    //
}
```

Ili, možete ručno registrirati promatrača pozivanjem `observe` metode na modelu koji želite promatrati. Možete registrirati promatrače u `boot` metodi `AppServiceProvider` klase vaše aplikacije:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Pokreni (engl. Bootstrap) bilo koju aplikacijsku uslugu.
```

```
*/  
public function boot(): void  
{  
    User::observe(UserObserver::class);  
}
```

**NAPOMENA:**

Postoje dodatni događaji koje promatrač može slušati, kao što su `saving` i `retrieved`. Ti su događaji opisani u dokumentaciji događaja .

**Promatrači i transakcije baze podataka**

Kada se modeli stvaraju unutar transakcije baze podataka, možda ćete htjeti uputiti promatrača da samo izvrši svoje rukovatelje događajima (engl. event handlers) nakon što se transakcija baze podataka preda. To možete postići implementacijom `ShouldHandleEventsAfterCommit` interface-a na vašem promatraču. Ako transakcija baze podataka nije u toku, rukovatelji događajima (engl. event handlers) će se odmah izvršiti:

```
<?php  
  
namespace App\Observers;  
  
use App\Models\User;  
use Illuminate\Contracts\Events\ShouldHandleEventsAfterCommit;  
  
class UserObserver implements ShouldHandleEventsAfterCommit  
{  
    /**  
     * Handle the User "created" event.  
     */  
    public function created(User $user): void  
    {  
        // ...  
    }  
}
```

*Isključivanje događaja*

Možda ćete povremeno trebati privremeno "utišati" sve događaje koje pokreće model. Ovo možete postići metodom `withoutEvents`. Metoda `withoutEvents` prihvaća zatvaranje kao jedini argument. Bilo koji kod koji se izvrši unutar ovog zatvaranja neće slati događaje modela, a bilo koja vrijednost vraćena zatvaranjem bit će vraćena `withoutEvents` metodom:

```
use App\Models\User;

$user = User::withoutEvents(function () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Snimanje jednog modela bez događaja

Ponekad ćete možda poželjeti "spremiti" dati model bez otpremanja ikakvih događaja. To možete postići pomoću `saveQuietly` metode:

```
$user = User::findOrFail(1);

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

Također možete "ažurirati", "izbrisati", "soft brisati", "vratiti" i "replicirati" dati model bez otpremanja bilo kakvih događaja:

```
$user->deleteQuietly();
$user->forceDeleteQuietly();
$user->restoreQuietly();
```



### Provjera latinica

```
$string_exp = "/^[A-Za-zšŠđĐžŽčĆć.'-]+$/" ;

if(!preg_match($string_exp,$first_name)) {

    $error_message .= '<p>Ime koje ste unijeli nije validano.</p>';

}
```