Testiranje integracije u distribuiranim sistemima i arhitekturi mikroservisa

---

Uvod i problem (Slajdovi 1–5)

[Slajd 1]

Zdravo svima i dobrodošli u novu lekciju iz naprednog testiranja softvera. Danas se bavimo jednim od najvećih izazova modernog razvoja softvera — testiranjem integracije u distribuiranim sistemima i arhitekturi mikroservisa. Tokom ove lekcije koristićemo Javu, Maven, lagani web framework Javalin, Swagger i alat koji je zvezda današnje priče — Pact.

[Slajd 2]

Naš plan za danas je jasan. Prvo ćemo pogledati zašto klasične metode integracionog testiranja pucaju kada pređemo na mikroservise. Zatim ćemo upoznati rešenje: ugovorom vođeno testiranje, sa posebnim fokusom na pristup vođen potrošačem. Na kraju ćemo sve to videti u praksi kroz konkretan kod, semantičko verzionisanje i CI/CD automatizaciju. Glavni cilj nam je da naučimo kako da budemo potpuno sigurni da dva servisa uspešno komuniciraju, a da pritom ne moramo da podižemo ceo ekosistem aplikacije.

[Slajd 3]

Da bismo razumeli problem, moramo da se podsetimo kako su se aplikacije menjale. Kod tradicionalnih, monolitnih sistema, sve komponente žive u istom memorijskom prostoru. Ako promenite potpis neke metode, kompajler vas odmah zaustavi i prijavi grešku. Međutim, u distribuiranim sistemima, komponente su razdvojene i komuniciraju preko mreže — putem HTTP-a, gRPC-a ili razmenom poruka. Tu kompajler više ne može da nam pomogne. Zato se i fokus QA inženjera drastično pomera: sa testiranja unutrašnjih metoda prelazimo na testiranje mrežne komunikacije i API-ja.

[Slajd 4]

Ranije smo integraciju rešavali poznatim strategijama: Big Bang pristupom gde spojimo sve odjednom, ili postepeno — odozgo nadole pomoću ***stub***-ova, odnosno odozdo nagore pomoću ***driver***-a. Sve ove tehnike imaju smisla, ali su nastale u eri monolita.

[Slajd 5]

Zašto te stare metode nisu dovoljne za mikroservise? Prvo, one polaze od pretpostavke da su sve komponente dostupne u istom okruženju. Drugo, ako pokušamo da testiramo mikroservise kroz klasične End-to-End testove, upadamo u takozvanu ***E2E zamku***. Podizanje desetina mikroservisa, baza podataka i redova poruka za svaki test je presporo i preskupo. Povrh svega, dobijamo ***flaky*** testove — test padne jer je mreža na trenutak štucnula, a ne zato što postoji stvaran bag u kodu. Nama treba način da proverimo interfejs između dva servisa izolovano.

Koncept Contract-Based Testing-a (Slajdovi 6–9)

[Slajd 6]

Rešenje leži u ugovorom vođenom testiranju, odnosno ***Contract-Based Testing***-u. Zamislite to kao pravi poslovni ugovor. Imamo dve strane: **Potrošača** (***Consumer***-a) koji šalje zahtev, i **Proizvođača** (***Provider***-a) koji vraća odgovor. Između njih stoji ugovor — običan JSON ili YAML fajl koji precizno opisuje šta se šalje i šta se očekuje nazad. Najveća prednost je što svaka strana testira svoju usaglašenost sa tim ugovorom potpuno izolovano, bez potrebe za živim sistemom.

[Slajd 7]

Hajde da preciziramo uloge. **Consumer**, ili potrošač, je klijentska komponenta koja inicira API poziv i koristi dobijene podatke. To može biti neka frontend aplikacija ili drugi mikroservis. **Provider**, ili dobavljač, je server ili backend servis koji te podatke pruža i odgovara na zahteve.

[Slajd 8]

Ugovor između ove dve strane može se definisati u dva smera. U ***Provider-Driven*** pristupu, gde se koristi alat **Swagger**, server diktira pravila, objavi specifikaciju, a klijenti moraju tome da se prilagode. Sa druge strane, **Consumer-Driven** pristup okreće igru: klijent definiše tačno ono što mu je potrebno od servera, a server preuzima obavezu da taj postojeći ugovor nikada slučajno ne naruši.

[Slajd 9]

Zašto je ovaj fokus na potrošaču toliko moćan? Prvo, donosi efikasnost — backend tim ne troši vreme na razvijanje polja koja nijedan klijent zapravo ne koristi. Drugo, štiti produkciju — ako programer na backend-u obriše ili preimenuje neko polje, ugovor će odmah alarmirati koji tačno klijent će zbog toga prestati da radi. Treće, dobijamo ekstremno brz feedback — ovi testovi se izvršavaju u samoj build fazi, za svega nekoliko sekundi.

Praktičan primer i OpenAPI (Slajdovi 10–14)

[Slajd 10]

Sada prelazimo na praksu. Zamislite jednostavan sistem sa dva mikroservisa. Imamo `Order-Service` koji upravlja narudžbinama i ima ulogu Consumer-a. Njemu su potrebni podaci o kupcu, pa šalje HTTP GET zahtev ka `User-Service`-u, koji je naš Provider napisan u Javalinu. Razmena ide preko REST-a, a format podataka je JSON.

[Slajd 11]

Da bismo podigli naš Provider servis u Javi, u `pom.xml` fajl dodajemo svega dve osnovne zavisnosti: sam `javalin` kao lagani web framework i `jackson-databind` koji će nam služiti za pretvaranje Java objekata u JSON i obrnuto.

[Slajd 12]

Pogledajte koliko je čist kod našeg `User-Service`-a. Podižemo server na portu 8081 i definišemo rutu `/users/{id}`. Ako klijent zatraži korisnika sa ID-jem `123`, vraćamo status `200`

`OK` i JSON sa podacima o Petru Petroviću. Za bilo koji drugi ID, vraćamo status `404` i poruku da korisnik nije pronađen.

[Slajd 13]

Ovde se često javlja jedna nedoumica: zar nam za sve ovo nije dovoljan Swagger, odnosno OpenAPI? Važno je da razumemo da Swagger i Pact imaju potpuno različite uloge. Swagger služi za dokumentovanje API-ja i daje ljudima čitljivu šemu. Problem sa dokumentacijom je što ona može da zastari ako programer promeni kod, a zaboravi da ažurira dokumentaciju. Pact nije dokumentacija — to je dinamički test koji proverava ponašanje stvarnog, izvršnog koda.

[Slajd 14]

Naravno, Swagger je i dalje izuzetno koristan za preglednost. U Javalinu ga lako dodajemo putem OpenAPI i Swagger plugin-a. Kao što vidite na slajdu, iznad naše metode dodamo `@OpenApi` anotaciju gde opišemo parametre i očekivane odgovore, registrujemo plugin-ove, i na ruti `/swagger` dobijamo interaktivnu dokumentaciju.

Pact u akciji: Consumer strana (Slajdovi 15–19)

[Slajd 15]

Sada da vidimo kako Pact povezuje sve kockice u četiri jasna koraka. Sve kreće od Consumer testa, gde klijent definiše šta očekuje. Kada taj test prođe, Pact generiše JSON fajl — naš ugovor. Zatim taj ugovor uzima Provider i verifikuje ga protiv svog stvarnog koda. Ako i to prođe, imamo potvrđenu usaglašenost.

[Slajd 16]

Krećemo od Consumer strane, odnosno `Order-Service`-a. U Maven dodajemo `pact.consumer` zavisnost za JUnit 5. Šta se dešava kada pokrenemo test? Pact u pozadini podiže lokalni, privremeni Mock Server. Taj server imitira pravi backend isključivo na osnovu pravila koja smo mi zapisali u testu.

[Slajd 17]

Evo kako se taj ugovor piše u Javi. Koristimo `@Pact` anotaciju i `PactBuilder`. Čitamo ga gotovo kao običnu englesku rečenicu: ***given*** — pod uslovom da korisnik 123 postoji; ***uponReceiving*** — kada primiš GET zahtev na putanji `/users/123`; ***willRespondWith*** — odgovori statusom 200 i ovim konkretnim JSON telom.

[Slajd 18]

Kada smo definisali pravilo, pišemo standardni `@Test` metod. Ovde naš stvarni klijentski kod — u ovom slučaju Javin `HttpClient` — šalje zahtev na adresu tog Pact Mock Servera. Zatim proveravamo da li naš klijent uspešno prima status 200 i da li ispravno parsira ime korisnika iz odgovora.

[Slajd 19]

U trenutku kada ovaj test pozeleni, Pact u `target/pacts` folderu automatski kreira JSON fajl. Pogledajte njegov sadržaj: jasno su zapisani ko je Consumer, ko je Provider, i kakva je tačna

interakcija između njih. Ovaj fajl je naš ugovor.

Pact u akciji: Provider strana i stanja (Slajdovi 20–23)

[Slajd 20]
Sada prelazimo na drugu stranu — u tim koji održava `User-Service`. Dodajemo `pact.provider` zavisnost u njihov `pom.xml`. Ovde je ključna razlika: na strani Provider-a **nema nikakvog mock-ovanja**. Podižemo pravi Javalin HTTP server u testnom režimu, a Pact sada preuzima ulogu klijenta, čita onaj JSON ugovor i ispaljuje prave zahteve ka našem kodu.
[Slajd 21]
U klasi `VerificationTest` pomoću `@Provider` anotacije kažemo koji servis testiramo, a pomoću `@PactFolder` pokazujemo gde se nalazi ugovor. U `@BeforeEach` metodi startujemo Javalin server na portu 8081 i usmeravamo Pact ka njemu. Zatim `@TestTemplate` metoda automatski prolazi kroz svaku interakciju iz ugovora i proverava da li server vraća ono što je obećano.
[Slajd 22]
Ali ovde se javlja jedan praktičan problem. Pact će poslati zahtev `GET /users/123`. Šta ako u našoj testnoj bazi taj korisnik trenutno ne postoji? Server će vratiti 404, a ugovor očekuje 200 — i test pada. Zato u ugovoru imamo onu klauzulu `given("User 123 exists")`. Na strani Provider-a to rešavamo pomoću `@State` anotacije, kojom pripremamo sistem pre samog zahteva.
[Slajd 23]
Pogledajte kako to izgleda u kodu. Za stanje `"User 123 exists"`, pre testa upisujemo korisnika u repozitorijum. Za stanje kada ne postoji, brišemo ga. Ovde zapamtite tri zlatna pravila: svaka `@State` metoda mora biti atomična i čista; nikada ne smete hardkodovati podatke u samom Javalin handleru zbog testova; i tekst unutar `given()` i `@State()` mora da se poklapa u svako slovo.

Evolucija API-ja, SemVer i Pact Broker (Slajdovi 24–30)

[Slajd 24]
Hajde da vidimo kako nas ovo spasava u svakodnevnom radu. Šta su takozvani ***API Breaking Changes***? To su sve izmene koje narušavaju kompatibilnost unazad: na primer, ako neko obriše polje `email` iz JSON-a, promeni tip podatka `id` iz String-a u Long, ili promeni putanju URL-a.
[Slajd 25]
Zamislite scenario: programer na `User-Service`-u slučajno obriše polje `email` i pokrene `mvn test`. Pact odmah pokreće verifikaciju ugovora, vidi da `Order-Service` i dalje zahteva to polje, i **build automatski pada**. Greška je zaustavljena na lokalnoj mašini programera, pre nego što je uopšte stigla do produkcije.

[Slajd 26]
Da bismo lakše upravljali ovim promenama, oslanjamo se na Semantičko verzionisanje, odnosno SemVer format: `Major.Minor.Patch`. **Major** menjamo kada uvodimo izmene koje lome postojeće klijente. **Minor** kada dodajemo nove funkcionalnosti koje su bezbedne za stare klijente, a **Patch** kada samo ispravljamo bagove.
[Slajd 27]
Za QA inženjere ovo je direktan signal kako da planiraju testiranje. Kod Minor i Patch verzija znamo da stari ugovori moraju da nastave da rade, pa radimo standardnu regresiju. Međutim, kada vidimo Major verziju, znamo da moramo da kreiramo novu verziju ugovora i koordiniramo deployment između timova.
[Slajd 28]
Naravno, u pravom projektu sa desetinama servisa nećemo ručno slati JSON fajlove mejlom ili preko Slack-a. Za to služi **Pact Broker** — centralni server koji čuva sve ugovore. Consumer tokom svog build-a automatski šalje ugovor na Broker, a Provider ga odatle automatski preuzima radi verifikacije.
[Slajd 29]
Prava magija Pact Broker-a dešava se u CI/CD pipeline-u pomoću alata `can-i-deploy`. Pre nego što pustimo novu verziju `Order-Service`-a na produkciju, pipeline pita Broker: ***"Da li je ova verzija našeg Consumer-a uspešno verifikovana protiv verzije Provider-a koja se trenutno vrti na produkciji?"*** Ako je odgovor ne, deployment se automatski blokira. Nema više nagađanja da li smemo da pustimo novu verziju.
[Slajd 30]
Konfiguracija u Maven-u za ovo je veoma jednostavna. Dodamo `pact` plugin, navedemo URL našeg Pact Broker-a i token za autentifikaciju iz environment promenljive. Nakon uspešnih testova, komanda `mvn pact:publish` šalje ugovor direktno na centralni server.

Idempotentnost i zaključak (Slajdovi 31–35)

[Slajd 31]
Za sam kraj, moramo da se dotaknemo još jednog ključnog koncepta za testiranje distribuiranih sistema — a to je **idempotentnost**. Kažemo da je zahtev idempotentan ako njegovo višestruko izvršavanje ima potpuno isti efekat na stanje sistema kao da je izvršen samo jednom. Zašto je ovo kritično za QA? Zato što u mikroservisima mreža često puca, pa klijenti imaju ugrađene ***retry*** mehanizme koji automatski ponavljaju zahteve ako ne dobiju odgovor na vreme.
[Slajd 32]
Pogledajmo kako se ponašaju standardne HTTP metode. `GET` samo čita podatke, pa je uvek idempotentan. `PUT` potpuno zamenjuje resurs — ako ga pošaljete pet puta sa istim podacima, krajnji rezultat u bazi je uvek isti. `DELETE` takođe — prvi poziv briše resurs, naredni će možda vratiti 404, ali stanje baze ostaje nepromenjeno. Jedini izuzetak je `POST`. Dva ista POST zahteva

mogu da kreiraju dve različite porudžbine i dva puta naplate korisniku, zato `POST` po prirodi nije idempotentan.

[Slajd 33]

Kako to testiramo u praksi? Vrlo jednostavno: pošaljemo `PUT` zahtev za izmenu korisnika, proverimo da li je status 200, a zatim odmah pošaljemo potpuno isti zahtev još jednom. Naš test mora da potvrdi da je baza ostala konzistentna i da nema dupliranih zapisa — što na serveru postižemo metodama poput `saveOrUpdate`.

[Slajd 34]

Da sumiramo današnju lekciju. Umesto sporih i nepouzdanih End-to-End testova, Contract testiranje nam daje brzinu i sigurnost kroz izolovane testove. Consumer-Driven pristup omogućava klijentima da vode razvoj API-ja bez straha od nepredviđenih promena, dok nam SemVer, Pact Broker i idempotentnost pomažu da gradimo otporne distribuirane sisteme.

Pre nego što završimo, ostavljam vam jedno pitanje za razmišljanje i diskusiju: sve ovo smo danas videli na primeru sinhronih HTTP poziva — a kako biste primenili Pact kada servisi komuniciraju asinhrono, putem poruka kroz RabbitMQ ili Kafku?

Kratak odgovor/objašnjenje

Za asinhronu komunikaciju (Kafka, RabbitMQ) koristi se **Message Pact** (asinhroni ugovori).
Glavna razlika: Nema HTTP zahteva i odgovora (***Request/Response***), niti se podiže Mock HTTP server ili broker poruka. Ugovor definiše isključivo **strukturu i sadržaj same poruke** (payload i metapodatke).

Kako izgleda proces u 3 koraka:

1. **Consumer (Subscriber):** U testu definiše kakvu JSON poruku očekuje da primi sa reda. Pact mu tu poruku prosleđuje direktno u metodu koja obrađuje poruke (handler/listener) kako bi proverio da li Consumer ume ispravno da je isparsira. Ako test prođe, generiše se JSON ugovor.
2. **Provider (Publisher):** U svom testu pokreće metodu koja kreira i šalje poruku. Umesto da poruku pošalje na pravu Kafku ili RabbitMQ, Pact presreće taj generisani objekat/JSON u memoriji.
3. **Verifikacija:** Pact upoređuje poruku koju je Provider upravo kreirao sa strukturom zapisanom u ugovoru koji je definisao Consumer.

Suština: Na isti način kao kod REST API-ja, potpuno izolovano testiramo kompatibilnost poruka bez potrebe da u testnom okruženju podižemo Kafku ili RabbitMQ.

[Slajd 35]

Na poslednjem slajdu imate linkove ka zvaničnoj dokumentaciji za Pact, Javalin i SemVer, kao i sjajan tekst Martina Fowlera koji je i postavio temelje ovog obrasca. Hvala vam na pažnji i vidimo se na narednom predavanju!
