چالش‌های رایج در تست کردن اسپرینگ بوت

تست‌نویسی توی اسپرینگ بوت، مخصوصاً برای تازه‌کارها، می‌تونه یه کم گیج‌کننده باشه. اگه ندونیم **تزریق وابستگی (Dependency Injection)** توی اسپرینگ چجوری کار می‌کنه یا **پیکربندی خودکار (Auto-Configuration)** اسپرینگ بوت دقیقاً چیه، احتمالاً شروع می‌کنیم به ریختن یه عالمه **annotation** روی تست‌هامون، به امید اینکه یه جوری کار کنه!

این روش آزمون و خطا شاید توی بعضی موارد جواب بده، ولی معمولاً نتیجه‌ش یه تست **ناقص و غیربهینه** می‌شه. اینجا باهم چند مورد از رایج ترین این اشتباه هات و بررسی میکنیم.

اشتباه شماره ۱: @Mock در مقابل @MockBean

یکی از رایج‌ترین اشتباهات توی تست‌های اسپرینگ بوت، **موک کردن وابستگی‌ها**ست، یعنی اشیایی که کلاس موردنظر ما بهشون وابسته‌ست.

اگه قبلاً با **Mockito** کار کرده باشی، احتمالاً می‌دونی که برای تست‌های واحد (unit tests) می‌تونیم از **@Mock** استفاده کنیم تا یه **mock object** بسازیم. حالا وقتی داریم تست‌های اسپرینگ بوت رو می‌نویسیم، لازم نیست چیزی رو که از **Mockito** بلدی، فراموش کنی.

اما یه نکته‌ی مهم هست: **باید بدونیم دقیقاً چه نوع تستی می‌نویسیم.** تست ما توی یه **Spring TestContext** اجرا می‌شه یا مستقل از اسپرینگ هم می‌تونه کار کنه؟ این نکته تأثیر زیادی روی انتخاب بین **@Mock** و **@MockBean** داره.

پس کِی از @Mock استفاده کنیم و کِی از @MockBean؟

همین که بدونیم تستمون **با یا بدون Spring TestContext** اجرا می‌شه، می‌تونه تعیین کنه که از **@Mock** استفاده کنیم یا **@MockBean**.

@Mock فقط برای **تست‌های واحد (Unit Tests)** که مستقل از اسپرینگ اجرا می‌شن، مناسبه. توی این تست‌ها، معمولاً وابستگی‌ها (Collaborators) رو **موک می‌کنیم** و از طریق **سازنده‌ی عمومی (public constructor)** به کلاس مورد تست تزریق می‌کنیم.

ولی توی تست‌هایی که با **Spring TestContext** اجرا می‌شن، مثل وقتی که از **@SpringBootTest** یا یکی از **Slice Testهای اسپرینگ بوت** استفاده می‌کنیم، داستان یه کم فرق داره. توی این شرایط، هنوزم می‌خوایم

وابستگی کلاس مورد تست رو موک کنیم، ولی این بار **اسپرینگ خودش همه‌ی Beanها رو جمع‌آوری می‌کنه و تزریق وابستگی‌ها رو انجام می‌ده.**

اینجاست که @MockBean وارد می‌شه!

وقتی می‌خوایم یه **نسخه‌ی موک‌شده از Bean** رو داخل **Spring TestContext** جایگزین کنیم یا اضافه کنیم، از **@MockBean** استفاده می‌کنیم. با این کار، اسپرینگ می‌فهمه که این **Bean** باید موقع تست، به‌جای نسخه‌ی اصلی، یه نسخه‌ی **Mock شده** باشه.

نکته‌ی مهم!

از نظر **Stub کردن (Stubbing)** توی **Mockito**، هم **@Mock** و هم **@MockBean** دقیقاً یه جور کار می‌کنن. مشکل اصلی زمانی پیش میاد که **توی یه تست، هر دو رو باهم استفاده کنیم یا اشتباهی از یکی به‌جای اون یکی استفاده کنیم.**

اشتباه رایج دوم: استفاده‌ی بیش‌ازحد از @SpringBootTest

وقتی شروع به تست‌نویسی توی اسپرینگ بوت می‌کنیم، خیلی زود با **@SpringBootTest** روبه‌رو می‌شیم. حتی وقتی یه پروژه‌ی جدید از **start.spring.io** بسازیم، خودش یه تست اولیه با همین annotation ایجاد می‌کنه.

اما یه اشتباه بزرگ اینجاست!

خیلیا فکر می‌کنن که **@SpringBootTest** لازمه و باید توی همه‌ی تست‌ها استفاده بشه، ولی این درست نیست!

@SpringBootTest زمانی استفاده می‌شه که بخوایم یه تست یکپارچه (Integration Test) بنویسیم که با کل Spring Context کار کنه. وقتی از این annotation استفاده می‌کنیم، **Spring Test یه TestContext ایجاد می‌کنه که شامل همه‌ی Beanها (مثل @Configuration، @Service، @Component و ...) می‌شه.**

این یعنی اگه سرویس ما به یه **زیرساخت خارجی (مثل دیتابیس، صف (Message Queue) یا API دیگه‌ای) وابسته باشه،** باید اونارو هم توی تست فراهم کنیم. مثلاً اگه یه اپلیکیشن **CRUD** بنویسیم که به دیتابیس وصل می‌شه، بدون یه دیتابیس واقعی یا Mockشده، Repositoryهای ما اصلاً کار نمی‌کنن.

مشکل کجاست؟

اگه **تک‌تک تست‌هامون رو با @SpringBootTest اجرا کنیم، خیلی زود می‌فهمیم که تست‌ها بی‌دلیل کند شدن.** چرا؟ چون **لود کردن کامل Spring Context زمان‌بره!** به‌جای اینکه تست‌هامون سریع و سبک اجرا بشن، مجبور می‌شیم برای هر بار اجرا، کل اپلیکیشن رو بالا بیاریم!

پس راه‌حل چیه؟
به‌عنوان یه **قانون کلی**، بهتره تا جایی که ممکنه، تست‌هامون رو توی **سطح‌های پایین‌تر** انجام بدیم.

تست‌های واحد (Unit Tests):

- اگه قراره فقط یه `if` توی یه کلاس **@Service** رو تست کنیم، نیازی به لود کردن کل اسپرینگ بوت نداریم! یه تست واحد ساده با JUnit و Mockito کاملاً کافیه.

تست‌های سطح میانی (Slice Tests):

- مثلاً اگه بخوایم پیکربندی **Spring Security** رو تست کنیم، **@WebMvcTest** گزینه‌ی بهتریه، چون فقط لایه‌ی **وب** رو تست می‌کنه و نیازی به بالا آوردن کل اپلیکیشن نیست.

تست‌های یکپارچه (Integration Tests):

- **@SpringBootTest** فقط وقتی لازمه که بخوایم ارتباط بین چندین کامپوننت مختلف رو تست کنیم یا تست End-to-End انجام بدیم.

اشتباه رایج سوم: استفاده نکردن از کش (Cache) در Spring TestContext

این اشتباه به مورد قبلی، یعنی **استفاده‌ی بیش‌ازحد از @SpringBootTest**، مرتبط است. یکی از مشکلاتی که باعث **کند شدن تست‌ها** می‌شود، **لود کردن Spring TestContext از ابتدا برای هر تست** است. اما چرا باید هر بار یک TestContext جدید بسازیم، در حالی که می‌توان از یک TestContext قبلی استفاده کرد؟

Spring TestContext به صورت خودکار کش می‌شود و این ویژگی می‌تواند سرعت اجرای تست‌ها را به طور قابل توجهی افزایش دهد.

Spring TestContext چگونه کش می‌شود؟

زمانی که یک تست اجرا می‌شود و قرار است **Spring TestContext** راه‌اندازی شود (چه یک **Slice Test** باشد، چه کل **Application Context**)، اسپرینگ بررسی می‌کند که آیا یک TestContext مشابه از قبل وجود دارد یا خیر.

- اگر یک **TestContext** با همان پیکربندی قبلی کش شده باشد، همان را مجدداً استفاده می‌کند.
- اگر تست جدید نیاز به یک پیکربندی متفاوت داشته باشد، یک **TestContext** جدید ساخته می‌شود و پس از اجرا، کش می‌شود تا تست‌های بعدی از آن استفاده کنند.

چگونه از این قابلیت بیشترین بهره را ببریم؟

فرض کنید دو تست داریم:

- یک تست که **Profile** به نام `"integration-test"` را فعال می‌کند.
- یک تست دیگر که **Profile** به نام `"web-test"` را فعال می‌کند.

چون پیکربندی این دو تست متفاوت است، **Spring نمی‌تواند از یک TestContext مشترک برای آنها استفاده کند** و مجبور می‌شود هر بار یک TestContext جدید بسازد که این باعث افزایش زمان اجرای تست‌ها می‌شود.

عوامل مختلفی وجود دارد که تعیین می‌کنند آیا یک TestContext می‌تواند کش شود یا نه. برخی از این عوامل شامل **فعال بودن پروفایل‌ها، مقداردهی اولیه‌ی دیتابیس، موک شدن Beanها، مقداردهی متغیرهای محیطی و موارد دیگر** هستند.

روش‌های بهینه‌سازی برای افزایش سرعت تست‌ها

برای اینکه از **قابلیت کش شدن TestContext** به بهترین شکل استفاده کنیم:

- تا جای ممکن از یک پیکربندی مشترک برای تست‌های یکپارچه **(Integration Tests)** استفاده کنیم.
- از تعریف چندین پیکربندی متفاوت برای تست‌هایی که کل **ApplicationContext** را نیاز دارند، خودداری کنیم.