Improving Developer Experiences & Processes Using Quarkus

September 20, 2021

▶ Quarkus Insights – Episode #64

QUARKUS

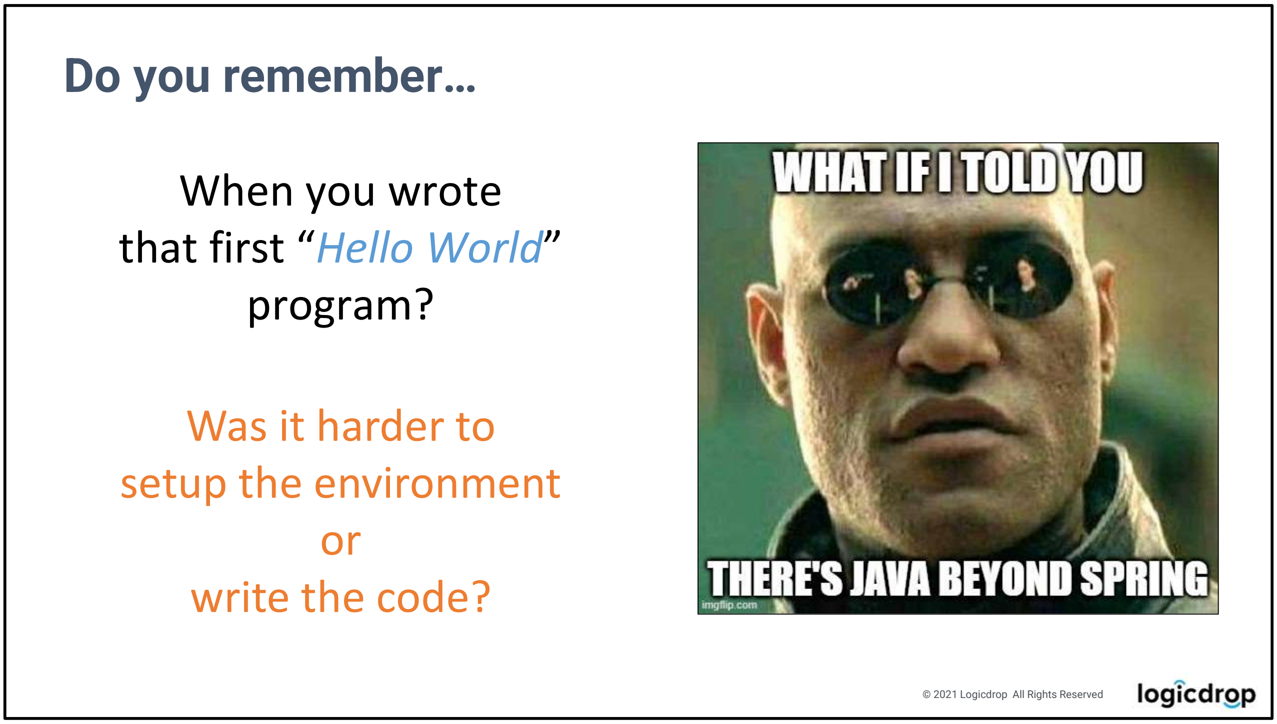
Topics

- Background
- Philosophy
- Improvements
- Making the Jump to Quarkus
- Marrying Quarkus with CI/CD

Our journey starts around Jan 1 2020 (https://groups.google.com/g/quarkus-dev/c/lr8EoVi0h4Y/m/U3ORlroxCwAJ).

It wasn't until about 5mo ago we made the jump – so we did spend ample time experimenting and did not do this without significant debate and validation.

The "point of no return" now – we are fully vested in Quarkus and not looking back.



Do you remember…

When you wrote
that first "*Hello World*"
program?

Was it harder to
setup the environment
or
write the code?

logicdrop

Eat our own dogfood, we make a living simplifying for others what is complex.

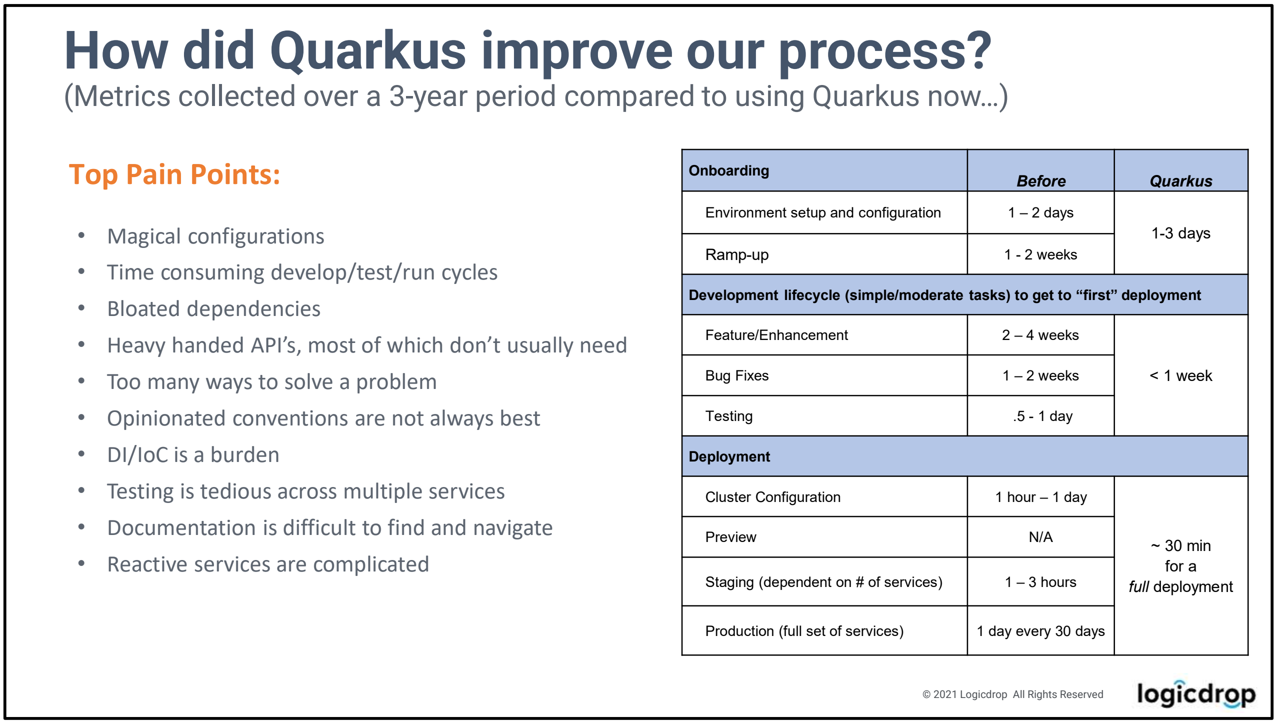
This is different than no-code or low-code. We don't believe things needs to be "dumbed" down instead allow developers to focus on the problem at hand rather than be bogged down by everything else (builds, K8S, testing, mocking, API's). This makes it easier to train, more understandable, easier to share knowledge across the team.

Why do people gravitate towards other languages like Python, Node, etc.? There is faster gratification starting up than with Java historically.

Very easy to introduce Quarkus to both novice and experienced developers.

Back in college, where did kids struggle – the setup not writing "hello world" (we actually took a poll of this)

Quarkus is easier to start and build upon than other frameworks



How did Quarkus improve our process?

(Metrics collected over a 3-year period compared to using Quarkus now…)

Top Pain Points:

- Magical configurations
- Time consuming develop/test/run cycles
- Bloated dependencies
- Heavy handed API's, most of which don't usually need
- Too many ways to solve a problem
- Opinionated conventions are not always best
- DI/IoC is a burden
- Testing is tedious across multiple services
- Documentation is difficult to find and navigate
- Reactive services are complicated

Onboarding	***Before***	***Quarkus***
Environment setup and configuration	1 – 2 days	1-3 days
Ramp-up	1 - 2 weeks	
Development lifecycle (simple/moderate tasks) to get to "first" deployment		
Feature/Enhancement	2 – 4 weeks	< 1 week
Bug Fixes	1 – 2 weeks	
Testing	.5 - 1 day	
Deployment		
Cluster Configuration	1 hour – 1 day	~ 30 min for a *full* deployment
Preview	N/A	
Staging (dependent on # of services)	1 – 3 hours	
Production (full set of services)	1 day every 30 days	

- First deployment is not even usually feature complete – that is the time it takes to go from a "ticket" to the first deployed iteration in its entirety – then it is iterated on till complete.

- For bug fixes, this is a high evaluation of the time it takes to complete the issue. We see times usually ~1-2 days to resolve a majority of issues so far

- AutoConfiguration, and the increased # of useless jars (only to solve Maven shortcomings with hacks like "optional" dependencies) has been the biggest issue.
- Because it was indiscriminate in loading jars the ordering of properties and overlaying them became problematic – this was a big win for us in Quarkus.

- The DevTools in Spring were just never stable enough, especially across large codebases with many modules.

- Project Reactor is a nice API but only a few could become adept at it and most struggled with it at one point or another more frequently than not.

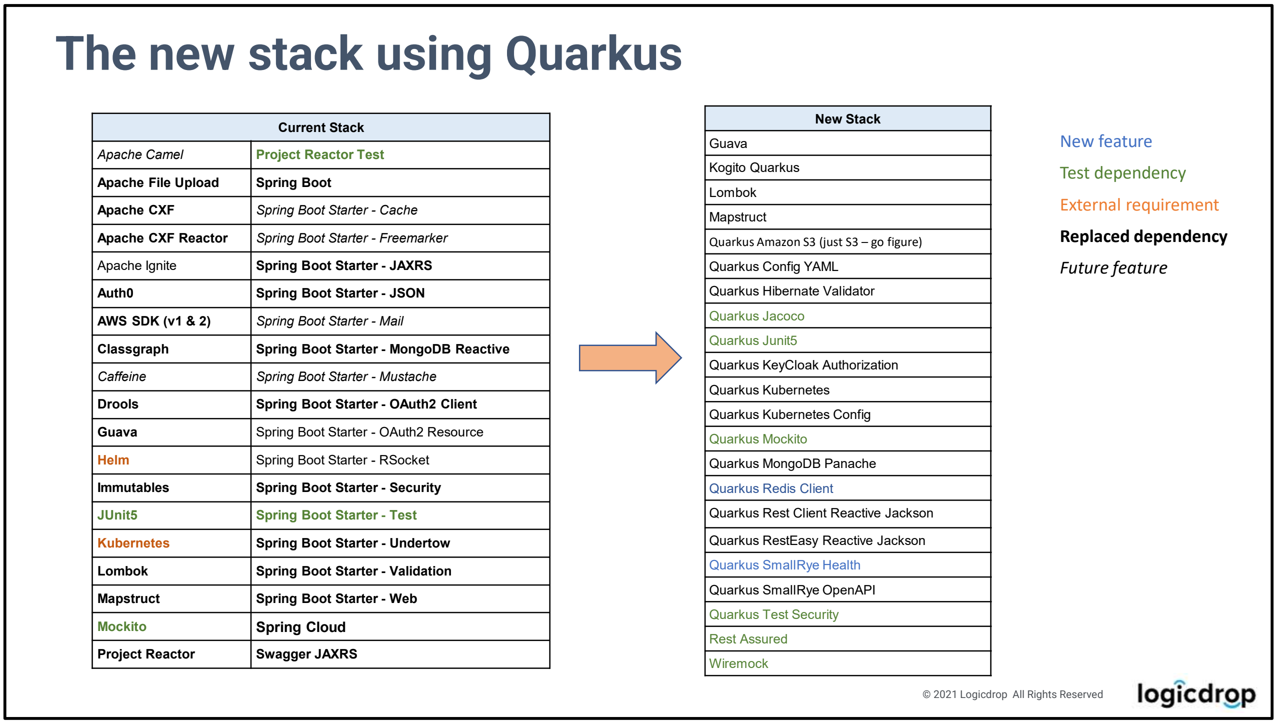
- Spring security is a great example of a million ways to solve the problem. This was the clencher for us with Quarkus when with a few line of code we switched between Keycloak and Auth0.

- It was impossible not to get intertwined with Spring and it numerous APIs.

- Documentation was difficult to navigate and/or find. Support was very limited to SO or Gittr which was hit or miss.

- Digging into the code was often times the best way to understand and debug a problem.

- Magical configurations burned us more than once. Things that worked, and were deployed, sometimes would seem to stop working for no reason or change.



The new stack using Quarkus

Current Stack	
Apache Camel	**Project Reactor Test**
Apache File Upload	**Spring Boot**
Apache CXF	*Spring Boot Starter - Cache*
Apache CXF Reactor	*Spring Boot Starter - Freemarker*
Apache Ignite	**Spring Boot Starter - JAXRS**
Auth0	**Spring Boot Starter - JSON**
AWS SDK (v1 & 2)	*Spring Boot Starter - Mail*
Classgraph	**Spring Boot Starter - MongoDB Reactive**
Caffeine	*Spring Boot Starter - Mustache*
Drools	**Spring Boot Starter - OAuth2 Client**
Guava	Spring Boot Starter - OAuth2 Resource
Helm	Spring Boot Starter - RSocket
Immutables	**Spring Boot Starter - Security**
JUnit5	**Spring Boot Starter - Test**
Kubernetes	**Spring Boot Starter - Undertow**
Lombok	**Spring Boot Starter - Validation**
Mapstruct	**Spring Boot Starter - Web**
Mockito	**Spring Cloud**
Project Reactor	**Swagger JAXRS**

New Stack
Guava
Kogito Quarkus
Lombok
Mapstruct
Quarkus Amazon S3 (just S3 – go figure)
Quarkus Config YAML
Quarkus Hibernate Validator
Quarkus Jacoco
Quarkus Junit5
Quarkus KeyCloak Authorization
Quarkus Kubernetes
Quarkus Kubernetes Config
Quarkus Mockito
Quarkus MongoDB Panache
Quarkus Redis Client
Quarkus Rest Client Reactive Jackson
Quarkus RestEasy Reactive Jackson
Quarkus SmallRye Health
Quarkus SmallRye OpenAPI
Quarkus Test Security
Rest Assured
Wiremock

New feature

Test dependency

External requirement

Replaced dependency

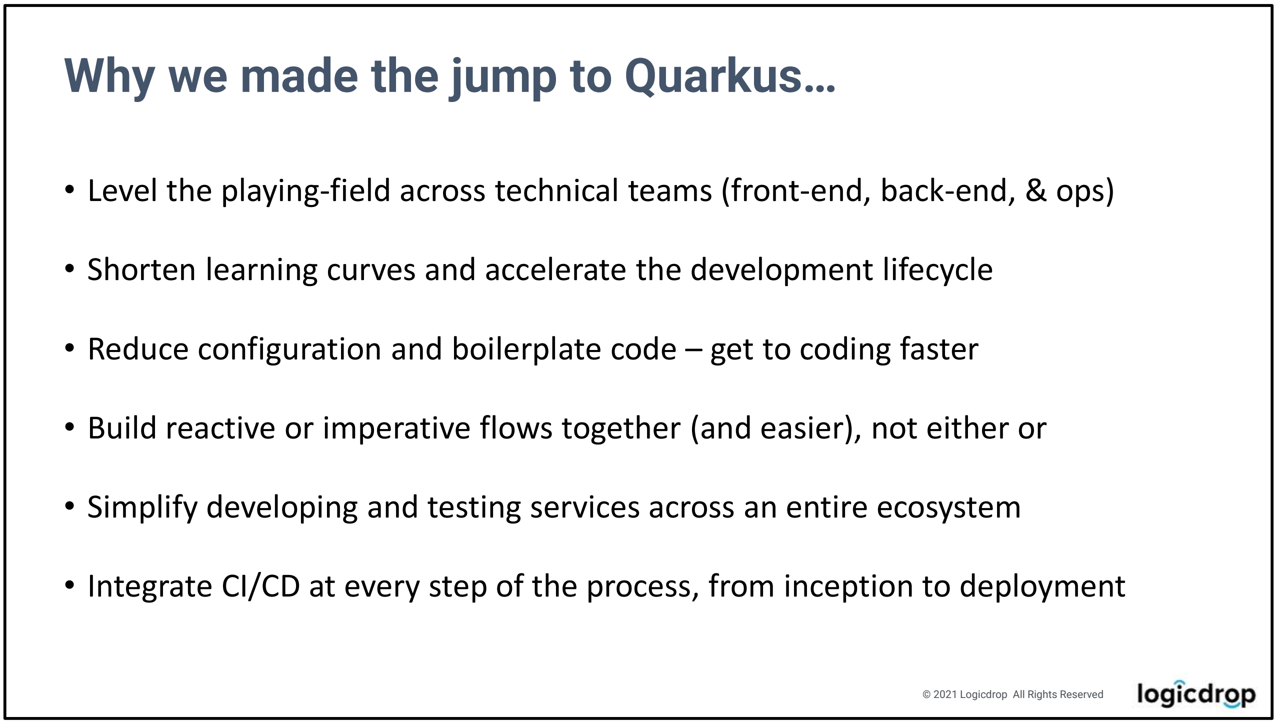
Future feature

- Bolded items have a direct correlation to the new stack. They were needed in the previous platform where as Quarkus provides that functionality/APIs through a smaller set of dependencies (and more understandable)

- In some case, there were equivalent Spring Boot Starters but, usually if we did not use Spring (or tried not to), we pulled them in even if they may have been transitive somewhere else (Reactor and Spring MongoDB Reactive)

- This does not even take into account transitive dependencies or “alternatives” such as VertX/Netty vs. Undertow

- Quarkus also replaced the need for tooling outside the platform such as Kubernetes and Helm.



Why we made the jump to Quarkus…

- Level the playing-field across technical teams (front-end, back-end, & ops)
- Shorten learning curves and accelerate the development lifecycle
- Reduce configuration and boilerplate code – get to coding faster
- Build reactive or imperative flows together (and easier), not either or
- Simplify developing and testing services across an entire ecosystem
- Integrate CI/CD at every step of the process, from inception to deployment

logicdrop

- Level the playing field
- Show how we standardized on VSCode and why
- We wanted a way to onboard that was consolidated to one place and easy to use. This replaced the variety of scripts, containers and utilities.

- Reduce learning curves
- Quarkus allowed us to train developers quickly with one API
- Quarkus allowed us to “skip” explaining Maven and having profiles for most developers
- Quarkus allowed us to “skip” explaining what was needed for K8S
- Documentation was easy to find and follow. Examples were usually relevant enough to provide a working understanding
- Where is code mostly working? Is it mostly resources, consolidate those to one repo because 70% of the work happens there.

- Reduce configuration and boilerplate
- Overall, using Quarkus, gave us a standard-based, consistent, and easy way to write code – DI works as it should and is not magical
- Got rid of the complexity of multiple AutoConfigurations which were magical,

fragile, confusing, and wasted artifacts within the process (jars that did nothing). Ordering also become very important as the configurations grew between different teams.

- Using Quarkus a lot of the "support" code written within the platform went away. No more reflection to scan all beans and find a set for something.
- Using things like the RESTTemplate, MongoTemplate, etc. usually ended up being too heavy and restrictive for common scenarios, we end up bending these to our will. Quarkus just did it a standard and simple way.
- We found ourselves building abstraction on top of Spring abstractions to better do what we needed. With Quarkus this almost all went away.
- Using Quarkus we were able to easily get rid of Helm charts specific to each service and automate it in such a way developer needed no knowledge of it.
- The use of "optional" all over the place in POMs for configuration – this was a nightmare.

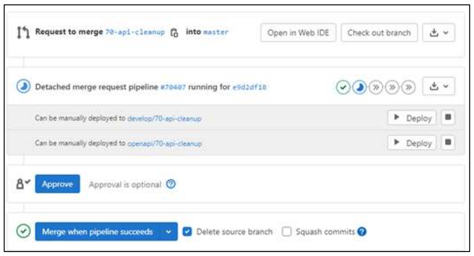
- Reactive Service
- One of the strong points of using Quarkus was the ability to write reactive and non-reactive code. This was confusing in Spring because it was more one way or the other.
- Project Reactor was a robust API but it was difficult for even the most seasoned developer to learn effectively.
- Mutiny as an API, we were apprehensive at first to use it because it was another different API, has been amazing. This has greatly reduced the complexity of writing rx streams.
- Because we did not want to do it the Spring way for rx we integrated CXF as our base to be similar to what Quarkus does out-of-the-box with Undertow/worker threads
- The ability to do imperative and reactive together was a big one for us. Reality is, a lot of API (at least at the time or we use) were not reactive yet so this made the transition easier. Our approach was to use CXF.

- Service-based development is difficult
- Quarkus allowed us to better structure code so that it was easier to test 1 or more services at the same time without a lot of container orchestration
- Combing Wiremock we were able to completely disconnect services from the core (i.e. not require the service to be running) and test them stand-alone – this was big for our developers
- With everything surrounding a service, writing the code was the easier part, Quarkus made it easier for us to get rid of barriers requiring additional skillsets and focus on code
- Quarkus:dev has made developing/test code, even across multiple services, much less painful.

- The combination of everything using Quarkus has made our development process more streamlined and easier to work with
- Testing and quality of tests is much better because it is more straight forward and easier to use.

Build, Quarkus, build…

1. Start with a Merge-Request…


Request to merge 70-api-cleanup into master

Detached merge request pipeline #70407 running for e9d2df18

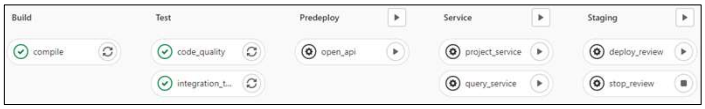
Can be manually deployed to develop/70-api-cleanup

Can be manually deployed to openapi/70-api-cleanup

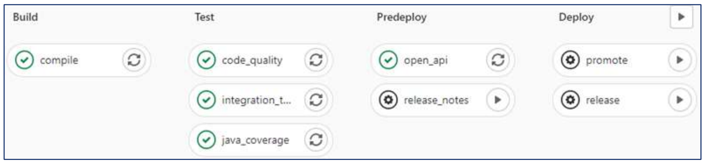
Approval is optional

☑ Delete source branch □ Squash commits

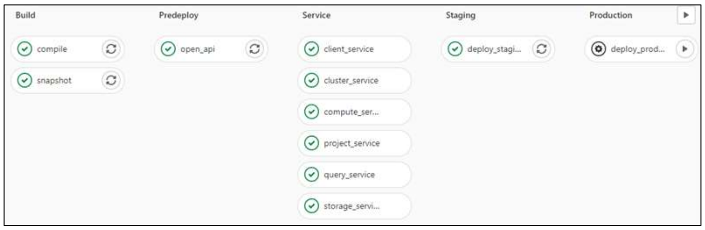
2. Test and deploy only services you changed in isolation…



3. Merge and promote…



4. Deploy



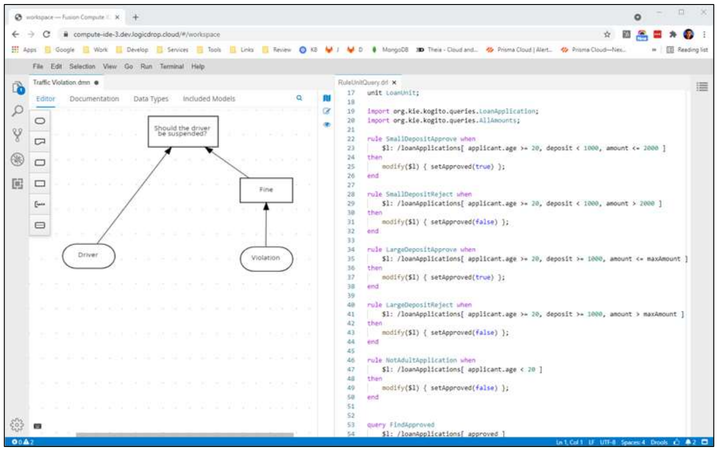
…with a single click

1. A ticket is opened, create a branch and an MR to work on. Only services you change will be deployable and they will automatically be tied to whatever you don’t deploy in staging.
2. Merge the ticket into master. Choose to promote (prepare) or release immediately. All versioning is orchestrated automatically.
3. Promote/release and deploy individual services or, usually, the whole suite of services to staging then production.

- Versioning is automatically handled. Tags can dictate if it is a major, minor, or patch increment automatically.
- Native images and containers are created/tested during promotion and available at any time to deploy or rollback to.
- Kubernetes manifests are held on to so that deployments can happen without having to repeat the process (mostly the native builds from scratch)

The CI/CD process, within each service will build the appropriate Native image, manifests, and container ready for deployment now or later.

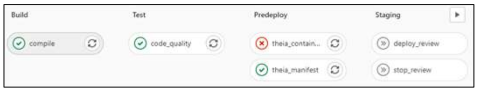
First one to market wins…
and with Quarkus now, I can deploy all day long…


workspace — Fusion Compute
compute-ide-3.dev.logicdrop.cloud/#/workspace
File Edit Selection View Go Run Terminal Help
Traffic Violation.dmn
Editor
Documentation
Data Types
Included Models
Should the driver be suspended?
Fine
Driver
Violation
RuleUnitQuery.drl
Ln 1, Col 1 LF UTF-8 Spaces: 4 Drools

```
unit LoanUnit;

import org.kie.kogito.queries.LoanApplication;
import org.kie.kogito.queries.AllAmounts;

rule SmallDepositApprove when
    $l: /loanApplications[ applicant.age >= 20, deposit < 1000, amount <= 2000 ]
then
    modify($l) { setApproved(true) };
end

rule SmallDepositReject when
    $l: /loanApplications[ applicant.age >= 20, deposit < 1000, amount > 2000 ]
then
    modify($l) { setApproved(false) };
end

rule LargeDepositApprove when
    $l: /loanApplications[ applicant.age >= 20, deposit >= 1000, amount <= maxAmount ]
then
    modify($l) { setApproved(true) };
end

rule LargeDepositReject when
    $l: /loanApplications[ applicant.age >= 20, deposit >= 1000, amount > maxAmount ]
then
    modify($l) { setApproved(false) };
end

rule NotAdultApplication when
    $l: /loanApplications[ applicant.age < 20 ]
then
    modify($l) { setApproved(false) };
end


query FindApproved
    $l: /loanApplications[ approved ]
```

This prototype was created and deploying in less than a week…

Unless the build fails (boo)…



But then I just fix it and deploy again…

logicdrop