What have the @nnotations done to us?

@adamwarski

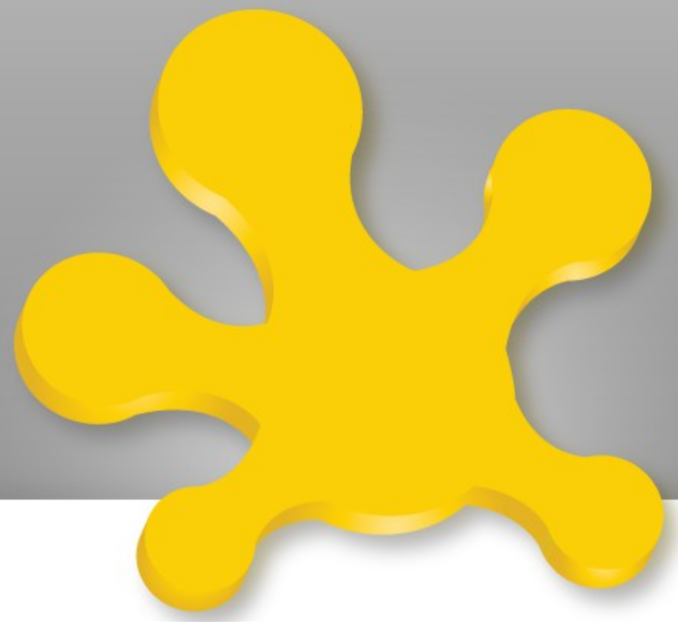
geecon

SOFTWAREMILL

Kraków, 9-11 May 2018

Adam Warski

Static analysis using JSR308 annotations



Let's move the Java world!

GeeCON 2009

PART 1
ABOUT ANNOTATIONS

@NNOTATIONS

- Introduced to Java in 2004
- Replaced xml programming

JAKARTA EE

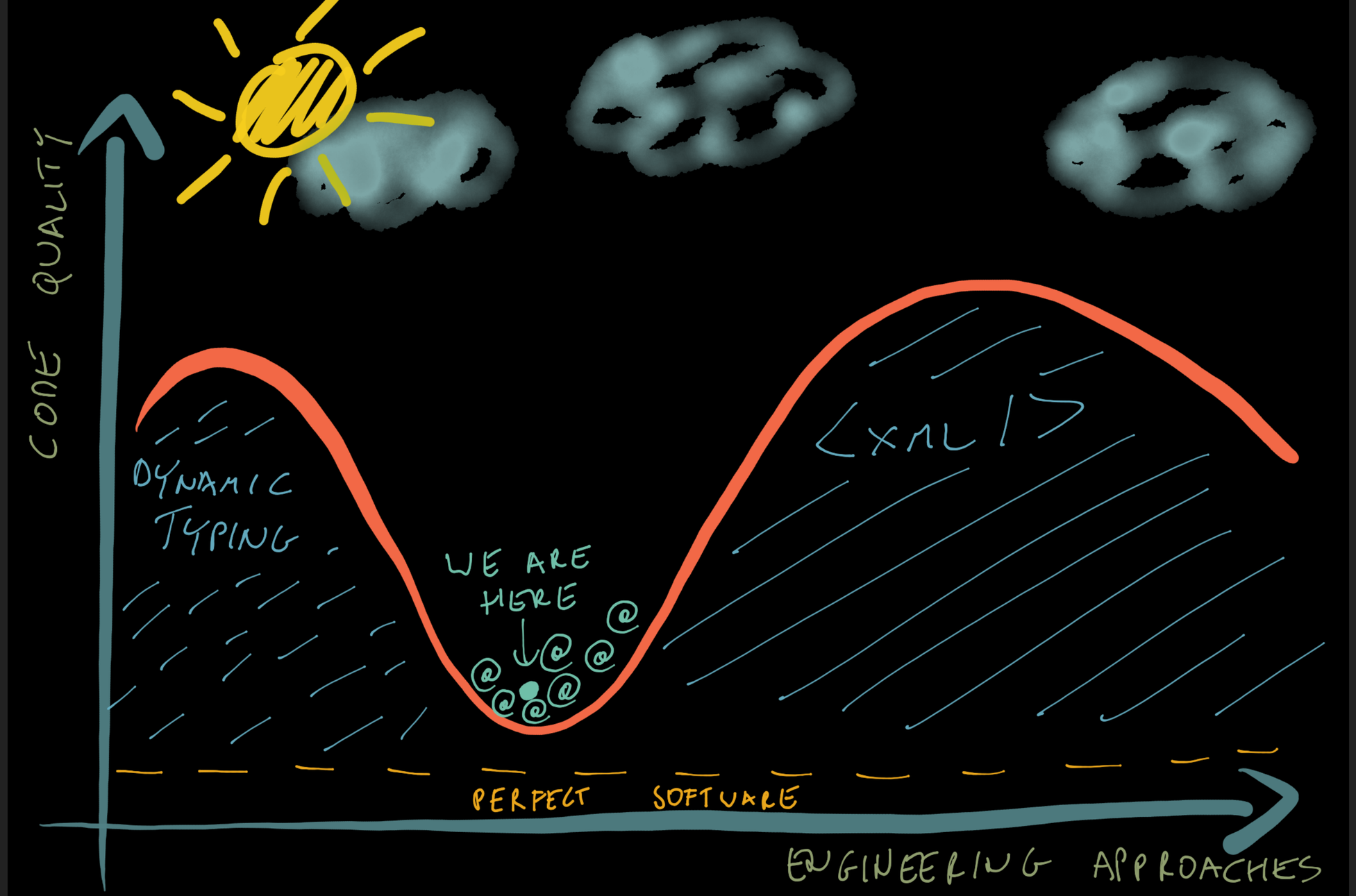
WHY @, IN THE FIRST PLACE?

- We need a way to express meta-data
- Describe classes, methods, fields (`@Entity`, `@JsonProperty`)
- Cross-cutting concerns (`@Secure`, `@Transactional`)
- Orchestrate the application (`@Inject`, `@EnableWebMvc`)

WHY @, IN THE FIRST PLACE?

- Easy to introduce, non-invasive
- Clearly separated
- Close to referenced elements
- Can be inspected statically & at run-time

MOST POPULAR != BEST



AS GOOD AS IT GETS IN JAVA <= 7

AN EMBEDDED MINI LANGUAGE, INTERPRETED AT RUN-TIME

What @ really are?

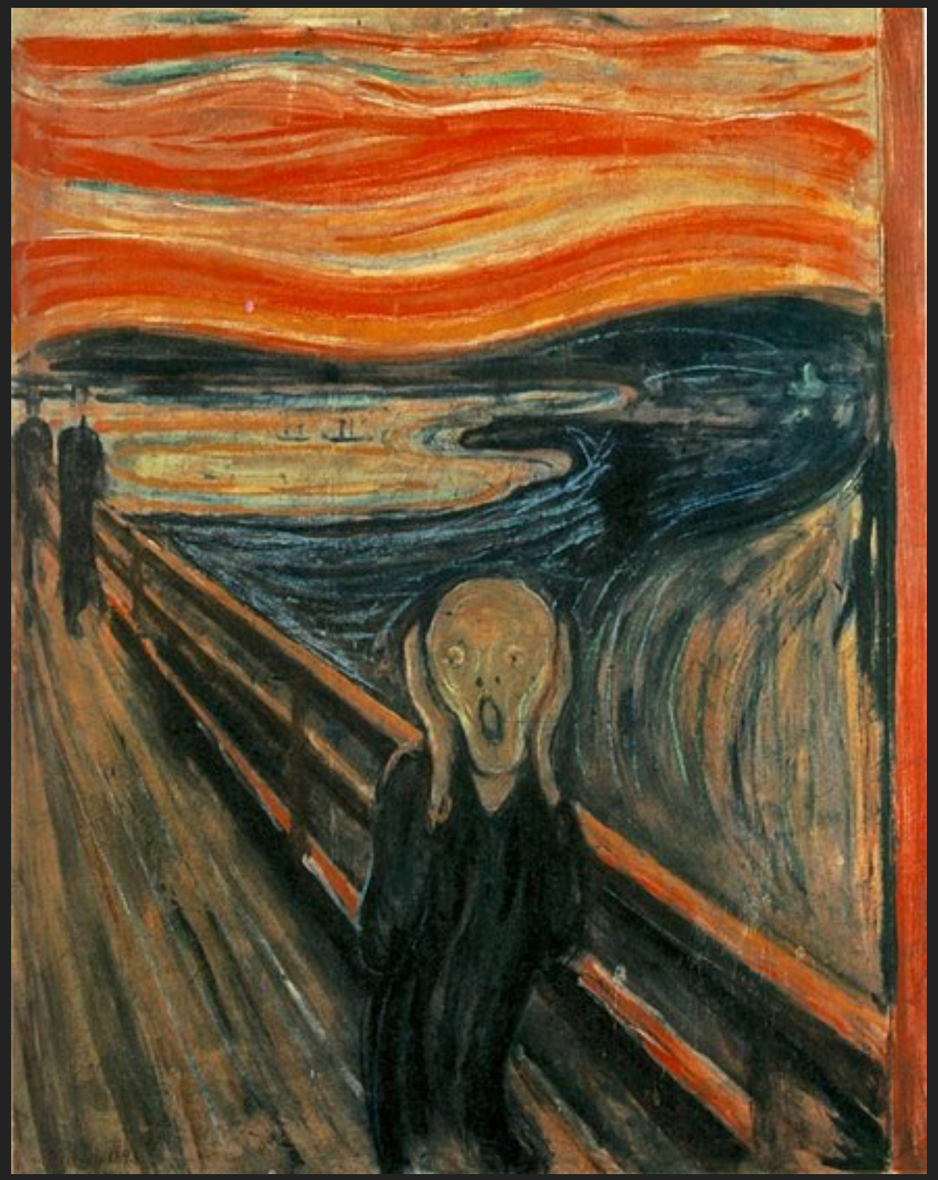
ANNOTATION INTERPRETERS; A RUN-TIME FOR THE DYNAMIC LANGUAGE

What are containers?


WE PROGRAM THE CONTAINERS USING ANNOTATIONS
What do we do?

PART 2
WHAT @ HAVE BECOME?
WHAT PATTERNS HAVE EMERGED?

```
@Functional
@Blockchain
@MachineLearning
@MicroService
@PleaseWork
public class DoesNotMatter {}
```



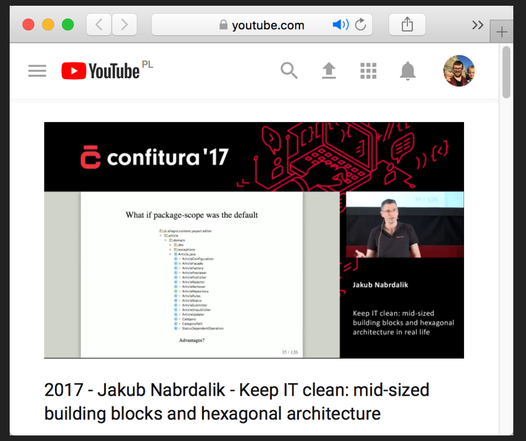
Fear of `new ...()`

```java
@Component
class PetRepository {
    @Autowired
    PetRepository(Database database) {}
}

@Component
class PetFormatter {
    @Autowired
    PetFormatter(PetRepository pets) { }
}

@Component
class PetValidator {}

@Controller
class PetController {
    @Autowired
    PetController(PetValidator petValidator, PetFormatter petFormatter) {}
}

@SpringBootApplication
class PetClinicApplication {
    public static void main(String[] args) {
        SpringApplication.run(PetClinicApplication.class, args);
    }
}
```

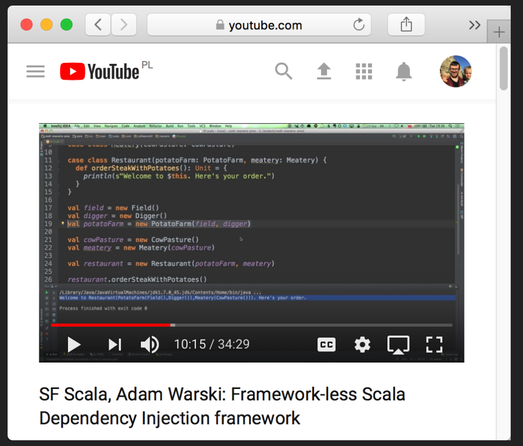

```java
interface Database {}

class PetRepository {
    PetRepository(Database database) {}
}

class PetFormatter {
    PetFormatter(PetRepository pets) { }
}

class PetValidator {}

class PetController {
    PetController(PetValidator petValidator, PetFormatter petFormatter) {}
}

class PetClinicApplication {
    public static void main(String[] args) {
        var database = new Database() {};
        var petRepository = new PetRepository(database);
        var petFormatter = new PetFormatter(petRepository);
        var petValidator = new PetValidator();
        var petController = new PetController(petValidator, petFormatter);
        Http.start(petController,  i: "localhost",  p: 8080);
    }
}
```

MANUAL DEPENDENCY INJECTION

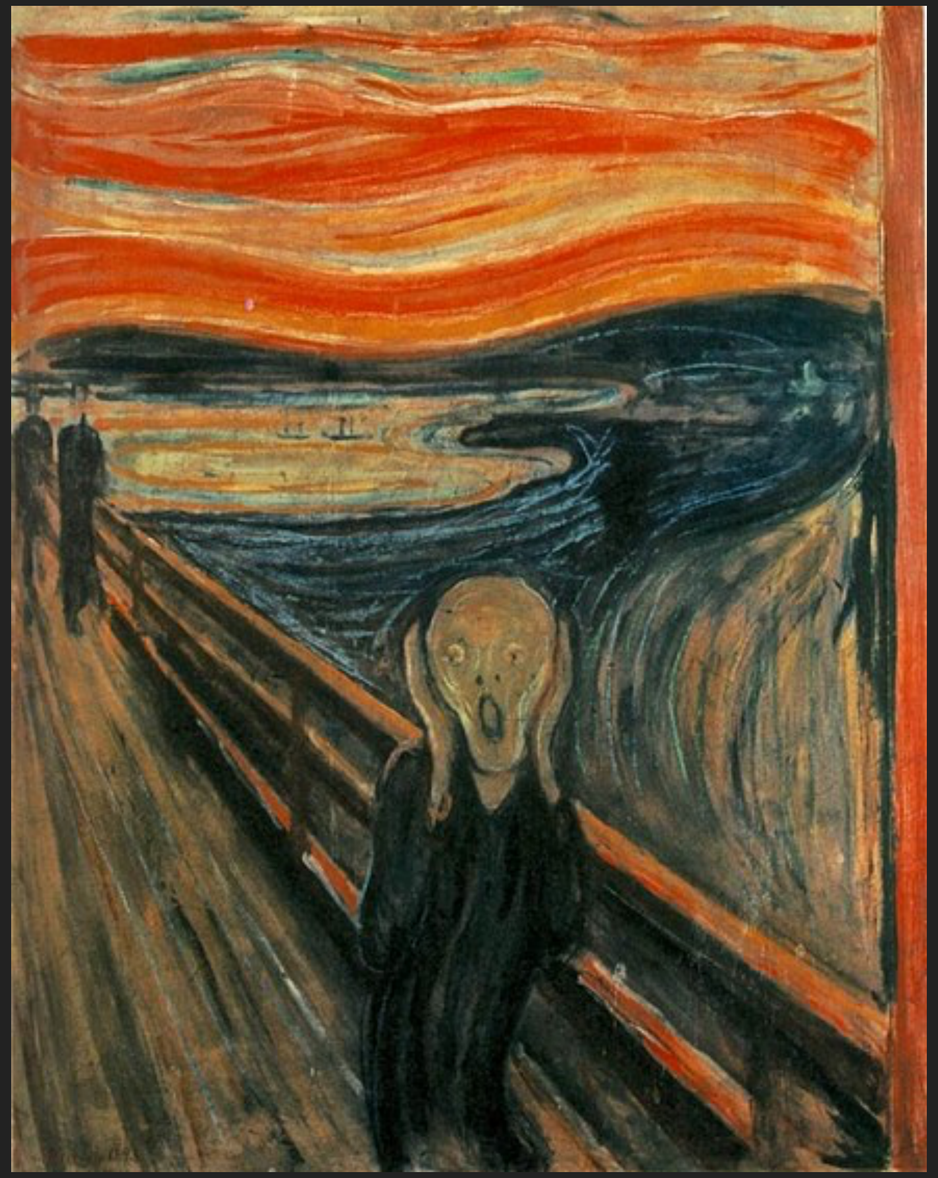
- Split object graph creation
  - e.g. per-package, functionality
  - package scope
- Create the objects how you want
- Reader vs writer convenience



2017 - Jakub Nabrdalik - Keep IT clean: mid-sized building blocks and hexagonal architecture



SF Scala, Adam Warski: Framework-less Scala Dependency Injection framework



Fear of `public static void main()`

main[1]

[meyn]

Spell Syllables

Synonyms Examples Word Origin

See more synonyms on Thesaurus.com

adjective

1. chief in size, extent, or importance; principal; leading:
   the company's main office; the main features of a plan.

STARTUP SEQUENCE

- Do you know how your JavaEE/Spring application starts?
- What happens and in what order?
- Reader vs writer convenience

```
public class MainTest {
    public static void main(String[] args) {
        var petControllerModule = new PetControllerModule();

        var http = new Http();
        var serviceRegistry = new ServiceRegistry("http://services.local");

        var boundHttp = http.bind(
            petControllerModule.getPetController(),
            i: "localhost", p: 8080);
        var registeredService = serviceRegistry.register( name: "pets", boundHttp);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            registeredService.deregister();
            boundHttp.unbind();
        }));
    }
}
```

CLASSPATH SCANNING

- a general mechanism to avoid `main` & `new`
- add a jar & it magically works
- very convenient for rapid bootstrapping

TRADE CERTAINTY & CONTROL

for

FAST & CONVENIENT BOOTSTRAP

EXPLORABILITY

Code should be easy to read
Easy to navigate

Understand what services are used, how and when
What's the ordering

Go-to-definition: best method to learn

META-DATA MAPPING

- Entities
- JSON
- HTTP endpoints
- ...

- **Describe** classes, methods and fields

TRANSFORMING A JAX-RS MAPPING

```java
@Path("/hello/{user}")
class Hello {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String helloWorld(@PathParam("user") String user) {
        return "Hello World, " + user;
    }
}
```

THE GOOD

- Meta-data separated from the business logic
- We can test the logic without the HTTP layer
- Readable code
- Automatically generate Swagger docs

THE BAD

- Is this the right combination of annotations? Maybe something is missing?
- Where is the endpoint exposed?
- What are all the endpoints exposed at a given path?
- Are the JAX-RS annotations tested?
- Stringly-typed parameter references

WHAT IF ...

- The endpoint is represented as a **Java value**
- Separation & testability of business logic maintained
- The **description** of the endpoint is also testable
- Basing on `Endpoint` values, Swagger docs could be generated
- Programmatically define endpoints

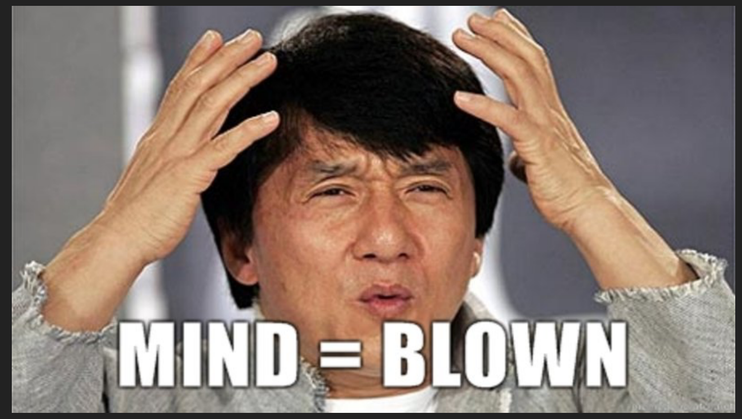
```
class Hello {
    String helloWorld(String user) {
        return "Hello World, " + user;
    }
}

class HelloEndpoints {
    Endpoint helloWorldEndpoint(Hello hello) {
        return Endpoint
            .withPath(root().segment("hello").captureSegment())
            .method(GET)
            .produces(MediaType.TEXT_PLAIN)
            .invoke(hello::helloWorld);
    }

    List<Endpoint> endpoints(Hello hello) {
        return Arrays.asList(helloWorldEndpoint(hello) /* ... */);
    }
}
```

JUST LOOKING AT THE CODE, IT'S CRYSTAL CLEAR WHAT'S HAPPENING AND WHEN.

```
class MyApplicationBootstrap {
    public static void main(String[] args) {
        // 1. create the object graph.
        var hello = new Hello();

        // 2. create a list of all endpoints our application will expose
        List<Endpoint> endpoints = new ArrayList<>();
        endpoints.addAll(new HelloEndpoints().endpoints(hello));
        // endpoints.addAll(...);

        // 3. start a web server
        Http.start(endpoints,  i: "localhost",  p: 8080);
    }
}
```

JUST USE JAVA! (OR A MORE ADVANCED JVM LANGUAGE OF YOUR CHOICE)

- Meta-data becomes **first-class values**
- A single language for code and meta-data
- Can be generated using:
  - *loops*
  - *conditionals*
  - *helper methods*



STATIC vs DYNAMIC
IMAGINE THAT ...

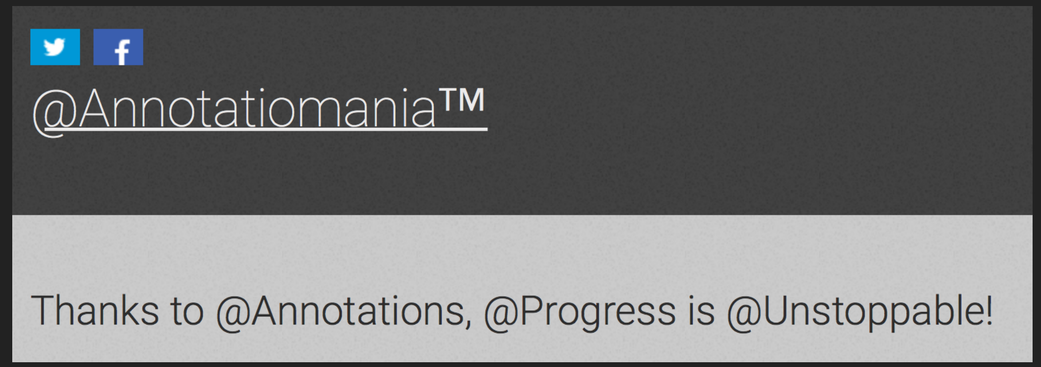
- You don't have to copy ----------> to all entities

```
@TableGenerator(
  name="tab",
  initialValue=0,
  allocationSize=50)
@GeneratedValue(
  strategy=GenerationType.TABLE,
  generator="tab")
@Id
```

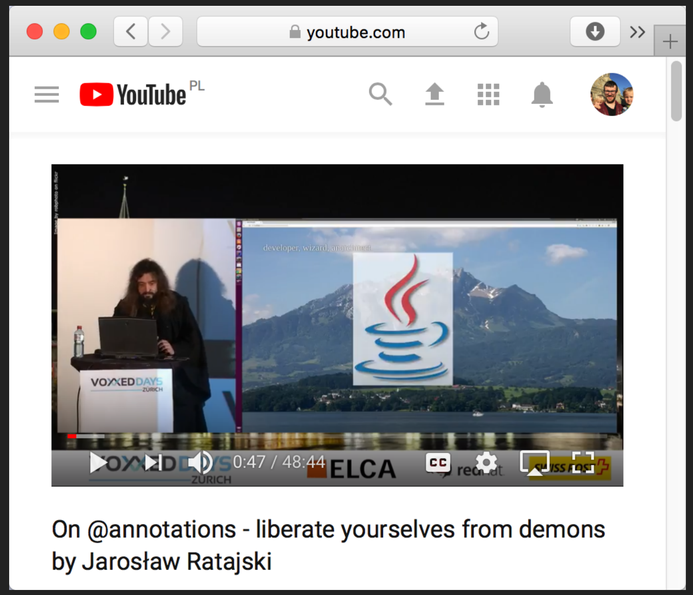
- Instead, loop over **descriptions**

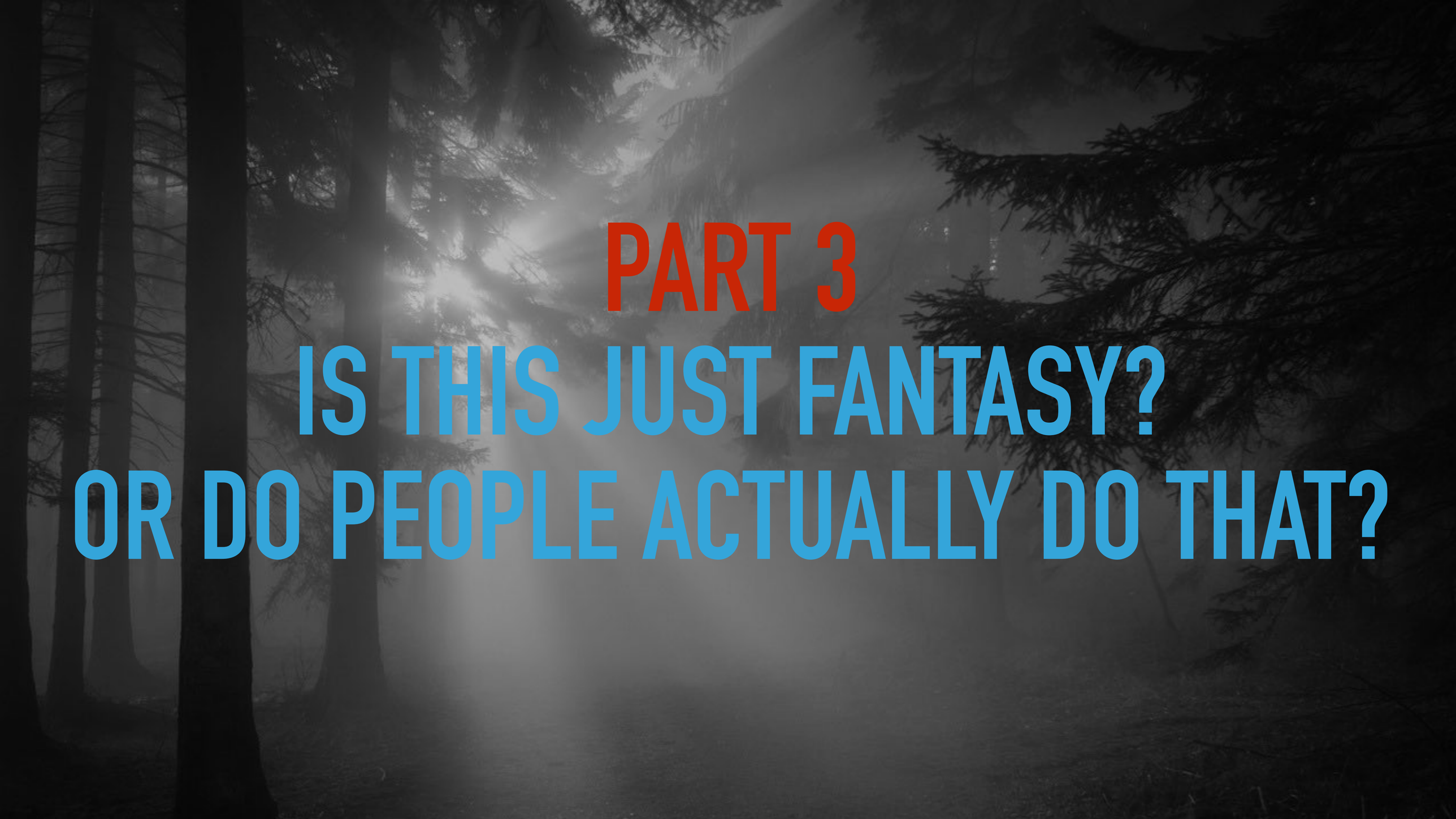
IF THAT'S NOT ENOUGH ...

- http://annotatiomania.com



- Jarek Ratajski's presentation on the same subject
  - composability: `@Retry + @Transactional`?




PART 3
IS THIS JUST FANTASY?
OR DO PEOPLE ACTUALLY DO THAT?

SPARK (SPARKJAVA.COM)

Quick start

Java

```
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

Route patterns can include named parameters, accessible via the `params()` method on the request object:

```
// matches "GET /hello/foo" and "GET /hello/bar"
// request.params(":name") is 'foo' or 'bar'
get("/hello/:name", (request, response) -> {
    return "Hello: " + request.params(":name");
});
```

Route patterns can also include splat (or wildcard) parameters. These parameters can be accessed by using the `splat()` method on the request object:

```
// matches "GET /say/hello/to/world"
// request.splat()[0] is 'hello' and request.splat()[1] 'world'
get("/say/*/to/*", (request, response) -> {
    return "Number of splat parameters: " + request.splat().length;
});
```

Path groups

If you have a lot of routes, it can be helpful to separate them into groups. This can be done by calling the `path()` method, which takes a `String prefix` and gives you a scope to declare routes and filters (or nested paths) in:

```
path("/api", () -> {
    before("/*", (q, a) -> log.info("Received api call"));
    path("/email", () -> {
        post("/add",       EmailApi.addEmail);
        put("/change",     EmailApi.changeEmail);
        delete("/remove",  EmailApi.deleteEmail);
    });
    path("/username", () -> {
        post("/add",       UserApi.addUsername);
        put("/change",     UserApi.changeUsername);
        delete("/remove",  UserApi.deleteUsername);
    });
});
```

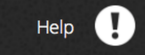
JOOQ (JOOQ.ORG)

Let's add a simple query constructed with jOOQ's query DSL:

```
DSLContext create = DSL.using(conn, SQLDialect.MYSQL);
Result<Record> result = create.select().from(AUTHOR).fetch();
```

```
create.transaction(configuration -> {
    AuthorRecord author =
    DSL.using(configuration)
       .insertInto(AUTHOR, AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
       .values("George", "Orwell")
       .returning()
       .fetchOne();

    DSL.using(configuration)
       .insertInto(BOOK, BOOK.AUTHOR_ID, BOOK.TITLE)
       .values(author.getId(), "1984")
       .values(author.getId(), "Animal Farm")
       .execute();

    // Implicit commit executed here
});
```

```
AuthorRecord author =
DSL.using(configuration) // This configuration will be attached to any record produced
by the below query.
   .selectFrom(AUTHOR)
   .where(AUTHOR.ID.eq(1))
   .fetchOne();

author.setLastName("Smith");
author.store(); // This store call operates on the "attached" configuration.
```



SPRING 5 (FUNCTIONAL WEB FRAMEWORK)

```
RouterFunction<?> route = route(GET("/person/{id}"),
  request -> {
    Mono<Person> person = Mono.justOrEmpty(request.pathVariable("id"))
      .map(Integer::valueOf)
      .then(repository::getPerson);
    return Response.ok().body(fromPublisher(person, Person.class));
  })
  .and(route(GET("/person"),
    request -> {
      Flux<Person> people = repository.allPeople();
      return Response.ok().body(fromPublisher(people, Person.class));
    }))
  .and(route(POST("/person"),
    request -> {
      Mono<Person> person = request.body(toMono(Person.class));
      return Response.ok().build(repository.savePerson(person));
    }));
```

```
RouterFunction<?> route =
  route(GET("/hello-world"), handler::helloWorld)
  .and(route(GET("/the-answer"), handler::theAnswer))
  .filter((request, next) -> {
    System.out.println("Before handler invocation: " + request.path());
    Response<?> response = next.handle(request);
    Object body = response.body();
    System.out.println("After handler invocation: " + body);
    return response;
  });
```

```
HttpHandler httpHandler = RouterFunctions.toHttpHandler(route);
ReactorHttpHandlerAdapter adapter =
  new ReactorHttpHandlerAdapter(httpHandler);
HttpServer server = HttpServer.create("localhost", 8080);
server.startAndAwait(adapter);
```

OTHERS?

- Scala
  - macros & implicits

- Ceylon
  - type-safe metamodel

PART 4
SUMMING UP

- Greats docs & integrations
- Learnable
  - just as a separate language
  - "Spring VM"
  - teams can be efficient
- Local optimum
  - As good as (pre-λ) Java gets
- Evolves in the same direction

SUMMARY

- Containers are interpreters for a "not very typesafe" language
  - Instead: use a proper language, e.g. Java
  - or Scala/Kotlin/Ceylon/...
- Don't fear! `main()` and `new` are **OK**

SUMMARY

- Writing a bit more code is **OK**, if that makes you:
  - retain control
  - more certain of what code does
  - increase explorability

- Meta-data, **descriptions**, as first-class values


BLOG.SOFTWAREMILL.COM

THANK YOU!

@adamwarski
adam.warski@softwaremill.com