Featured in Development

Big Data Analytics with Spark Book Review and Interview

Big Data Analytics with Spark book, authored by Mohammed Guller, provides a practical guide for learning Apache Spark framework for different types of big-data analytics projects, including batch, interactive, graph, and stream data analysis as well as machine learning. InfoQ spoke with author about the book & development tools for big data applications.

All in Development

- Architecture & Design
  - Architecture
  - Enterprise Architecture
  - Scalability/Performance
  - Design
  - Case Studies
  - Microservices
  - Patterns
  - Security

Featured in Architecture & Design

Big Data Analytics with Spark Book Review and Interview

Big Data Analytics with Spark book, authored by Mohammed Guller, provides a practical guide for learning Apache Spark framework for different types of big-data analytics projects, including batch, interactive, graph, and stream data analysis as well as machine learning. InfoQ spoke with author about the book & development tools for big data applications.

All in Architecture & Design

- Data Science
  - Big Data
  - Machine Learning
  - NoSQL
  - Database
  - Data Analytics
  - Streaming

Featured in Data Science

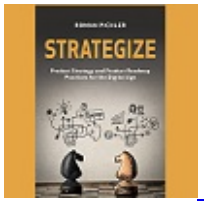
Big Data Analytics with Spark Book Review and Interview

Big Data Analytics with Spark book, authored by Mohammed Guller, provides a practical guide for learning Apache Spark framework for different types of big-data analytics projects, including batch, interactive, graph, and stream data analysis as well as machine learning. InfoQ spoke with author about the book & development tools for big data applications.

All in Data Science
- Culture & Methods
  - Agile
  - Leadership
  - Team Collaboration
  - Testing
  - Project Management
  - UX
  - Scrum
  - Lean/Kanban
  - Personal Growth

Featured in Culture & Methods

Q&A with Roman Pichler about Strategize



The book Strategize by Roman Pichler provides practices, advice, and examples for product strategy and roadmapping that you can use to create successful products. InfoQ interviewed Pichler about applying product strategy and roadmapping with agile, innovation in product strategy, eliminating features when defining

products, different kinds of roadmaps, and on measurements for product management.

All in Culture & Methods

- DevOps
  - Infrastructure
  - Continuous Delivery
  - Automation
  - Containers
  - Cloud

Featured in DevOps

Virtual Panel on (Cloud) Lock-In


There's no shortage of opinions on the topic of technology lock-in. InfoQ reached out to four software industry leaders to participate in a lively virtual panel on this topic: Joe Beda, Simon Crosby, Krish Subramanian, and Cloud Opinion.

All in DevOps

San Francisco Nov 7-11

London Mar 6-10, 2017

New York Jun 26-30, 2017

- Mobile
- HTML5
- JavaScript
- APM
- IoT
- Microservices
- DevOps
- Java
- Database

All topics

Configure Once, Run Everywhere: Decoupling Configuration and Runtime

Posted by Anatole Tresch on Jun 17, 2016 | Discuss

Key takeaways

- Configuration is a cross cutting concern across all application types, yet there is currently no Java standard for managing configuration.
- Apache Tamaya is an incubator project designed to provide community collaboration for producing a configuration standard.
- Tamaya decouples configuration from configuration-data providers.
- Reasonable choices are selected by default, in case of collisions, but that default behavior may be overridden by custom mappers.
- Coding API supports a variety of runtime environments such as standalone, CDI, Spring and more.

Two years have come and gone since the gallant attempt by Credit Suisse and Oracle to start a JSR for Java EE configuration went belly up. Reasons for the demise were numerous, and it is not our objective here to elaborate on the details. Suffice it to say that even though the official JSR was never ratified by the JCP Executive Committee, the initiative for standardizing configuration in Java never stopped. In this article I would like to focus on the subsequent work and current state of the initiative.

Why a Configuration Standard is important?

Configuration is a common cross cutting concern across all applications. Properties are usually specified as key = value pairs, and are supplied in files that can be loaded into a Java Properties object. Unfortunately OSGI, Spring, Java EE, SE and other frameworks and solution running in Java all provide their own configuration APIs and formats. Many of them use propertiary XML formats, others use more modern formats such as Yaml. Java EE even does not support dynamic or remote configuration in most cases. And combining different frameworks in an application always is cumbersome due to different configuration formats,

locations and redundancies. All these add unnecessary complexity and is error prone. And it affects code written for one application, but also has impact on integration with surrounding systems. Java in many areas did a tremendous job the last two decades, creating an unbeatable eco-system for development of any kind of applications. It feels quite strange, such a common concern like configuration is lacking a standardized API, which would obviate the need for applications to build their own solution and simplify integration of configured modules provided by different stakeholders.

Motivation and Background

Opinions vary widely on what configuration should do and should be. As a consequence a configuration standard ought not focus on what to configure or when. With that as a driver, we moved the existing body of knowledge and the experimental code base into a new incubator project called Apache Tamaya. We centered early discussions around existing ideas and requirements, but ultimately took a step back, redefined our use cases from scratch, and fashioned a shiny new implementation. Given that configuration is one of the most widely used cross-cutting concerns, it is our hope and our expectation that this work will emerge in some form as a standard that will benefit the whole Java ecosystem.

Some of Tamaya's features include:

- Defines a set of configuration annotations (tamaya-inject-api), which can be added to client code to inject configured values. The annotations work in a uniform way whether your code is running as a plain old Java SE application, or within a CDI container or Spring environment. Even OSGi services are supported.
- Tamaya support is not limited to `String` values, but can be any Java type, as long there are `PropertyConverters` registered that can derive the typed value from the raw configuration value (of type `String`), for example, as a Date or a URL.
- Additionally Apache Tamaya provides myriad extensions and integrations, providing the ability to customize configuration runtime according to the user's requirements. (This solves the challenge of configuration complexity by allowing users to select the most appropriate functionality for their systems). One nice perk is that none of the extensions depend on the core module except in test scope, which provides a functional implementation to execute our test cases).

In summary, with the current release 0.2-incubating of Apache Tamaya, we have a fully functional configuration solution that provides a stable and proven API/SPI. The numerous extensions have successfully demonstrated that it is feasible to model configuration in a reusable and extensible way.

Additionally we have defined a complete annotation based injection API also providing support for Java EE/CDI (and a few other runtime platforms.) So it may be time to revive discussions on whether to cast our ideas into a JSR. For that, every Java enthusiast and expert (current reader included) should have a look at Tamaya and provide feedback, or make some noise on the social media. Needless to say we are not currently endowed with a multi-million dollar marketing budget, so your help would be greatly appreciated.

Now let us dive a bit deeper, and see what "configure once, run everywhere" means to you day-to-day as a developer.

Use Cases

Apache Tamaya covers a variety of use cases, and thanks to our modular structure you can easily add anything you need that isn't already there. At a glance Tamaya's most important features are:

- A complete and uniform Java API for programmatic or annotation based configuration access; suitable for both Java SE and EE environments.
- Support for frameworks such as Spring and OSGI.

- Support for dynamic enrichment (allowing you to introduce additional property sources).
- Filtering of key/values accessed as single properties or as full configuration. This allows keys and values to be changed (e.g. passwords to be masked), omitted or added. Or it would allow access to be constrained based on access control lists.
- By default configuration is organized as an ordered list of PropertySources, each with an ordinal value. PropertySources can provide any kind of properties, e.g. system properties, environment properties, file based properties or any kind of source and format. PropertySources with higher ordinals basically override (by default) entries provided by PropertySources with lower ordinals (one PropertySource implements the mapping for a single resource.)
- As explained above more significant values (values provided by a PropertySource with higher ordinal value) override values with less significance of the same key. To support use cases where a custom behaviour is more appropriate, a user defined combination policy supports a more flexible overriding mechanism. For example, we could define a behaviour to combine two values "a, b" and "c, d, e" to yield "a, b, c, d, e", when configuring a `List` value.
- Support for placeholders within configuration files, e.g. `${sys:a.sys.property}`, `${url:config.server:8090/v1/a.b.c}`, `${conf:cross.ref}`.
- Support for multiple configuration formats such as Yaml, Json, properties etc.
- Integration with new, specialized configuration back ends, such as etcd and Consul.
- Support for dynamic and flexible resource locations, for example in a database, in Consul or etcd, or in a file.
- Support for dynamic configuration handling, events and mutable configuration.

Not yet released features (planned for 0.3-incubating), include:

- Support for configuration usage metrics.
- Support for configuration validation and documentation, e.g. for generating output for a CLI -help command option.
- An easy to use YAML-based configuration DSL.
- A web component for visualizing and managing configuration.

Let's look at some examples of "configure once, run everywhere". Implementing a configurable component using Tamaya normally includes the following steps:

- Implement the component.
- Decide on the configurable aspects.
- Define keys and in-line reasonable defaults, enabling "convention over configuration".
- Add the Tamaya libraries that provide integration with your target runtime, perhaps a Java EE servlet container or Spring Boot application.
- Use the component with the hard-coded defaults, and continue with your normal implementation work without any further configuration. You can document and observe your configuration using Tamaya, and assign the effective file format or configuration back-end later as a simple dependency.

Implementing a SupportContact Class

Let's consider a simple example: suppose we are building a component providing support contact information for an application. The `SupportContact` component is defined as follows:

```
package com.mycompany;

public class SupportContact{
  private String supportOrganization;
  private String phoneNumber;
```

```
  private String email;
  private boolean supports24x7;
  private Collection<Contact> supportContacts;

  public String getSupportOrganization(){
    return supportOrganization;
  }

  public String getPhoneNumber(){
    return phoneNumber;
  }

  [...]
}
```

To configure this class we can implement a constructor to perform the configuration logic:

```
    public SupportContact(){
        this.supportOrganization =
                  ConfigurationProvider.getConfiguration()
                   .getOrDefault("support.organization", "N/A");
    this.phoneNumer =
                  ConfigurationProvider.getConfiguration()
                   .getOrDefault("support.phone", "N/A");
    [...]
}
```

This declarative access actually works, but most developers who work with dependency injection frameworks like Spring will want to use the tamaya-injection module to configure the instance:

```
public SupportContact(){
        ConfigurationInjection.getConfigurationInjector()
           .configure(this);
}
```

The configuration code could also be located in an external configuration class leaving the original class untouched. All of the built-in Java types, such as `String`, `boolean`, and `int`, as well as types from `java.math.BigDecimal` and `BigInteger`, are supported by default. Collection types are also supported by adding the tamaya-collections extension module as a dependency. Tamaya's ConfigurationInjector, an interface for injecting configuration into POJOs, tries to configure all properties found using a best-guess approach, if no config annotations are present. It combines the package, class and field name to an ordered list of candidate keys and tries to lookup corresponding configuration values. The first non-null value encountered is injected. Undefined properties, (where none of the candidate keys matched to a value) are logged with a warning, but no exception is thrown. This makes it possible to provide default values in your code using standard Java, overriding them as required using properties:

```
private String supportOrganization = “N/A”;
private String phoneNumber = “N/A”;
private String email = “N/A”;
private boolean supports24x7 = true;
private Collection<Contact> supportContacts = new ArrayList<>();
```

Diving deeper, Tamaya’s property mapping mechanism will map these entries to the following list of candidate keys:

```
com.mycompany.SupportContact.supportOrganization
SupportContact.supportOrganization
supportOrganization
com.mycompany.SupportContact.phoneNumber
SupportContact.phoneNumber
phoneNumber
com.mycompany.SupportContact.email
SupportContact.email
email
com.mycompany.SupportContact.supports24x7
SupportContact.supports24x7
supports24x7
com.mycompany.SupportContact.supportContacts
SupportContact.supportContacts
supportContacts
```

More explicit methods for assigning configuration keys for configuring your code are also possible by using the Tamaya annotations defined by the tamaya-injection-api extension module. Using that approach you can annotate the class and the properties as follows:

```
@Config(“organization”, defaultValue=“N/A“)
private String supportOrganization;

@Config(value=“phone“, defaultValue=“N/A“)
private String phoneNumber;

@Config(defaultValue=“N/A“)
private String email;

@Config(defaultValue=“true“)
private boolean supports24x7;

@Config(“contacts”, defaultValue=“Admin:admin“)
private Collection<Contact> supportContacts;
}
```

This gives you full control over how Tamaya’s property mapping mechanism will map these

entries:

```
support.organization
support.phone
support.email
support.supports24x7
support.contact
```

So we could put these properties into a simple `.properties` file:

```
support.organization=MyCompany
support.phone=+41(1)23 553 234
support.email=chief-admin@mycompany.com
support.supports24x7=true
support.contact=Chief Admin:Peter Cole;Advisory Admin:John Doe
```

...or a yaml file:

```
---
support:
        organization:      MyCompany
        phone:          +41(1)23 553 234
        email:          chief-admin@mycompany.com
        supports24x7:      true
        contacts:
        - Chief Admin
          Peter Cole
          Advisory Admin
          John Doe
```

Tamaya will handle details like configuration formats, locations, and overrides for you. To discover the location of a project's configuration hooks, the tamaya-model extension can provide a list of defined configuration properties, and also measure their usage.

And Tamaya is flexible enough that you can connect it, with whatever back-end you (or your customers) use, so developers can focus on what to configure. The how becomes an integration aspect that can be handled independently.

Note:

In practice Tamaya offers several options for integration with configuration back-ends:

- Add some test configuration to `META-INF/javaconfiguration.properties` in your (test) classpath. This is the default location, which works out of the box (it can be switched off, if not needed).
- Write your own PropertySource and register it with the JDK `ServiceLoader` to load any configuration (including dynamic values).
- Add a dependency to aTamaya configuration meta-model, which provides and registers pre-implemented and configured PropertySources and PropertySourceProvider instances. This meta-model defines the mapping, location and formats of your configuration. This file can be created and managed by a dedicated platform engineering team, and published

globally to all developers in an organization to ensure a uniform way how applications or services are configured.

- As of the next release 0.3-incubating, a meta-model DSL is planned, that will allow you to describe your configuration runtime system as you would a logging configuration. With that approach a set of profiles and formats are defined and ordered, and each profile is assigned a list of PropertySources and a base ordinal value. Profiles defined being “defaults” are always considered. “default-active” defines the profile active (additional to the any default profiles), when no profile is set and “evaluation” allows to define how the currently active profile is to be determined (using Tamaya’s placeholder mechanism). As an example the following Yaml file defines a complete configuration system:

```
TAMAYA:
 PROFILES_DEF:
  - profiles:            DEFAULTS,DEV,TEST,PTA,PROD
  - supports-multi:      false
  - defaults:            DEFAULTS
  - default-active:      DEV
  - evaluation:          sys-property:ENV, env-property:ENV

 FORMAT-DEF:
  - formats: yaml, properties
  - suffixes: yml,yaml,properties

 PROFILES:
   <ALL>:
     - sources:
       - named:env-properties    # provider name, or class name
       - named:main-args
       - classpath:META-INF/defaults/**/*.SUFFIX
       - file:${config.dir}/defaults**/*.SUFFIX             ?optional=true
       - classpath:META-INF/config/**/*.SUFFIX
       - named:sys-properties
   DEFAULTS:
     - prio:       0        # optional
     - filters:
       - include:DEFAULTS\.*?ignoreCase=true
       - exclude:_\.*   # removes all meta-entries
   DEV:
    - prio:        100          # optional
    - filters:
        - include:DEV\.*?ignoreCase=true
   [...]
   PROD:
    - prio:        1000         # optional
    - filters:
        - include:PROD\.*?ignoreCase=true
```

Typed Configuration Templates

As an alternative to annotating classes, you can opt to use Tamaya's templating feature, which uses typed interfaces to define the configuration. For example to configure a simple web server one could write the following configuration interface and enrich it with Tamaya's annotations:

```
@ConfigSection("server")
public interface ServerConfig{
    @Config(defaultValue="8080");
    int getPort();
    @Config(defaultValue="/");
    int getRootContext();
}
```

Using CDI the configuration can then be injected directly:

```
@Inject
ServerConfig serverConfig;
```

Tamaya will implement the bean based on the annotations and current configuration back ends.

Using the SupportContact Component

Plain Java SE

As we saw above you can inject properties into a Java SE application using Tamaya's SE injection module:

```
SupportContact contact = new SupportContact();
ConfigurationInjection.getConfigurationInjector()
.configure(contact);
```

Java EE

In Java EE, CDI is the lifecycle manager of choice, so we tell CDI to "inject" our component (CDI will actually create a component using the @Dependent pseudo-scope). To achieve this you have to add the tamaya-injection-cdi extension module, which leverages CDI with Tamaya's configuration injection mechanism, so finally you simply let CDI inject your class:

```
@Inject
private SupportContact contact;
```

Spring

Spring also uses the CDI approach, albeit in a Spring way. Just add the tamaya-spring integration module and all of your Spring beans are configurable. So you can then add the SupportContact bean to the Spring context and it will be implicitly configured before

injection:

```
@AutoWire
private SupportContact contact;
```

Vertx

As a last example we want to show how easy it is to add Tamaya's configuration flexibility to your projects. Therefore we look at vertx.io. In Vertx the main abstraction is a verticle. To keep things simple we will have our verticles extend a reusable base class we will call `ConfigurableVerticle`:

```
public abstract class ConfigurableVerticle extends AbstractVerticle{
    public ConfigurableVerticle(){
        ConfigurationInjection.getConfigurationInjector()
            .configure(this);
    }
}
```

We can now configure our verticles using Tamaya's annotations:

```
public class MyVerticle extends ConfigurableVerticle{
    @Config(value="monitoring.count-limit", defaultValue="100")
    private int countLimit;
    [...]
}
```

Of course, this is a fairly simplistic example. But it shows that it is possible to configure a component decoupled from its target runtime without adding any significant complexity to a developer's daily live.

Connecting Configuration Back-Ends

Using default javaconfiguration.properties

Tamaya by default reads environment properties, system properties and `META-INF/javaconfiguration.properties` from the classpath. System properties overrule any others. As a consequence we can configure our component:

- by setting corresponding environment properties. For example, when running in Docker, we can add the following environment properties:

```
ENV support.supportOrganization foo
ENV support.phoneNumber bar
ENV support.email foo2
ENV support.supportContacts bar2
```

- by setting system properties, e.g. `-Dsupport.supportOrganization=Tamaya`
- Or by adding configuration to `META-INF/javaconfiguration.properties` and ensure the resource is visible on your classpath.

This is all still very simple, and as I mentioned we are adding a meta-configuration DSL with 0.3-incubating, that will make it even more flexible.

Adding Test Configuration

Testing is a breeding ground for issues, especially when tests run in parallel threads, that configuration is shared globally. That alone is intrinsically not a problem; however in tests you generally want to test a variety of configurations to ensure your components behave as expected, so sharing configuration in such scenarios might end up in race conditions invalidating your test results. One way to solve this (to be possibly introduced in the next release) would be to implement and register a single `PropertySource` with a very high ordinal. The ordinal value will override any existing properties provided by existing property sources. So we just have to ensure that our `PropertySource`, which is still shared among multiple threads, internally uses a `ThreadLocal` to provide isolation. Together with some static accessors our configuration test setup is ready for use:

```
public TestConfig extends BasePropertySource{
    private static ThreadLocal<Map<String,String>> maps =
                                                new ThreadLocal<>(){
                            [...]
    };

    public TestConfig(){
       super(100000); // default ordinal
    }

    @Override
    public Map<String,String> getProperties(){
        return maps.get();
    }

    public static String put(String key, String value){
        return TestConfig.maps.put(key, value);
    }

    public static String remove(String key){
        return TestConfig.maps.remove(key);
    }
}
```

We can then register our `PropertySource` with the `ServiceLoader` by adding the following line to `META-INF/services/org.apache.tamaya.spi.PropertySource`:

```
com.mycompany.TestConfig
```

And now we can use it directly in our JUnit code:

```
public class TestSupportContact{

  @Before
```

```
  public void setup(){
      TestConfig.put("support.email", "test@127.0.0.1");
  }

  @Test
  public void test(){
      [...]
  }

  @After
  public void teardown(){
      TestConfig.remove("support.email");
  }
}
```

Adding Remote Back-ends

Our sample component configuration was based on classpath resources or local and test files. For most scenarios this is a reasonable solution. But let's imagine that after having completed the application, our customer learned about etcd as a distributed key/value store and asks us to support etcd as a back-end. With Tamaya you have options:

- Implement and register a custom
- Using Tamaya's etcd extension module.

The first approach is similar to the testing scenario we saw earlier.

The second approach is not more difficult; we just add the required entry to Tamaya's extension module. (In case there are conflicts or special mapping requirements, there are a couple of system properties that can be set to adapt the module's behaviour, including the etcd servers being looked up). The important point here is there is still no need to change anything in your code. Your Java code is agnostic to its configuration location and how it is overwritten. This means we have successfully decoupled configuration from its back-ends: "Configure once, run everywhere."

Summary and Outlook

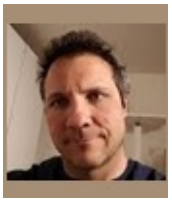
We have seen only a few of the benefits a standardized configuration API offers to the Java ecosystem. In the absence of a standard, everybody is doing it their own way. One goal of this article is to spawn a public discussion and gather your opinions: should we propose this as a new JSR at by the next JavaOne conference in September? From a scope perspective it should be relatively compact:

- A set of annotations as we discussed, leveraging CDI.
- A minimal Java API for use cases where CDI is not available.
- An SPI to provide extension points for custom additions. The Tamaya SPI must be extremely flexible while retaining its simplicity, so that would be a good place to kick off discussions.

Basically this can be run as a Java EE JSR, resulting in something like a "Java EE configuration JSR reloaded" based on the work done at Apache. Since it is basically compatible with CDI 1.1 and does not impose any further requirements on the Java EE platform I think it would still be possible to add it on the current Java EE release train...

Regardless of the JSR decision, Tamaya is looking for additional committers. So if you feel enthusiastic about the topic please contact us!

About the Author



Anatole Tresch - after his studies in information sciences and Economics at the University of Zurich, Anatole worked several years as a managing partner in a consulting firm, being able to gain wide experiences in all areas of the Java ecosystem, in both, small and large enterprise context. Currently Anatole works as a principal consultant for Trivadis and drives mainly Java Money & Currency and Configuration. Anatole is Oracle Star Spec Lead, PPMC member and founder of the Apache Tamaya poddling.

- View
- Reply
- Back to top

Close

Subject Your Reply Quote original message

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

Email me replies to any of my messages in this thread

Post Message Cancel

Close

Subject Your Reply

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

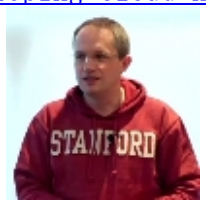
Email me replies to any of my messages in this thread

Cancel

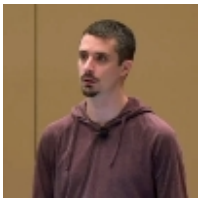
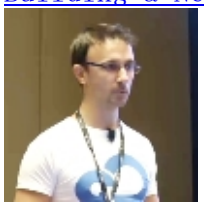
Close

OK

RELATED CONTENT

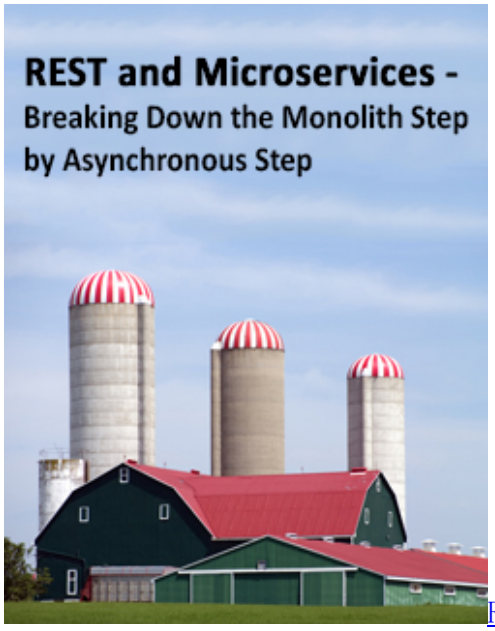
- RightScale DevOps Trends Report: Docker Adoption Rising in the Enterprise, Chef and Puppet Dominate Jun 05, 2016
- Spring Releases Version 1.1 Statemachine Framework Jun 06, 2016
- Spring Cloud Brixton.RELEASE Reaches General Availability Jun 13, 2016
- Spring Framework 5 - Preview & Roadmap Apr 01, 2016 
- Developing Cloud-native Applications with Eclipse and the Spring Tool Suite Apr 11, 2016 

- #NoXML: Eliminating XML in Your Spring Projects Mar 12, 2016 
- Introducing CallTracing(tm) Based on RabbitMQ, Spring and Zipkin Feb 29, 2016


- Isomorphic Templating with Spring Boot, Nashorn and React Feb 29, 2016 
- Building a Next-generation Cloud e-Commerce Platform with Spring Feb 22, 2016

- Developing Cloud-native Applications with the Spring Tool Suite Feb 05, 2016

- HTTP/2 for the Web Developer Jan 29, 2016 

SPONSORED CONTENT

- 

REST and Microservices - Breaking Down the Monolith Step by Asynchronous Step

  Mark Little explores the misuse and misunderstanding of REST (typically HTTP) for microservices.

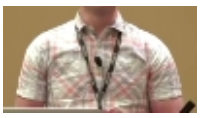
- 

Five tips for CI-friendly Git repos

  Learn some tips for getting your CI system to interact optimally with your Git repository.

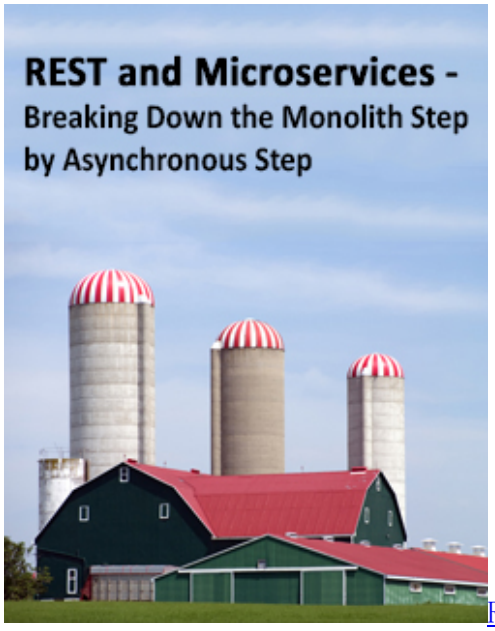
RELATED CONTENT

- Get the Most out of Testing with Spring 4.2 Jan 09, 2016 

- Documenting RESTful APIs Jan 03, 2016 
- InfoQ eMag: Java Agents and Bytecode Jun 06, 2016 
- Eclipse Foundation Releases Neon Jun 22, 2016
- New JSON Binding Library Is Ready for Public Review Jun 20, 2016
- Test Well and Prosper: The Great Java Unit-Testing Frameworks Debate Jun 20, 2016
- Gil Tene: Understanding Hardware Transactional Memory Jun 20, 2016
- Pairing Apache Shiro and Java EE 7 May 31, 2016 

- Java EE Guardians Unite to Save Java EE Jun 17, 2016
- Gluon Announces Full Java 9 Mobile Initiative Jun 10, 2016
- The Roadmap to Kotlin 1.1 Jun 08, 2016

SPONSORED CONTENT

- 

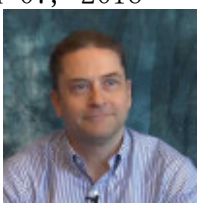
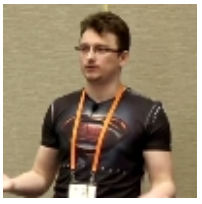
REST and Microservices - Breaking Down the Monolith Step by Asynchronous Step

  Mark Little explores the misuse and misunderstanding of REST (typically HTTP) for microservices.

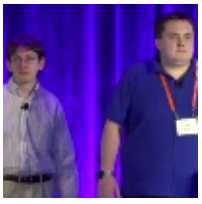
-  [What is Version Control?]

  While it is possible to develop software without using any version control, doing so subjects the project to a huge risk that no professional team would be advised to accept. So the question is not whether to use version control but which version control system to use.

RELATED CONTENT

- Java 9 Will Remove CORBA from Default Classpath Jun 07, 2016

- Interview with Simon Ritter on Java 9 May 17, 2016
- Log4j 2.6 Goes Garbage-Free May 30, 2016

- Building Modern UI for Eclipse RCP Jun 16, 2016

- Science at Eclipse Jun 16, 2016

- Faster Index for Java, or CDT Pays Its Debt to JDT Jun 16, 2016
- Twitter Releases Pants 1.0 Polyglot Build Tool May 28, 2016
- Q&A with Mark Stoodley, Architect of Eclipse OMR Toolkit for Creating Language Runtimes May 27, 2016

- Adventures in 3D with Eclipse ICE and JavaFX Jun 10, 2016

- Java 9's Other Puzzle Pieces May 28, 2016

- Living in the Matrix with Bytecode Manipulation May 26, 2016

InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ

Your email here Subscribe

Development

Big Data Analytics with Spark Book Review and Interview

Safari 10 Ships WebDriver

C# 7 and Beyond with Mads Torgersen

Architecture & Design

Big Data Analytics with Spark Book Review and Interview

Cloud Identity Summit Pushes Change in Identity and Security

C# 7 and Beyond with Mads Torgersen

Culture & Methods

Why Diversity and Inclusion Matters, and How to Drive It

Q&A with Roman Pichler about Strategize

Kevin Miyashiro on Agilility Readiness Canvas

Data Science

Big Data Analytics with Spark Book Review and Interview

Apache TinkerPop Graduates to Top-Level Project

Test Well and Prosper: The Great Java Unit-Testing Frameworks Debate

DevOps

Cloud Identity Summit Pushes Change in Identity and Security

Five Ways to Not Mess Up Microservices in Production

Virtual Panel on (Cloud) Lock-In

- London
  Mar 6-10, 2017
- New York
  Jun 26-30, 2017

InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ

Click to view an example

Your email here Subscribe

- Your personalized RSS
- For daily content and announcements
- For major community updates
- For weekly community updates

Personalize Your Main Interests

- [x] Development
- [x] Architecture & Design
- [x] Data Science
- [x] Culture & Methods
- [x] DevOps

This affects what content you see on the homepage & your RSS feed. Click preferences to access more fine-grained personalization.