BT

- About InfoQ
- Our Audience
- Contribute
- About C4Media

- Exclusive updates on:
-
-
-
-
-

Facilitating the spread of knowledge and innovation in professional software development

Search

Login

- En
- 中文
- 日本
- Fr
- Br

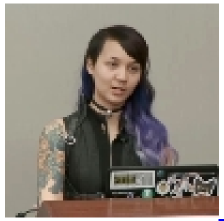
1,296,180 Jun unique visitors

- Development
    - Java
    - Clojure
    - Scala

- .Net
- Mobile
- Android
- iOS
- IoT
- HTML5
- JavaScript
- Functional Programming
- Web API

Featured in Development

The Nihilist's Guide to Wrecking Humans & Systems


Christina Camilleri talks about how social engineering can be used in conjunction with technical attacks to create sophisticated and destructive attack chains, shares some real world war stories and highlights what can be done to protect against these threats.

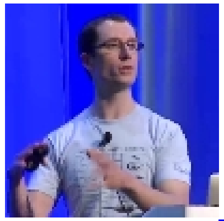
All in **Development**

- Architecture & Design
  - Architecture
  - Enterprise Architecture
  - Scalability/Performance
  - Design
  - Case Studies

- Microservices
- Patterns
- Security

Featured in Architecture & Design

The Seven (More) Deadly Sins of Microservices

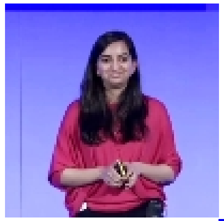
 Daniel Bryant talks about the 2016 edition of the seven deadly sins that if left unchecked could easily ruin our next microservices project, and some of the nastiest anti-patterns in microservices, providing tools to not only avoid but also slay these demons before they tie up the project in their own special brand of hell.

All in **Architecture & Design**

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

Featured in Data Science

[Creating Customer-centric Products Using Big Data]

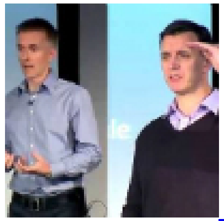

[Kriti Sharma talks about how Barclays is solving some of the toughest big data challenges in financial services using scalable, open source technology. He shares case studies in the areas of: user experience centric predictive analytics and marketing, personalization of applications and targeted offers, social data analytics and sentiment analysis, open stack and open data collaboration etc.]

[All in **Data Science**]

- [Culture & Methods]
  - [Agile]
  - [Leadership]
  - [Team Collaboration]
  - [Testing]
  - [Project Management]
  - [UX]
  - [Scrum]
  - [Lean/Kanban]
  - [Personal Growth]

Featured in Culture & Methods

Culture Eats Principles for Breakfast

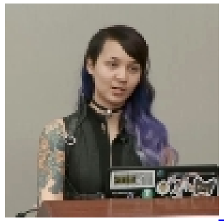

Ian Dugmore and Jonathan Smart talk about the story of Agile transformation across Barclays, what they have done and how they’ve done it, not only in IT but beyond IT. They look at the challenge of using Agile to bring about culture change from different perspectives, comparing and contrasting what needs to be done at an organizational level to make change happen.

All in **Culture & Methods**

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

Featured in DevOps

The Nihilist’s Guide to Wrecking Humans & Systems


Christina Camilleri talks about how social engineering can be used in conjunction with technical attacks to create sophisticated and destructive attack chains, shares some real world war stories and highlights what can be done to protect against these threats.

All in **DevOps**

San Francisco	Nov 7-11
London	Mar 6-10, 2017
New York	Jun 26-30, 2017

- Mobile
- HTML5
- JavaScript
- APM
- Java

- API Design
- Microservices
- IoT
- Database

All topics

You are here: InfoQ Homepage Articles Building Microservices with Spring Boot

Building Microservices with Spring Boot

Posted by Dan Woods on Feb 10, 2015 | 11 Discuss

- Share
- |
-
-
-
-
-
-
-
- "Read later"
- "My Reading List"

The concept of a "microservice architecture" has steadily gained a foothold in software development over the past several years. As a successor to "Service

Oriented Architecture" (SOA), microservices can be categorized in the same family of "distributed systems", and carry forward many of the same concepts and practices of SOA. Where they differ, however, is in terms of the scope of responsibility given to an individual service. In SOA, a service may be responsible for handling a wide range of functionality and data domains, while a general guideline for a microservice is that it is responsible for managing a single data domain and the corresponding functions around that domain. The approach of distributed systems is to decompose a monolithic service infrastructure into individually scalable subsystems, which are able to be organized through vertical slicing of the stack, and interconnected through a common transport.

In a monolithic infrastructure, the services that comprise the system are organized logically within the same code base and unit of deployment. This allows the interdependency between services to be managed within the same runtime, and can mean that common models and resources are able to be shared by the various components of the system. The interconnectivity between the subsystems within a monolithic infrastructure means that business logic and data domains can achieve a high amount of reusability in abstractions and utility functions, often through tight coupling, but with the potential benefit of being able to ascertain how a single change may affect the entire system. That luxury comes at the price of scalability of the individual components of the infrastructure, and means that the resource footprint for the system is at the mercy of its least scalable aspects.

A distributed system decomposes the components of a monolith into individual units of deployment, which are able to evolve their own scaling requirements irrespective of the other subsystems. This means that the resource footprint for the overall system can be more efficiently managed, and the interconnection between components can share a less-rigid contract, since the interdependency is no longer managed through the runtime environment. In traditional SOA, the service boundary may encapsulate a breadth of functionality for a business function, centralized around potentially many data domains. A microservice architecture marries the concept of system distribution with the promise of managing only a single business function and data domain, which means that logically getting a handle on the capabilities of a subsystem is fairly easy to do. It also means that the scope for documenting and testing a subsystem can be more easily managed, and therefore should facilitate a higher coverage of both.

Like SOA, a microservice architecture must be interconnected through a common transport, and in recent years HTTP has proven a formidable avenue for this. Many other options exist, such as binary transport protocols and messaging brokers, and nothing in a microservice architecture particularly favors one mechanism over another, except for maturity of, and accessibility to, libraries that facilitate the intercommunication between services. HTTP, being a mature transport protocol, has client libraries in nearly every programming language and framework, making it an excellent choice for the intercommunication protocol between services. One area where a microservice architecture does express some opinion is around the statefulness of service interactions. Regardless of the underlying transport, it is generally well accepted that microservices should remain stateless in their communication, and follow RESTful paradigms to achieve this. This means that every request and response to a microservice must ensure the state for the function that is being invoked is made fully available every time. To put this more plainly, the service must not make assumptions about the data required for a request based off of some prior interaction. Ensuring REST is properly implemented means that a microservice is inherently designed to scale, and can ensure that new deployments of a service can be met with minimal or no downtime.

Understanding where to slice a monolith and build microservices can be somewhat difficult, especially for legacy code bases that tightly couple a data domain across service boundaries. As a rule of thumb, a vertical slice in the infrastructure can be drawn at the edge of a particular business function. Many microservices may work cooperatively within the context of a vertical slice to achieve its business function. Consider, for example, the functionality of an e-commerce site, which has clearly delineated business functions through the flow of landing page, to customer interaction with a product, to a customer purchasing a product. This process can be broken down into vertical slices for viewing product details; holding a product in a "shopping cart"; and processing the order for one or many products. Within the business context of a customer viewing a product, many microservices may be engaged in the process of gathering the data to show the details for a particular product. On the site's landing page, for example, the titles, images, and prices may be shown for a number of products. The page may gather these details from two backend microservices: one to provide product details; another each product's price. When a customer engages a particular product, the site may additionally call on two other microservices that are designed to provide product ratings and customer reviews. So, to accommodate the architectural vertical slice that makes up the business function of "viewing product details", the slice may be implemented across four backend microservices.

Each of the microservices in the "product" vertical slice is designed to facilitate different insight into the "product" domain, and each has its own ability to scale and be made available according to the system requirements. It is reasonable to assume, for example, that the services responsible for producing the experience on the landing page need to accommodate significantly higher request rates than those services that provide in-depth detail about an individual product. They may also be built on different technical decisions, like caching strategies, which have no valid application in the services that show product ratings or customer reviews. Allowing microservices to make the proper technical decisions for their function enables higher efficiency of resource utilization. In a monolithic architecture, the product ratings and customer review services would be captive to the scaling and availability requirements of the product details and pricing services.

The complexity of a microservice, however, speaks nothing to the actual size of its code base. A common misconception is that the code base for a microservice should also adopt the "micro" concept, but this does not make much sense when considering the goal that a microservice architecture aims to achieve. The goal addresses the decomposition of services into a distributed system, and the complexity of their implementation can be realized through whatever size code base is necessary. The "micro" nomenclature expresses the pattern of responsibility across disparate subsystems, not the code base. Given that the responsibility of a microservice is limited to a single function of a vertical slice in a system, however, the code bases are often concise, easily grasped, and ship with a small deployable. A favored pattern of microservices is that they also ship all the resources necessary to get them running. This means that microservice deployables often embed their own runtime, and can be invoked in a standalone fashion, drastically simplifying the operational overhead associated with deployments.

The history of deploying Java web applications tells a tale of bulky, pre-configured application servers, which take a specialized web archive and extract it into a prescribed, often stateful, runtime environment. These application servers could potentially take tens of minutes of downtime to extract an archive and begin serving the new application code, making it difficult to iterate on changes, and operationally unappealing to have multiple deployments for a single system. As frameworks have evolved to facilitate microservice development, so too has the process of packaging these artifacts for deployment. In modern day Java, microservice web applications are able to easily embed their runtime environment and be packaged into a runnable archive. Modern embedded runtimes, like Tomcat and Jetty, are lightweight versions of their predecessor application servers, and are generally capable of starting up in a measure of seconds. Any system with Java installed is then able to run the deployment, therein simplifying the process of deploying new changes.

Spring Boot

One such framework that has been evolved to be formidable for Java microservice development is Spring Boot. Boot is built on top of the Spring Framework, and garners all the benefits of its maturity, while decorating the complexities of the underlying framework with opinionated fixtures that aid in the development of microservices. Much of Spring Boot is aimed at developer productivity by making common concepts, like RESTful HTTP and embedded web application runtimes, easy to wire up and use. In many respects, it also aims to serve as a "micro-framework", by enabling developers to pick-and-choose the parts of the framework they need, without being overwhelmed by bulky or otherwise unnecessary runtime dependencies. This also enables Boot applications to be packaged into small units of deployment, and the framework is able to use build systems to generate those deployables as runnable Java archives.

The Spring Boot team has provided a convenient mechanism for getting started with building applications, known as the Spring Initializr. This page is designed for bootstrapping the build configuration for a Boot-based web application, and allows developers to categorically choose what libraries they need included in their project. By entering some metadata about their project, choosing their dependencies, and clicking the "Generate Project" button, developers are able to have produced for them a zip file of either a Maven or Gradle based Spring Boot project. This provides the scaffolding for getting started and is an excellent starting place for newcomers to the framework.

As a framework, Boot is built into aggregate modules, known as "starters". These starter modules are compositions of known-good-interoperable version of libraries that it can use to provide some functionality for an application. They are also the structure by which Boot is able to affix its opinions around application configuration, which affords for the simplicity of convention-over-configuration during the development cycles. Many of the starter modules are designed specifically to accommodate a microservice architecture, exposing key functionality to application developers for free. An HTTP-based RESTful microservice in Spring Boot can get away with simply including the actuator and web starter modules. The web module will provide the embedded runtime and ability to develop the microservice's API on top of RESTful HTTP controllers, and the actuatormodule will operationalize the microservice by providing structure and RESTful HTTP endpoints for exposing metrics, configuration parameters, and internal component mappings, which are useful in debugging.

Integral to much of its value proposition as a microservice framework is Boot's ability to seamlessly provide build tooling for projects based on Maven and Gradle. Little additional configuration is required outside of applying the Spring Boot plugin to employ the framework's ability to package a project into a lightweight, runnable deployable. The code shown in Listing 1 shows a Gradle build script, which can be used as the starting point for a Spring Boot microservice. The somewhat more verbose Maven POM example can be chosen from the Spring Initializr site, which reveals the necessity to inform the plugin of the location to the starting class for the application. No such configuration is required when using Gradle, as the plugin will discover the class' location.

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
   classpath 'org.springframework.boot:spring-boot-gradle-plugin:1.2.0.RELEASE'
  }
}
apply plugin: 'spring-boot'
repositories { jcenter()
}
dependencies {
  compile "org.springframework.boot:spring-boot-starter-actuator"
  compile "org.springframework.boot:spring-boot-starter-web"
}
```

Listing 1 - Gradle build script

Using a project from the Spring Initializr will reveal the general project structure required, which is nothing more than following Maven-style conventions for project layout. The code must be placed in src/main/java for it to be properly compiled. The project must then provide an entry point to the application. From the scaffolding from Spring Initializr, there is a DemoApplication.java file, which acts as the main class for this project. The naming of this class is arbitrary, and often simply calling it "Main" will suffice. The example in Listing 1.1 depicts the minimum required code to get started with developing a microservice.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
@EnableAutoConfiguration
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class);
    }
}
```

Listing 1.1 - Spring Boot App

By instructing Boot to "EnableAutoConfiguration", as the class has been annotated, the framework will form opinions about bootstrapping the application to get up-and-running. These opinions are largely derived in a convention-over-configuration manner, to which Boot will perform classpath scanning to ascertain what capabilities the microservices needs to have. In the above examples, the microservice has included the actuator and web starter modules, so the framework will determine that the project is intended as a microservice, and will bootstrap an embedded Tomcat container and begin serving the pre-configured endpoints. The code in the example above does not do much yet, but simply running it will reveal the operational endpoints exposed by actuator. Importing the project into any IDE will allow the microservice to be started by creating a "Run as Java Application" configuration against the "Main" class. Alternatively, the application can be started from the command-line by running the gradle bootRun Gradle task or mvn spring-boot:run Maven goal, depending on what project configuration was chosen.

Working with Data

Building on the "product vertical slice" example from earlier, consider the "product detail" service, which, in conjunction with the "product price" service, provided the detail for the landing page experience. In terms of the microservice's responsibilities, its data domain will be the attribute subset of a "product" related specifically to its name, a short description, a long description, and an inventory id. These details can be modeled as the Java bean depicted in Listing 1.2.

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class ProductDetail {
    @Id
    private String productId;
    private String productName;
    private String shortDescription;
    private String longDescription;
    private String inventoryId;
    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }
    public String getProductName() {
        return productName;
    }
    public void setProductName(String productName) {
        this.productName = productName;
    }
    public String getShortDescription() {
        return shortDescription;
    }
    public void setShortDescription(String shortDescription) {
        this.shortDescription = shortDescription;
    }
    public String getLongDescription() {
        return longDescription;
    }
    public void setLongDescription(String longDescription) {
        this.longDescription = longDescription;
    }
    public String getInventoryId() {
        return inventoryId;
    }
    public void setInventoryId(String inventoryId) {
        this.inventoryId = inventoryId;
    }
}
```

Listing 1.2 - Product Detail POJO

An important thing to notice about the ProductDetail Java bean is that it is annotated using JPA annotations to indicate that it is an entity. Spring Boot provides a starter module for working with JPA entities and RDBMS datasources. Considering the build script in Listing 1, we can add the "dependencies" section to include the Boot starter modules for working with persistent data sets, as shown in Listing 1.3.

```
dependencies {
  compile "org.springframework.boot:spring-boot-starter-actuator"
  compile "org.springframework.boot:spring-boot-starter-web"
  compile "org.springframework.boot:spring-boot-starter-data-jpa"
  compile 'com.h2database:h2:1.4.184'
}
```

Listing 1.3 - Spring Boot dependencies in build script

For demonstration and prototyping purposes, the project now also includes the embedded h2 database type. Boot's autoconfiguration mechanism will identify that h2 is on the classpath, and will generate the necessary table structure for the ProductDetail entity. Under the covers, Boot is leveraging Spring Data for its Object Relational Mapping, and given that, we can leverage its conventions and mechanisms for working with databases. A convenient abstraction that Spring Data provides is the concept of a "repository", which is essentially a data access object (DAO) that is automatically wired together on our behalf. To get CRUD functionality for the ProductDetail entity, we need only create an interface that extends Spring Data's CrudRepository, as demonstrated in Listing 1.4.

```
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
@Repository
public interface ProductDetailRepository extends CrudRepository <ProductDetail, String>{
}
```

Listing 1.4 - Product Detail Data Access Object (Spring Data Repository)

The @Repository annotation on the interface informs Spring that this class should be respected in its specialized role as a DAO. The annotation also serves as a mechanism by which we can inform the framework to automatically wire it into the microservice's configuration, so that we can get access to it through dependency injection. To enable this feature of Spring, we must add the @ComponentScan additional annotation to the Main class from Listing 1.1. When the microservice is started, Spring will scan the project's classpath looking for components, and make them available as autowire candidates within the application.

To demonstrate the microservice's new capabilities, consider the code in Listing 1.5, which simply utilizes the fact that Boot will give us a handle on the Spring ApplicationContext within the main() method.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.ComponentScan;
@ComponentScan
@EnableAutoConfiguration
public class Main {
    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication.run(Main.class);
        ProductDetail detail = new ProductDetail();
        detail.setProductId("ABCD1234");
        detail.setProductName("Dan's Book of Writing");
        detail.setShortDescription("A book about writing books.");
        detail.setLongDescription("In this book about writing books, Dan will show you how to write a book
        detail.setInventoryId("009178461");
        ProductDetailRepository repository = ctx.getBean(ProductDetailRepository.class);
        repository.save(detail);
        for (ProductDetail productDetail : repository.findAll()) {
            System.out.println(productDetail.getProductId());
        }
    }
}
```

Listing 1.5 - Demonstration for bootstrapping data

In this simple example, a ProductDetail object is hydrated with some data, the ProductDetailRepository is leveraged to save that detail, and it is finally used a second time to recall that detail from the database. So far, no additional configuration has been required to get the microservice working with persistent data. We can leverage the prototype code in Listing 1.5 as a basis for defining the RESTful HTTP API contract through Spring's @RestController mechanism.

Designing the API

For the "product detail" microservice, it is probably sufficient to expose simple CRUD-like capabilities, but it may also need the ability to provide some extended functionality, like paged result sets and data filtering. The API for the data set can begin to be realized by starting with a simplecontroller, which Spring will map to an HTTP route. A simple start can be realized through the code sample shown in Listing 1.6, which exposes create and findAll methods, which demonstrate the functionality that was prototyped in the prior example.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/products")
public class ProductDetailController {
    private final ProductDetailRepository repository;
    @Autowired
    public ProductDetailController(ProductDetailRepository repository) {
        this.repository = repository;
    }
    @RequestMapping(method = RequestMethod.GET)
    public Iterable findAll() {
        return repository.findAll();
    }
    @RequestMapping(method = RequestMethod.POST)
    public ProductDetail create(@RequestBody ProductDetail detail) {
        return repository.save(detail);
    }
}
```

Listing 1.6 - Product Detail Controller Class

Spring's @RestController stereotype informs the framework that it should do most of the heavy lifting related to data serialization and binding. Additionally, for those services that will be producing data to this microservice, we need only annotate the create() method's parameter as the@RequestBody in order for Spring to automatically hydrate that object on our behalf. The ProductDetail object can then be saved using the autowired ProductDetailRepository. Boot has decorated these built-in capabilities of Spring with some additional data converters that will leverage Jackson to serialize the ProductDetail objects to JSON for consumers of the microservice's API. Based on the controller example in Listing 1.6, if the service were to receive a JSON payload, like the one depicted in Listing 1.7, to the /products endpoint, a new product detail entry would be created.

```
{
    "productId": "DEF0000",
    "productName": "MakerBot",
    "shortDescription": "A product that makes other products",
    "longDescription": "This is an extended description for a makerbot, which is basically a product that r
    "inventoryId": "00854321"
}
```

Listing 1.7 - JSON Structure Representing a Product

Refreshing the product detail list through an HTTP GET to /products will show the newly created product detail.

Naively binding data and saving it to a repository may be the only use-case for the microservice's create() function. More than likely, however, the service will need to perform some non-trivial business logic to ensure the data going into the product detail is accurate. We can utilize Spring's internal validation framework to ensure that the product detail is validated against the microservice's business logic during data binding. The code in Listing 1.8 shows an implementation of the ProductDetail validator, which reaches out to another microservice to determine the validity of the supplied inventory ID.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.validation.*;
@Component
public class ProductDetailValidator implements Validator {
    private final InventoryService inventoryService;
    @Autowired
    public ProductDetailValidator(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }
    @Override
    public boolean supports(Class<?>clazz) {
        return ProductDetail.class.isAssignableFrom(clazz);
    }
    @Override
    public void validate(Object target, Errors errors) {
        ProductDetail detail = (ProductDetail)target;
        if (!inventoryService.isValidInventory(detail.getInventoryId())) {
            errors.rejectValue("inventoryId", "inventory.id.invalid", "Inventory ID is invalid");
        }
    }
}
```

Listing 1.8 - The ProductDetail Validator

The InventoryService in this example is contrived, but it can be seen that this mechanism for performing data validation is inherently flexible through its ability to query other microservices about their knowledge of some subset of the data domain.

To employ the ProductDetailValidator at data binding time, it will need to be registered with Spring's data binder, which happens at a controller-specific level. The modified controller code shown in Listing 1.9 shows the validator being autowired into the controller, then subsequently registered with the data binder through the initBinder() method. The @InitBinder annotation on that method informs Spring that we wish to customize the default data binder for this class. Additionally, notice the @Valid annotation that is now applied to the ProductDetail object on thecreate() method. This annotation informs the data binder that we wish to perform validation against the request body during data binding. Spring's built-in validator will also provide validation for JSR-303 and JSR-349 (Bean Validation) field-level annotations.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.*;
import javax.validation.Valid;
@RestController
@RequestMapping("/products")
public class ProductDetailController {
    private final ProductDetailRepository repository;
    private final ProductDetailValidator validator;
    @Autowired
    public ProductDetailController(ProductDetailRepository repository, ProductDetailValidator validator) {
        this.repository = repository;
        this.validator = validator;
    }
    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(validator);
    }
    @RequestMapping(method = RequestMethod.GET)
    public Iterable findAll() {
        return repository.findAll();
    }
    @RequestMapping(method = RequestMethod.POST)
    public ProductDetail create(@RequestBody @Valid ProductDetail detail) {
        return repository.save(detail);
    }
}
```

Listing 1.9 - Updated Product Detail Controller, now with Validator

If a consumer of the API were to POST a JSON structure that did not include a valid inventory ID, Spring would identify the validation failure and respond to the consumer with a "400 - Bad Request" HTTP status code. Since the controller has been annotated as a RestController, Spring would also properly serialize the validation failure to a structure the consumer was expecting. As developers of the microservice, we need not do any additional configuration to get this capability.

For the purposes of the e-commerce site example, a product detail microservice with a naive CRUD REST API is not of much value. The service will also need to provide the ability to page and sort the product detail result list, as well as provide some level of search capability. To accommodate the former requirement, the findAll() controller action on the ProductDetailController can be modified to accept query parameters that will limit the result set to a range defined by API consumers. It can then leverage Spring Data's out-of-the-box PagingAndSortingRepositorytype to supply paging and sorting parameters during the findAll() call to the repository. The ProductDetailRepository needs to be modified to inherit from its new type, as shown in Listing 1.10.

```
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.stereotype.Repository;
import java.util.List;
@Repository
public interface ProductDetailRepository extends PagingAndSortingRepository<ProductDetail, String> {
}
```

Listing 1.10 - Updated ProductDetailRepository, now with paging and sort support

The code in Listing 1.11 shows the modified findAll controller action, which employs the repository's new paging and sorting capabilities. An API call to the

/products endpoint, supplying the query string, ?page=0&count=20 will return the first 20 results from the database. In this example, the code is leveraging Spring's ability to specify default values for query parameters, which enables most of them to be optional.

```
@RequestMapping(method = RequestMethod.GET)
public Iterable findAll(@RequestParam(value = "page", defaultValue = "0", required = false) int page,
                                       @RequestParam(value = "count", defaultValue = "10", required = false
                                       @RequestParam(value = "order", defaultValue = "ASC", required = fal
                                       @RequestParam(value = "sort", defaultValue = "productName", require
    Page result = repository.findAll(new PageRequest(page, count, new Sort(direction, sortProperty)));
    return result.getContent();
}
```

Listing 1.11 - Updated findAll action on ProductDetailController, now with paging and sort

When a user of the e-commerce site hits the landing page, the web page may eagerly query for 10 or 20 results, and then lazily query for 50 more after a certain scroll point or time on the page. Having built-in paging capabilities gives control to the consumer for the amount of data returned during any given call. The fully implemented ProductDetailController is depicted in Listing 1.12.

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.MutablePropertyValues;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.*;
import org.springframework.http.*;
import org.springframework.validation.DataBinder;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.*;
import javax.servlet.http.HttpServletRequest;
import javax.validation.Valid;
import java.io.IOException;
```

```
@RestController
@RequestMapping("/products")
public class ProductDetailController {
    private final ProductDetailRepository repository;
    private final ProductDetailValidator validator;
    private final ObjectMapper objectMapper;
    @Autowired
    public ProductDetailController(ProductDetailRepository repository, ProductDetailValidator validator,
                                   ObjectMapper objectMapper) {
        this.repository = repository;
        this.validator = validator;
        this.objectMapper = objectMapper;
    }
    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(validator);
    }
    @RequestMapping(method = RequestMethod.GET)
    public Iterable findAll(@RequestParam(value = "page", defaultValue = "0", required = false) int page,
                                           @RequestParam(value = "count", defaultValue = "10", required = 
                                           @RequestParam(value = "order", defaultValue = "ASC", required =
                                           @RequestParam(value = "sort", defaultValue = "productName", requ
        Page result = repository.findAll(new PageRequest(page, count, new Sort(direction, sortProperty)));
        return result.getContent();
    }
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public ProductDetail find(@PathVariable String id) {
        ProductDetail detail = repository.findOne(id);
        if (detail == null) {
            throw new ProductNotFoundException();
        } else {
            return detail;
        }
    }
    @RequestMapping(method = RequestMethod.POST)
    public ProductDetail create(@RequestBody @Valid ProductDetail detail) {
        return repository.save(detail);
    }
    @RequestMapping(value = "/{id}", method = RequestMethod.PUT)
    public HttpEntity update(@PathVariable String id, HttpServletRequest request) throws IOException {
        ProductDetail existing = find(id);
        ProductDetail updated = objectMapper.readerForUpdating(existing).readValue(request.getReader());
        MutablePropertyValues propertyValues = new MutablePropertyValues();
        propertyValues.add("productId", updated.getProductId());
        propertyValues.add("productName", updated.getProductName());
        propertyValues.add("shortDescription", updated.getShortDescription());
        propertyValues.add("longDescription", updated.getLongDescription());
        propertyValues.add("inventoryId", updated.getInventoryId());
        DataBinder binder = new DataBinder(updated);
        binder.addValidators(validator);
        binder.bind(propertyValues);
        binder.validate();
        if (binder.getBindingResult().hasErrors()) {
            return new ResponseEntity<>(binder.getBindingResult().getAllErrors(), HttpStatus.BAD_REQUEST);
        } else {
            return new ResponseEntity<>(updated, HttpStatus.ACCEPTED);
        }
    }
    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    public HttpEntity delete(@PathVariable String id) {
        ProductDetail detail = find(id);
        repository.delete(detail);
        return new ResponseEntity<>(HttpStatus.ACCEPTED);
    }
    @ResponseStatus(HttpStatus.NOT_FOUND)
    static class ProductNotFoundException extends RuntimeException {
    }
}
```

Listing 1.12 - Fully implemented ProductDetailController

In addition to paged and sorted data, the e-commerce site will undoubtedly need to expose some search engine-like ability. Since each microservice in the vertical slice maintains its own subset of the data domain, it makes sense that they would manage their own searching capabilities. It also allows consumers to asynchronously search a wide range of properties across the breadth of the data domain.

Spring Data allows for customized queries to be associated with a method signature that is attached to the repository interface. This means the repository can have a prescribed JPA query that searches a subset of the properties on each persisted product detail object, allowing the microservice to expose some primitive search-like capabilities. The ProductDetailRepository is modified as shown in Listing 1.13 to incorporate a search() method, which takes a query term and matches that case-insensitive term against the productName or longDescription fields. A list of results is returned to the caller.

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.stereotype.Repository;
import java.util.List;
@Repository
public interface ProductDetailRepository extends PagingAndSortingRepository<ProductDetail, String> {
    @Query("select p from ProductDetail p where UPPER(p.productName) like UPPER(?1) or " +
            "UPPER(p.longDescription) like UPPER(?1)")
    List search(String term);
}
```

Listing 1.13 - Custom query on the ProductDetailRepository

To surface this search functionality, we can build another RestController and map it to the /search endpoint, as demonstrated in Listing 1.14.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.ArrayList;
import java.util.List;
@RestController
@RequestMapping("/search")
public class ProductDetailSearchController {
    private final ProductDetailRepository repository;
    @Autowired
    public ProductDetailSearchController(ProductDetailRepository repository) {
        this.repository = repository;
    }
    @RequestMapping(method = RequestMethod.GET)
    public List search(@RequestParam("q") String queryTerm) {
        List productDetails = repository.search("%"+queryTerm+"%");
        return productDetails == null ? new ArrayList<>() : productDetails;
    }
}
```

Listing 1.14 - Search controller for ProductDetails

A future enhancement to the ProductDetailSearchController may implement the same paging and sorting capabilities for search that the ProductDetailController employs.

Configuration

Spring Boot's opinions about application configuration can get a microservice pretty far off the ground, and in some cases will not need to be overridden at all. When it is time for the service to be deployed, some configuration directives, like what port to run the embedded container on, may need to be derived environmentally or as a result of some other external influence. Boot provides microservice developers a range of ways to overload its opinionated configuration, and the framework understands that configuration may need to be derived from multiple sources.

An important thing to consider with respect to microservice configuration is the service's runtime environment. If the deployment is in a static infrastructure, then it may be ok to pre-define certain configuration directives. To illustrate this more clearly, consider from the previous examples that the datasource for the microservice was simply an embedded h2 instance. In a production deployment, the microservice should point to a persistent datasource, like a MySQL or Oracle database, so the application will need to be configured with the appropriate JDBC URL, username, password, and link to the appropriate JDBC driver class. In a static infrastructure, these values can be pre-defined and packaged in with the application. Out of the box, Boot can resolve configuration from Java properties files, XML configuration files, or YAML configuration files, and will look for configuration files on the root of the classpath in files named application.properties, application.xml, or application.yml (or application.yaml) respectively. For a pre-defined configuration, the properties file in Listing 1.15 shows the configuration directives that will override the default datasource configuration.

```
spring.datasource.url=jdbc:mysql://prod-mysql/product
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Listing 1.15 - Configuration file for a datasource

An important capability of Boot's configuration mechanism is that configuration can be overridden through Java system properties supplied at startup. Any configuration supplied at JVM startup will supercede that which is found in the application.properties file on the classpath. This means that the runtime environment can customize the configuration based on aspects that are unknowable during the packaging of the microservice. For example, if the microservice is running in a non-static environment, like in a cloud deployment, the database host may need to be derived according to the locality of the VM or container. This may be made accessible to the application through system environment variables. Those environment variables can be consumed and exposed easily through JVM startup parameters, or directly within the configuration. In the case of the latter, Spring's property placeholder notation can be used to get a handle on the directive. The configuration file shown in Listing 1.16 is a modified version of that in Listing 1.15, this time demonstrating property placeholder notation with a default value.

```
spring.datasource.url=${JDBC_URL:jdbc:mysql://prod-mysql/product}
spring.datasource.username=${JDBC_USER:root}
spring.datasource.password=${JDBC_PASS:}
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Listing 1.16 - Updated configuration file to use environment vars with defaults

Boot will also look on the filesystem to find a directory named "config", relative to the startup path. In there, it will look for the same sequence of configuration files, and if found will digest that configuration first, before applying any configuration found on the classpath. Thespring.config.location Java system property can also be used to inform Spring as to the location of configuration files. For example, if the microservice's configuration file was homed in /etc/spring/boot.yml, then specifying -Dspring.config.location=/etc/spring/boot.yml would favor that configuration file off the file system. Classpath resources can be ingested through the same mechanism, only adding the classpath: prefix to the property value.

The server port for the embedded container can also be customized through the same configuration mechanism, under the key server.port. This is of paramount importance when running in a PaaS cloud environment, like Heroku, which maps the port range and exposes it to the application through an environment variable. Configuration directives like those shown in Listing 1.16 can be used to map-in the PORT environment variable. Listing 1.17 depicts this configuration.

```
server.port=${PORT:8080}
```

Listing 1.17 - Configuration to map the startup port to an environment var

Packaging

Once the microservice is ready for deployment, Boot's tooling of the build system can be leveraged to generate a lightweight, runnable artifact. As discussed previously in the article, Boot provides plugins for both Gradle and Maven, which allow it to create a runnable JAR file for distribution. Using the same Gradle build script depicted in earlier listings, the JAR file can be built simply by invoking the build task on the project, gradle build. Boot will intercept the jar task and repackage the regularly-generated artifact with one that has all of the dependencies included as a so-called "fat" or "uber" JAR file. Under a Maven project configuration, the package goal will also be intercepted by the Boot plugin, and will perform the same repackaging operation.

Boot's Gradle plugin has an additional benefit in that it integrates with the application plugin, which will generate a distributable tarball, which comes with all the dependencies pre-packaged, and startup scripts for Unix variants and Windows alike. This method of packaging is ideal for deployments, because all of the startup scripts are already written for the microservice. The tarball can simply be extracted on the destination server, and the microservice started through the project-named script in the bin folder.

Although the standalone deployable is the preferred and generally best accepted deployable unit for a microservice, nothing strictly specifies that they must run in a standalone fashion. To that extent, Boot applications are also able to be packaged as WAR files and deployed to an application container. The Gradle build script will need to be modified to apply the war plugin, as shown in Listing 1.18. Similar to before, the build task will produce the web artifact.

```
buildscript {
  repositories {
    jcenter()
  }
  dependencies {
    classpath 'org.springframework.boot:spring-boot-gradle-plugin:1.2.0.RELEASE'
  }
}
apply plugin: 'spring-boot'
apply plugin: 'war'
repositories {
  jcenter()
}
dependencies {
  compile "org.springframework.boot:spring-boot-starter-actuator"
  compile "org.springframework.boot:spring-boot-starter-web"
  compile "org.springframework.boot:spring-boot-starter-data-jpa"
  compile 'mysql:mysql-connector-java:5.1.34'
}
```

Listing 1.18 - Gradle build script with Boot and War plugins applied

In a Maven project, the war packaging can be achieved by changing the packaging configuration in the project's pom.xml file. The snippet in Listing 1.19 shows that modified configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.infoq</groupId>
    <artifactId>sb-microservices</artifactId>
    <version>0.1.0</version>
    <packaging>war</packaging>
    <!-- ...remaining omitted for brevity... -->
</project>
```

Listing 1.19 - Depiction of the start of a Maven pom.xml with War packaging

Gateway API

The template for developing the product detail microservice that was explored in depth in the prior sections can be applied in a similar manner for the other services within each of the e-commerce site's vertical slices. Once the component pieces of each vertical slice are decomposed into a collection of microservices, the system is then realized as a fully distributed, microservice infrastructure. This does present some complications, however, for external consumers, like the e-commerce site's web page, who need to consume data from the system that may be spread across several different microservices. Without some mechanism synthetically recomposing the services into a monolith-looking API, the onus of calling for the disparate data sets and recomposing them into a usable structure, would be on every consumer of the API. This can be a fairly costly tax to consumers who may need to establish potentially dozens of HTTP connections to aggregate some data set. It also means that if one of the services is unavailable or offline, each consumer is responsible for properly handling the scenario of missing data.

A pattern is emerging for microservice infrastructures that presents the idea of a gateway API service, which sits in front of the disparate backend services to provide a comprehensive, easily consumable API. Following through on the example of the e-commerce site, when a visitor to the site decides to view details about a product, there are four services that need to be involved to roll-up the data to product detail view. Instead of having the web page make a call to each of those services, it would merely access an aggregate API endpoint on the gateway service, which would perform the calls itself and merge the result set for the web page. From the web page's perspective, it's only made a single call, but it's getting back the full scope of the data that it needs to render the page.

This has an added benefit in that the data transfer is able to be more-appropriately managed between consumer and backend service. For example, the gateway service may have logic in its service tier for recognizing when a high volume of requests is being made for a specific product's details, and instead of calling the product detail microservice with every request, it could elect to serve that data from cache for some predefined period of time. This effect can dramatically improve performance and reduce network overhead.

Equally important is abstracting the availability of the backend services. The gateway service may be able to make intelligent decisions about what data to serve in the event that a backend service is not accessible. There are a variety of ways to achieve this, but perhaps the most noteworthy mechanism for ensuring the durability of distributed systems at the gateway service is a library from Netflix called Hystrix. There are a lot of capabilities in Hystrix for ensuring resiliency to failure and providing performance optimizations for high volume request sets, but its perhaps most appealing feature is its implementation of the circuit breaker design pattern. Specifically, Hystrix will observe when a link to a backend service has gone dead, and instead of bombarding the offline service with network traffic and waiting for timeouts, it will open that service's circuit, causing subsequent calls to be delegated to a "fallback" method in lieu of actually making the call. Behind the scenes, Hystrix will periodically check the link to see if the backend service has returned to an operational state, and if so, will reestablish communication.

While the circuit is open, the gateway service can serve whatever response it chooses to consumers. This may include some "last known good" data set, perhaps an empty response with some header attached indicating to consumers that the backend circuit is open, or maybe some combination of the two. The resiliency that Hystrix provides is a critical component in any non-trivial distributed system. To understand Hystrix's capabilities more plainly, consider again the e-commerce product vertical slice, with its four services that must be called on to get data for the product detail view. In Listing 1.20 is shown what a ProductService might look like within the gateway API service.

```java
import com.netflix.hystrix.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
import java.util.*;
import java.util.concurrent.*;
@Service
public class ProductService {
    private static final String GROUP = "products";
    private static final int TIMEOUT = 60000;
    private final ProductDetailService productDetailService;
    private final ProductPricingService productPricingService;
    private final ProductRatingService productRatingService;
    private final ProductReviewService productReviewService;
    @Autowired
    public ProductService(ProductDetailService productDetailService, ProductPricingService productPricingSe
                          ProductRatingService productRatingService, ProductReviewService productReviewServ
        this.productDetailService = productDetailService;
        this.productPricingService = productPricingService;
        this.productRatingService = productRatingService;
        this.productReviewService = productReviewService;
    }
    public Map<String, Map<String, Object>> getProductSummary(String productId) {
        List<Callable<AsyncResponse>> callables = new ArrayList<>();
        callables.add(new BackendServiceCallable("details", getProductDetails(productId)));
        callables.add(new BackendServiceCallable("pricing", getProductPricing(productId)));
        return doBackendAsyncServiceCall(callables);
    }
    public Map<String, Map<String, Object>> getProduct(String productId) {
        List<Callable<AsyncResponse>> callables = new ArrayList<>();
        callables.add(new BackendServiceCallable("details", getProductDetails(productId)));
        callables.add(new BackendServiceCallable("pricing", getProductPricing(productId)));
        callables.add(new BackendServiceCallable("ratings", getProductRatings(productId)));
        callables.add(new BackendServiceCallable("reviews", getProductReviews(productId)));
        return doBackendAsyncServiceCall(callables);
    }
    private static Map<String, Map<String, Object>> doBackendAsyncServiceCall(List<Callable<AsyncResponse>:
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        try {
            List<Future<AsyncResponse>> futures = executorService.invokeAll(callables);
            executorService.shutdown();
            executorService.awaitTermination(TIMEOUT, TimeUnit.MILLISECONDS);
            Map<String, Map<String, Object>> result = new HashMap<>();
            for (Future<AsyncResponse> future : futures) {
                AsyncResponse response = future.get();
                result.put(response.serviceKey, response.response);
            }
            return result;
        } catch (InterruptedException|ExecutionException e) {
            throw new RuntimeException(e);
        }
    }
    @Cacheable
    private HystrixCommand<Map<String, Object>> getProductDetails(String productId) {
        return new HystrixCommand<Map<String, Object>>(
                HystrixCommand.Setter
                        .withGroupKey(HystrixCommandGroupKey.Factory.asKey(GROUP))
                        .andCommandKey(HystrixCommandKey.Factory.asKey("getProductDetails"))
                        .andCommandPropertiesDefaults(
                                HystrixCommandProperties.Setter()
                                        .withExecutionIsolationThreadTimeoutInMilliseconds(TIMEOUT)
                        )
        ) {
            @Override
            protected Map<String, Object> run() throws Exception {
                return productDetailService.getDetails(productId);
            }
            @Override
            protected Map getFallback() {
                return new HashMap<>();
            }
        };
    }
    private HystrixCommand<Map<String, Object>> getProductPricing(String productId) {
        // ... snip, see getProductDetails() ...
    }
    private HystrixCommand<Map<String, Object>> getProductRatings(String productId) {
        // ... snip, see getProductDetails() ...
    }
    private HystrixCommand<Map<String, Object>> getProductReviews(String productId) {
        // ... snip, see getProductDetails() ...
    }
    private static class AsyncResponse {
        private final String serviceKey;
        private final Map<String, Object> response;
        AsyncResponse(String serviceKey, Map<String, Object> response) {
            this.serviceKey = serviceKey;
            this.response = response;
        }
    }
    private static class BackendServiceCallable implements Callable<AsyncResponse> {
        private final String serviceKey;
        private final HystrixCommand<Map<String, Object>> hystrixCommand;
        public BackendServiceCallable(String serviceKey, HystrixCommand<Map<String, Object>> hystrixComman
            this.serviceKey = serviceKey;
            this.hystrixCommand = hystrixCommand;
        }
```

```
            @Override
            public AsyncResponse call() throws Exception {
                return new AsyncResponse(serviceKey, hystrixCommand.execute());
            }
        }
    }
```

Listing 1.20 - An example gateway API service that is async and uses Hystrix

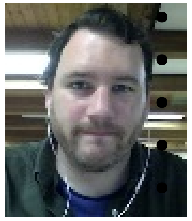
The services depicted in the example should be considered RESTful HTTP clients, perhaps built on top of Spring's RestTemplate, or another HTTP client framework, like Retrofit. The getProductSummary() performs an asynchronous call to the backend services needed to get the product details for the landing page. Similarly, the getProduct() call asynchronously retrieves the details about a product from all of the responsible backend services, and rolls it up for API consumers. In this example, the details about a product will rarely change, and it makes sense for the gateway service to reduce the volume of calls to the backend services when possible, so the getProductDetails() method is able to leverage Spring's @Cacheable to ensure that calls there are cached for a reasonable period of time. The gateway service would then surface the rolled-up details through aRestController that maps to /products. Similar endpoints could be designed for all of the vertical slices of the microservice architecture, and consumers of the system's API would then be able to access it in the same way they would a more traditional monolithic application.

Summary

Spring Boot recognized from an early time the benefits in decomposing monolithic services into distributed microservices, and was designed in a way that makes developing and building microservices a resource-conscious, developer-focused process. By providing starter modules that enable autoconfiguration mechanisms within the framework, applications are able to tap into a powerful subset of functionality that might otherwise require explicit configuration and programmatic composition. These autoconfigured modules can serve as the basis upon which a comprehensive microservice infrastructure, including a gateway API service, is built.

About the Author

Daniel Woods is a Technology Enthusiast specialising in enterprise Java, Groovy, and Grails development. He has over a decade of experience in JVM software development, and shares his experience by contributing to open source projects like the Grails and Ratpack web frameworks. Dan has been a speaker at the Gr8conf and SpringOne 2GX conferences, where he presents his expertise in enterprise application architecture on the JVM.



- Personas
- Architecture & Design
- Development
- Topics
- Architecture

Related Editorial

Design for Continuous Evolution: Immutable Model Is Key for Robustness

HTTP-RPC: A Lightweight Cross-Platform REST Framework

Vowpal Wabbit, A Machine Learning System

Juval Löwy: Why Every Class Should Be a Service

Article Series: Cloud and "Lock-in"

Hello stranger!

You need to Register an InfoQ account or Login or login to post comments. But there's so much more behind being registered.

Get the most out of the InfoQ experience.

Tell us what you think

Please enter a subject

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

Community comments Watch Thread

What about security? by Daniel Stori Posted Feb 11, 2015 01:23
Re: What about security? by Richard GUITTER Posted Feb 11, 2015 02:56
Re: What about security? by Dan W Posted Feb 12, 2015 05:49
Gateway API with Reactive by Zachariah Young Posted Feb 19, 2015 03:46
communication between services by Puneet Arora Posted Feb 20, 2015 10:41
Sample codes by Bai Hantsy Posted Feb 23, 2015 07:42
Sample codes by Bai Hantsy Posted Feb 23, 2015 07:42
@Repository annotation by Arghya Sadhu Posted Feb 23, 2015 08:09
about the update method by Arghya Sadhu Posted Feb 25, 2015 01:42
great stuff by Ethan Winograd Posted Jun 16, 2015 04:27
Issue with Spring Boot jar loading by Himangshu Chowdhary Posted Jan 06, 2016 12:33

What about security? Feb 11, 2015 01:23 by "Daniel Stori"

Great article, but I missed the security point. In general, articles that talk about microservices fails at this point, the security integration of all the services.

- Reply
- Back to top

Re: What about security? Feb 11, 2015 02:56 by "Richard GUITTER"

+1 for the question (that part is always left aside).

After further reading (Start with David Syer presentation), I tried to secure services with OAuth2 for which spring has a quite nice implementation and it worked fine.

- Reply
- Back to top

Re: What about security? Feb 12, 2015 05:49 by "Dan W"

FWIW, I do cover Boot security in an earlier article on the framework: www.infoq.com/articles/microframeworks1-spring-...

- Reply
- Back to top

Gateway API with Reactive Feb 19, 2015 03:46 by "Zachariah Young"

Could you have used a reactive framework for the Gateway API

- Reply
- Back to top

communication between services Feb 20, 2015 10:41 by "Puneet Arora"

Nice article Dan, a great head start for me. How does the communication between services happen? Can you provide configuration and code involved.
Also, when you talk of vertical slicing around business logic boundaries, there could be a case, referring E-commerce example, where i want to search a list of products having a defined price range. In this case, how the product service is going to interact with pricing service to get the desired result.
Is the communication protocol going to be sync or async?

- Reply
- Back to top

Sample codes Feb 23, 2015 07:42 by "Bai Hantsy"

Great article! But where I can get the sample codes of this article?

- Reply
- Back to top

Sample codes Feb 23, 2015 07:42 by "Bai Hantsy"

Great article! But where I can get the sample codes of this article?

- Reply
- Back to top

@Repository annotation Feb 23, 2015 08:09 by "Arghya Sadhu"

I think it would work without the @Repository annotation on the interface ProductDetailRepository ....right?

- Reply
- Back to top

about the update method Feb 25, 2015 01:42 by "Arghya Sadhu"

how the update method is saving data without calling any method from the repository

- Reply
- Back to top

great stuff Jun 16, 2015 04:27 by "Ethan Winograd"

really nicely presented. thank you.

- Reply
- Back to top

Issue with Spring Boot jar loading Jan 06, 2016 12:33 by "Himangshu Chowdhary"

Hi Dan,

We are planning to create a micro service architecture using Boot. In the process of it, there's one sequence where after packaging a Spring Integration application as an executable jar, we 're trying to load it on the fly in a Java class using URLClassLoader.
The issue that we face here is that although the jar gets loaded, the beans configured in the spring-context.xml is not constructed due to which the entire execution halts. Any help with this would be of great help.

PS: The executable jar, when run standalone does run smoothly.

Regards,
Himangshu

- Reply
- Back to top

Close

by

on

- View
- Reply
- Back to top

Close

Quote original message

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

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

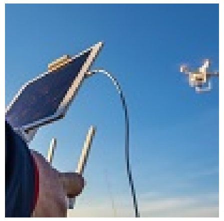
Cancel

Close

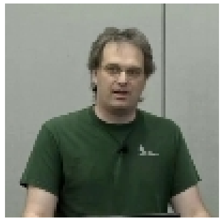
OK

RELATED CONTENT

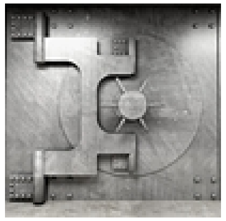
- Design for Continuous Evolution: Immutable Model Is Key for Robustness Jul 13, 2016



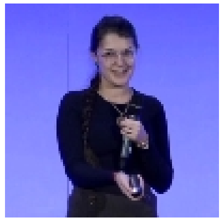
- HTTP-RPC: A Lightweight Cross-Platform REST Framework Jul 12, 2016



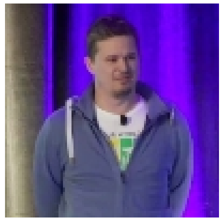
- Vowpal Wabbit, A Machine Learning System Jul 03, 2016
- Juval Löwy: Why Every Class Should Be a Service Jul 01, 2016



- Article Series: Cloud and "Lock-in" Jul 01, 2016



- Distributed Systems in Practice, in Theory Jun 28, 2016
- Stop Over-Engineering, Build What the Customer Really Needs Jun 25, 2016
- FaaS, PaaS, and the Benefits of the Serverless Architecture Jun 25, 2016
- The InfoQ Podcast: James Shore, Llewellyn Falco, and Rebecca Wirfs-Brock on TDD and Architecture Jun 03, 2016

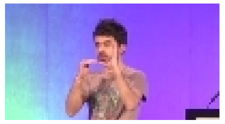


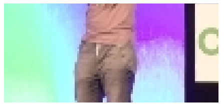
- A Review of Eclipse 4, Its APIs and Architecture May 26, 2016

- A Reference Architecture for the Internet of Things (Part 2) May 25, 2016

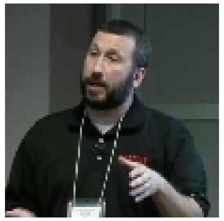
QCon
SAN FRANCISCO 2016
November 7 - 11
International Software Development Conference
SAVE $700 Before Jul 30th

RELATED CONTENT





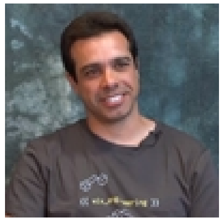
- Subside Failure: Partitioning Time and Space May 21, 2016



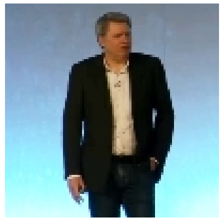
- Netflix Keystone - How We Built a 700B/day Stream Processing Cloud Platform in a Year May 20, 2016

- Jason McGee of IBM Talks about Open Source Projects and the Interactions at the Collaboration Summit May 15, 2016

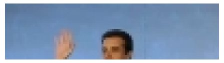
- The Morning Paper Quarterly Review May 09, 2016

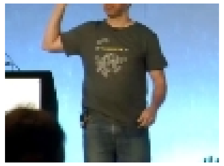
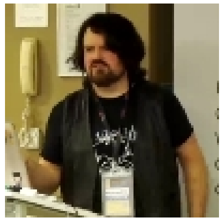


- Aviran Mordo Discusses Technical Leadership, DevOps and Distributed Computing May 06, 2016

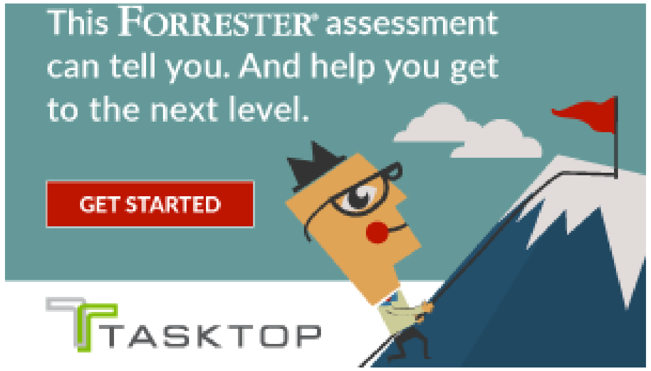


- DDD and Microservices: At Last, Some Boundaries! Apr 16, 2016

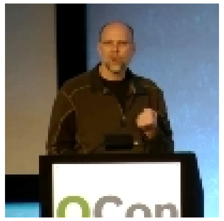


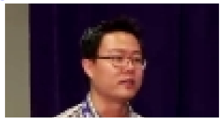
- The Microservices and DevOps Journey Apr 09, 2016 
- Interview with Gil Tene on Hardware Transactional Memory Apr 08, 2016
- Domain Architecture Isomorphism: How Spotlight Inverted Conway's Law Apr 08, 2016 
- InfoQ eMag: QCon London 2016 Report Apr 06, 2016

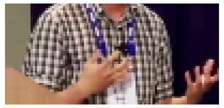
- Key Takeaway Points and Lessons Learned from QCon London 2016 Apr 05, 2016

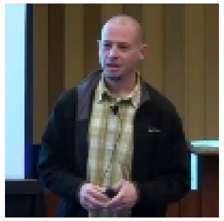

This FORRESTER® assessment can tell you. And help you get to the next level.
GET STARTED
TASKTOP

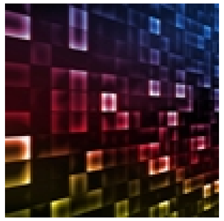
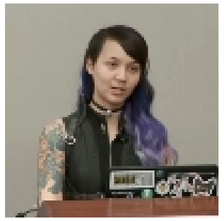
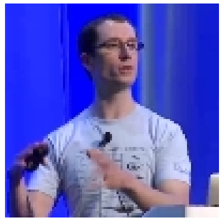
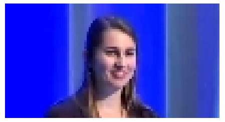
RELATED CONTENT

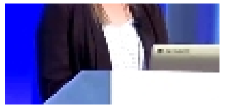
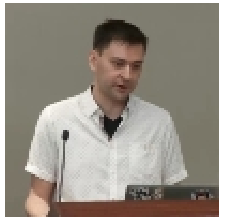
- #NetflixEverywhere Global Architecture Mar 23, 2016 
- One API, Many Facades? Mar 13, 2016
- Will AI Surpass Human Intelligence? Interview with Prof. Jürgen Schmidhuber on Deep Learning Mar 08, 2016





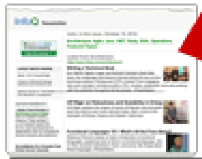
- Architecting Distributed Databases for Failure Feb 28, 2016 
- Just Enough Software Development in Times of Rapid Change Feb 13, 2016

- A Reference Architecture for the Internet of Things Jan 29, 2016 
- Microservices in the Real World Jan 27, 2016 
- The Nihilist's Guide to Wrecking Humans & Systems Jul 18, 2016 
- The Seven (More) Deadly Sins of Microservices Jul 18, 2016 

- Lessons Learned on Uber's Journey into Microservices Jul 18, 2016 
- Modern iOS Application Security Jul 17, 2016 

InfoQ Weekly Newsletter

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



Your email here Subscribe

Development

The Nihilist's Guide to Wrecking Humans & Systems

Modern iOS Application Security

Facebook Announces Open-Source Swift SDK Beta for iOS

Architecture & Design

The Seven (More) Deadly Sins of Microservices

Lessons Learned on Uber's Journey into Microservices

Basho Open Sources Time Series Database Riak TS 1.3

Culture & Methods

Culture Eats Principles for Breakfast

Full Stack Kanban - Live

Making a Sandwich: Effective Feedback Techniques

Data Science

Creating Customer-centric Products Using Big Data

HTTP-RPC: A Lightweight Cross-Platform REST Framework

Key Takeaway Points and Lessons Learned from QCon New York 2016

DevOps

The Nihilist's Guide to Wrecking Humans & Systems

Fighting the #fintech Wave with DevOps

InfoQ eMag: Exploring Container Technology in the Real World

- Home
- All topics
- QCon Conferences
- About InfoQ
- Our Audience
- Contribute
- About C4Media
- Create account
- Login

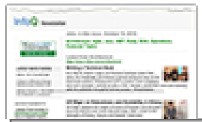
- **QCons Worldwide**
- Shanghai

Oct 20-22, 2016
- San Francisco
  Nov 7-11, 2016
- Tokyo 2016
- London
  Mar 6-10, 2017
- Beijing
  Apr 16-18, 2017
- São Paulo
  Apr 24-26, 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

- ✓ Development
- ✓ Architecture & Design
- ✓ Data Science
- ✓ Culture & Methods
- ✓ DevOps

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

General Feedback	Bugs	Advertising	Editorial	Marketing
feedback@infoq.com	bugs@infoq.com	sales@infoq.com	editors@infoq.com	marketing@infoq.com

BT