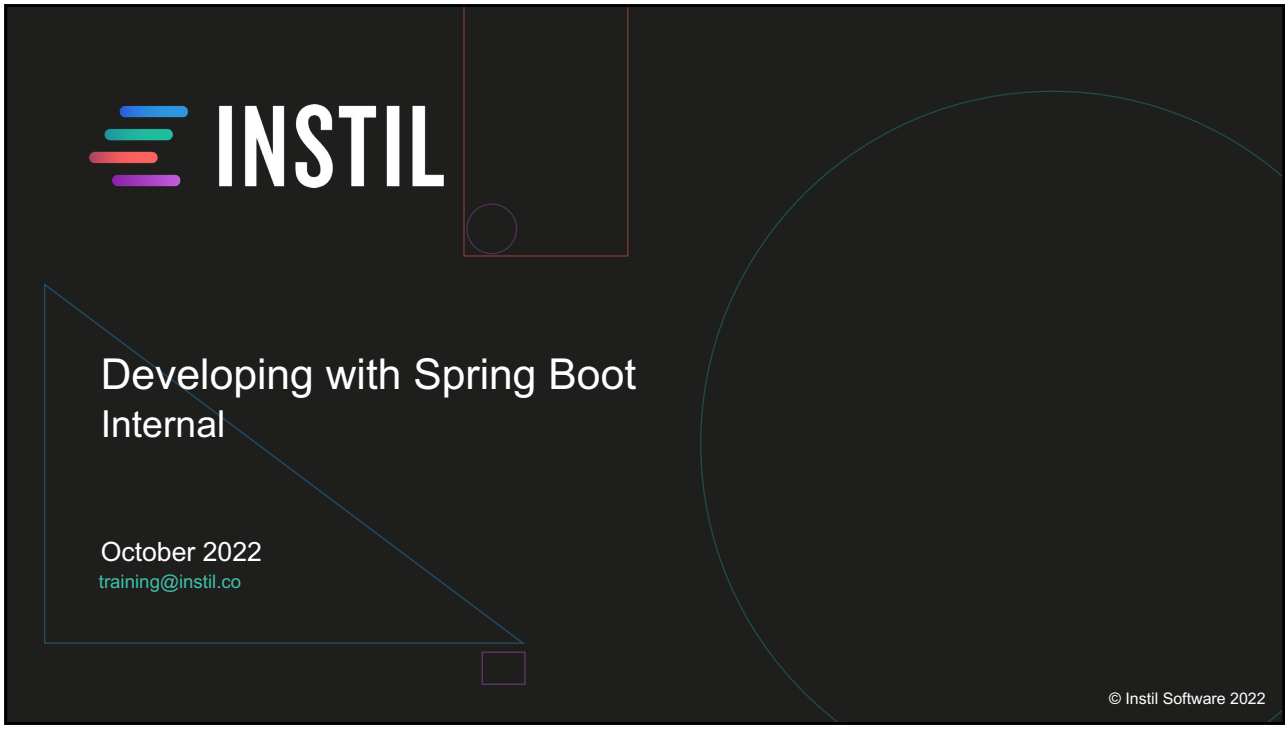
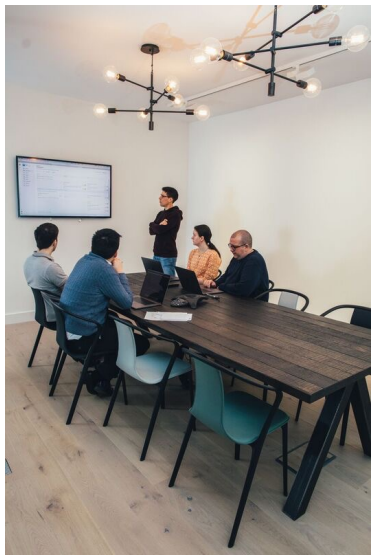


1

2

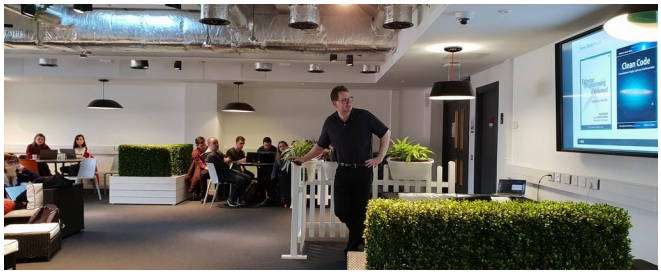
About Instil



INSTIL

3

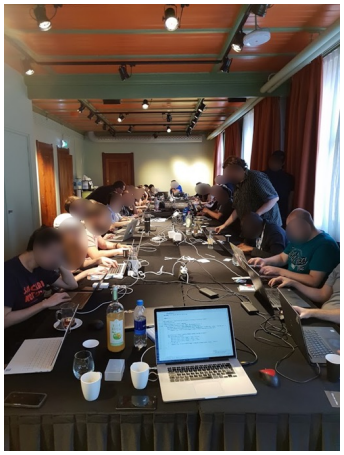
About Your Instructor



INSTIL

4

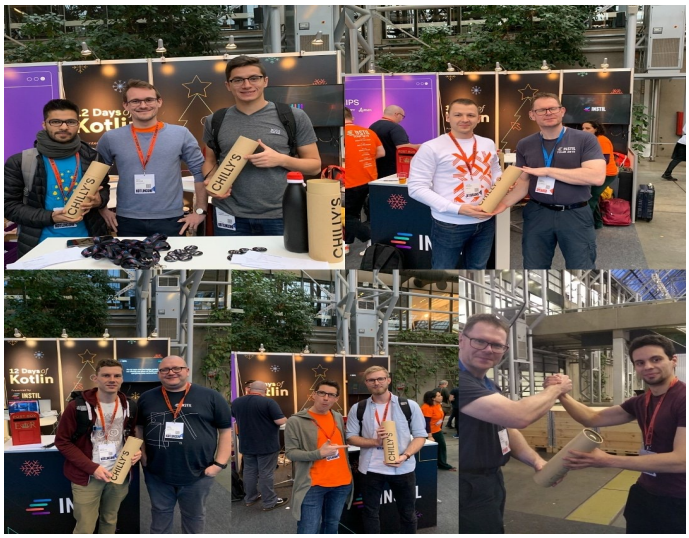
Kotlin Conf 2018



INSTIL

5

Kotlin Conf 2019



INSTIL

6

Facilities and Logistics

Prefer video on
- It's better if we see each other

Default to being on mute
- But do contribute to discussion

Use the Galley Mode in Zoom
- So you can see everyone while also viewing the screen

INSTIL

7

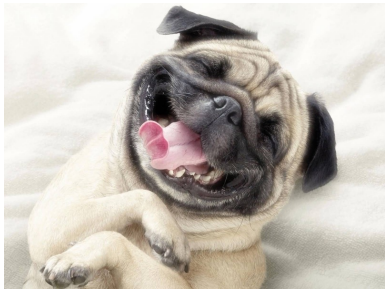
Distributed not remote

There is no back seat in online education
- Every student is in the front row

You're learning programming
- The best way to learn is to do

Let's spend more time doing and talking
- Shout at me if I talk for more than 15 minutes
- Without giving opportunities for contribution

Have fun!
- Play is the most efficient way to learn
- Just ask any puppy or kitten ☺



INSTIL

8

About These Slides

These slides cover Spring from the bottom up

- Starting with Dependency Injection and AOP
- Then working through microservices and database access to testing and security

This is the order in which to review the material

- But not necessarily how we deliver the course

We may take a top-down approach

- Building a sample app with minimal knowledge
- Adding the fine detail once we have a skeleton

INSTIL

9

Introducing Spring

And the Spring Ecosystem

INSTIL

10

The Evolution of Spring

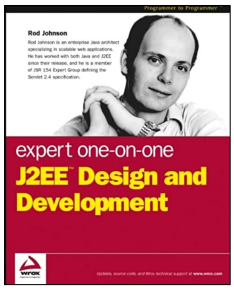
Spring was born from a dislike of JEE

- JEE can be viewed as cumbersome, awkward and intrusive
- Especially for small to medium size projects

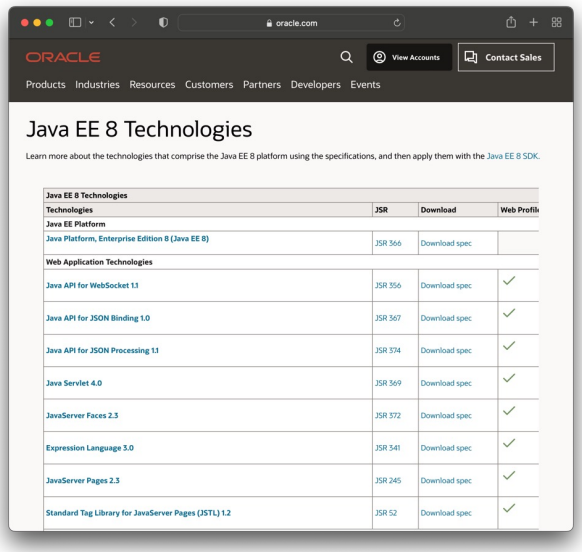
The creator of Spring (Rod Johnson) laid the foundations for a simpler approach in his book 'Expert One on One J2EE'

Today Spring is effectively an alternative to JEE

- Especially with the additions of Spring Boot and WebFlux



The JEE Specifications



The Growth Of Spring

Initially Spring was a Dependency Injection container

- We will cover what Dependency Injection is later

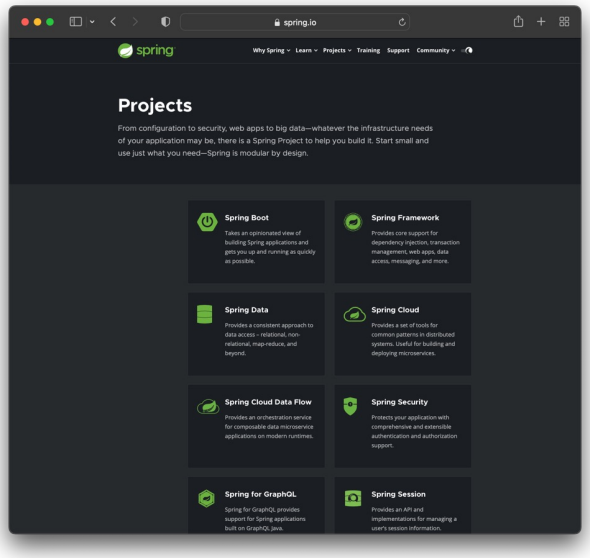
It slowly evolved into a family of frameworks

- Which lived on top of, or replaced, the matching JEE Specifications

All the frameworks interoperate with each other

- Spring MVC + Spring Security + Spring Data is a common combination
- Obviously not all combinations would make sense or be practical

The Many Spring Frameworks



JEE Specifications vs. Spring Frameworks

Area	Frameworks	Official JEE Spec
Web Tier	Spring MVC Struts 2 Play *Many more…*	JSF JSR371
Dependency Injection	Spring Guice	CDI
Database Access	Hibernate MyBatis JOOQ	JPA
SOAP	Spring WS	JAX-WS
REST	Spring MVC Spring WebFlux	JAX-RS
Security & Transactions	Spring Services	EJB

15

Spring History

- Expert One on One J2EE (November 2002)
- Spring 1.0 Milestone 1 (August 2003)
- Spring 1.0 (March 2004)
- Spring 1.1 (September 2004)
- Spring 1.2 (May 2005)
- Spring 2.0 (October 2006)
- Spring 2.5 (November 2007)
- Spring 3.0 (December 2009)
- Spring 3.1 (December 2011)
- Spring 3.2 (August 2013)
- Spring 4.0 (December 2013)
- Spring 5.0 (September 2017)

16

Key Features of Spring Core

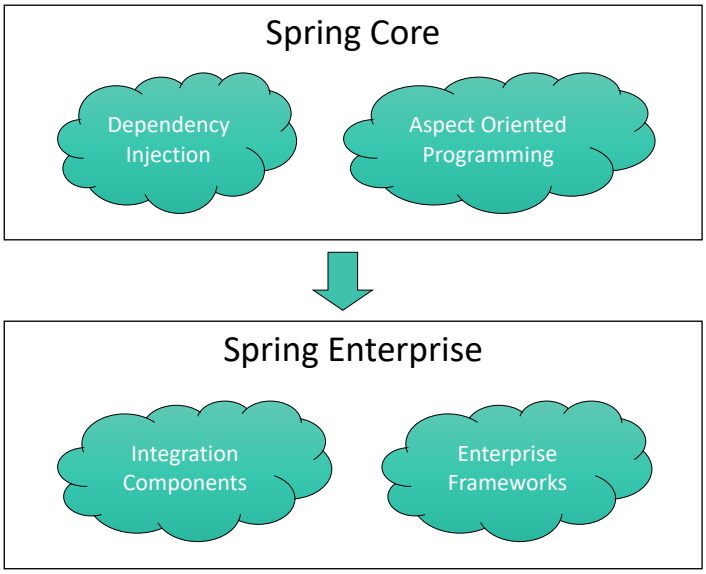
Spring Core has three key features
- The first two features enable the third

Spring builds beans via dependency injection
- Which enables your application to be reconfigured on the fly
- This is very useful if you are adopting Agile Development

Spring can act as a platform for AOP
- In a manner that makes adopting AOP much less scary

Spring works as an integration tool
- Enabling you to integrate other frameworks
- Such as Kafka for messaging or Neo4j for storage

17



18

Spring Core vs. Spring Boot

Spring Core is used in many places:

- Thick clients, android apps, batch processing systems etc…
- The most common use is Spring MVC based Web Apps

It is typically combined with other frameworks

- From both inside and outside the Spring ecosystem

This presents quite a challenge for configuration

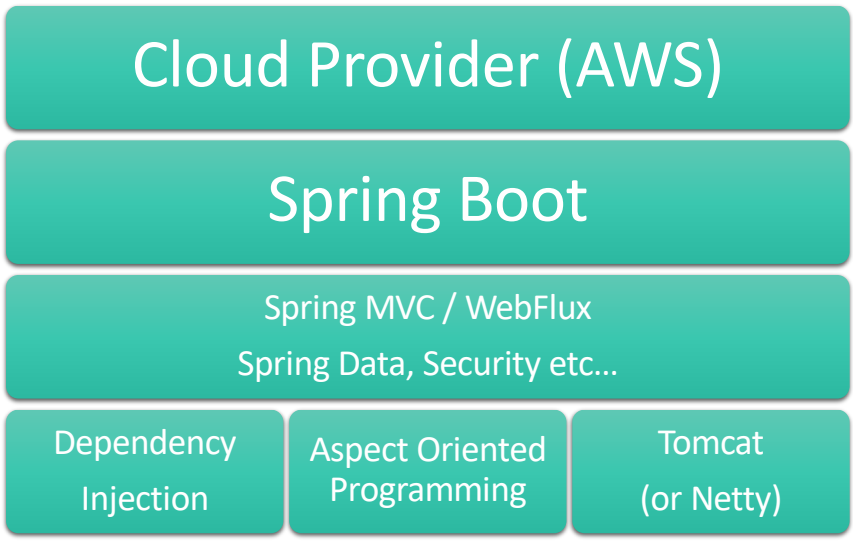
- Spring apps used to have lengthy XML configuration
- This was a major impediment to the growth of Spring

Spring Boot was designed to solve this problem

- It is a 'meta-framework' that configures Spring for you

19

Common Configuration for RESTful Services



20

Features Added in Spring 5

Java 8 became the minimum language version
- Meaning that existing API's can be refactored with default methods, better generics and support for functional idioms
- Method parameters can now be accessed via reflection

Java 9 was supported
- Allowing you to take advantage of the module system

Features relating to Enterprise Java required JEE7
- So you can use JSON-B, newer versions of JPA etc…

Functional programming featured prominently
- Spring MVC added support for Reactor and RxJava
- Spring WebFlux was added as an alternative to MVC

INSTIL

21

Dependency Injection

- Via Annotations, XML and DSLs

INSTIL

22

Introducing Dependency Injection

Consider the '`SampleShop`' class on the next slide

- Disregard any deviations from your preferred syntax ☺

It has a '`makePurchase`' method with 3 dependencies:

- The Stock Check Engine queries stock levels
- The Pricing Engine applies offers and discounts
- The Payment Engine processes credit card transactions

Think about the following questions

- Does this count as good object oriented design?
- What problems would this cause for testing and QA?
- What issues might be encountered in production?

Should This Be Considered Good Code?

```
public class SampleShop {
    public SampleShop() {
        pricingEngine = new PricingEngine(500, 10);
        stockCheckEngine = new StockCheckEngine();
        paymentEngine = new PaymentEngine("www.somewhere.com");
    }

    public boolean makePurchase(String itemNo,
                                int quantity,
                                String cardNo) {
        if (stockCheckEngine.check(itemNo) >= quantity) {
            double charge = pricingEngine.price(itemNo, quantity);
            if (paymentEngine.authorize(cardNo, charge)) {
                return true;
            }
        }
        return false;
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v0/SampleShop.java

Introducing Dependency Injection

The '`SampleShop`' encapsulates the creation of its dependencies
- It is tightly coupled to a single implementation of each engine

These are its Encapsulated Dependencies
- '`PricingEngine`', '`PaymentEngine`' and '`StockCheckEngine`' are hardwired

This raises two closely related problems
- It is impossible to create a unit test for the '`SampleShop`'
  - The closest we might get is an integration test
- We cannot create custom configurations for special needs
  - E.g. during QA we might want to have engines that use local databases

Introducing Dependency Injection

The OO best practice is now to **inject** dependencies
- Also known as 'configuration from above'

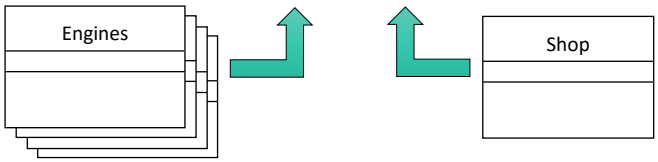
If class 'Foo' is dependent on 'Bar' then we:
- Extract a base class or interface from 'Bar' (e.g. 'IBar')
  - This makes room for multiple implementations
- Lighten the references in 'Foo' to be of type 'IBar'
  - Again making room for multiple implementations
- Pass the implementation of 'IBar' into 'Foo'
  - This could be done by an argument to the constructor
  - JavaBean properties are the other standard option
  - The reflection API would let you set fields directly

Extracting Interfaces

```
public interface PaymentEngine {
    public boolean authorize(String cardNo, double amount);
}
```

```
public interface PricingEngine {
    public double price(String itemNo, int quantity);
}
```

```
public interface StockCheckEngine {
    public int check(String itemNo);
}
```



INSTIL

27

Breaking Up Dependencies

```
public class PaymentEngine {
    public boolean authorize(String cardNo, double amount) {
        //implementation omitted
    }
}
```

```
public interface PaymentEngine {
    public boolean authorize(String cardNo, double amount);
}
```

```
public class PaymentEngineStub implements PaymentEngine {
    public boolean authorize(String cardNo, double amount) {
        return amount < 1000;
    }
}
```

```
public class PaymentEngineWebService implements PaymentEngine {
    public boolean authorize(String cardNo, double amount) {
        //implementation omitted
    }
}
```

INSTIL

28

Dependency Injection via JavaBean Properties

```
public class SampleShop {
    public void setPaymentEngine(PaymentEngine paymentEngine) {
        this.paymentEngine = paymentEngine;
    }

    public void setPricingEngine(PricingEngine pricingEngine) {
        this.pricingEngine = pricingEngine;
    }

    public void setStockCheckEngine(StockCheckEngine stockCheckEngine) {
        this.stockCheckEngine = stockCheckEngine;
    }

    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        if (stockCheckEngine.check(itemNo) >= quantity) {
            double charge = pricingEngine.price(itemNo, quantity);
            if (paymentEngine.authorize(cardNo, charge)) {
                return true;
            }
        }
        return false;
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

SampleShop.java

29

Dependency Injection via Constructor Parameters

```
public class SampleShop {
    public SampleShop(PricingEngine pricingEngine,
                      PaymentEngine paymentEngine,
                      StockCheckEngine stockCheckEngine) {
        super();
        this.pricingEngine = pricingEngine;
        this.paymentEngine = paymentEngine;
        this.stockCheckEngine = stockCheckEngine;
    }

    @Override
    public boolean makePurchase(String itemNo,
                                int quantity,
                                String cardNo) {
        if (stockCheckEngine.check(itemNo) >= quantity) {
            double charge = pricingEngine.price(itemNo, quantity);
            if (paymentEngine.authorize(cardNo, charge)) {
                return true;
            }
        }
        return false;
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

SampleShop.java

30

The Role of Spring Core

Spring takes the concept of DI one stage further

- Given a class and a set of dependencies it wires the two together
- You no longer need to explicitly state how to build (for example) the shop

The wiring instructions were originally given as XML

- Over time using annotations became more popular
- Other options (like Groovy and Kotlin DSLs) are also available

XML based wiring is discouraged not deprecated

- Some teams prefer to have wiring fully separate from code
- In complex scenarios it still remains a viable option

Some Spring Annotations for DI

Annotation	Description
`@Autowired`	Marks a method or constructor as a target to be invoked
`@Qualifier`	Specifies the name of the bean to be inserted
`@Service, @Component, @Repository, etc…`	Different annotations for marking a class as a Spring bean. From the perspective of DI all these annotations are equivalent, the beans are treated differently by higher layer frameworks such as Spring MVC
`@Configuration`	Marks a class as a provider of beans
`@Bean`	Marks a method as a bean provider
`@Value`	Runs a Spring EL and injects the result into the method or field
`@DependsOn`	Used to specify beans that must be instantiated before the current bean. Used when 'A' depends on 'B' but 'B' is not injected into 'A'. In other words 'A' relies on a side effect of 'B'
`@Primary`	Specifies that a bean should be preferred above others when several candidates are found during autowiring by type
`@Lazy`	Specifies whether a bean should be instantiated lazily

The Application Context

Spring Core is mostly the Application Context

- This is the registry of all the Spring components we have discovered
- Note that the term 'bean' simply means Java classes that follow the JavaBean spec for writing properties

Contexts are built on the idea of Bean Factories

- But the types for bean factories are now deprecated
- Application Contexts add support for I18N, AOP etc…

There are many kinds of Application Context

- '`AnnotationConfigApplicationContext`' is the most common

A Hello World Example

In the following slides we:

- Annotate our three engines with '`@Component`'
- Annotate the shop type with '`@Service("shop")`'
- Annotate the setter methods of shop with '`@Autowired`'
- Create an application context, passing in a array of class objects
- Ask the context object to build the bean called 'shop'

Dependencies are resolved via 'autowiring by type'

- Spring knows to call '`setPricingEngine`' via '`@Autowired`'
- The matching bean is found thanks to '`@Component`'
- There can be only one match (unless we use '`@Primary`')

Basic Spring Dependency Injection

```
public class Program {
   public static void main(String[] args) throws Exception {
      try (AbstractApplicationContext context = buildContext()) {
         Shop shop = context.getBean("shop", Shop.class);
         useShop(shop);
      }
   }

   private static void useShop(Shop shop) {
      if (shop.makePurchase("AB123", 20, "DEF456GHI78")) {
         System.out.println("Purchase Succeeded");
      } else {
         System.out.println("Purchase Failed");
      }
   }

   private static AbstractApplicationContext buildContext() {
      Class<?>[] beans = { SampleShop.class,
                           PaymentEngineMock.class,
                           PricingEngineMock.class,
                           StockCheckEngineMock.class };
      return new AnnotationConfigApplicationContext(beans);
   }
}
```

v3/Program.java

INSTIL

35

Basic Spring Dependency Injection

```
@Service("shop")
public class SampleShop implements Shop {
    @Autowired
    public void setPaymentEngine(PaymentEngine paymentEngine) {
        this.paymentEngine = paymentEngine;
    }
    @Autowired
    public void setPricingEngine(PricingEngine pricingEngine) {
        this.pricingEngine = pricingEngine;
    }
    @Autowired
    public void setStockCheckEngine(StockCheckEngine stockCheckEngine) {
        this.stockCheckEngine = stockCheckEngine;
    }
    @Override
    public boolean makePurchase(String itemNo, int quantity,
                                String cardNo) { ... }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v4/SampleShop.java

INSTIL

36

Basic Spring Dependency Injection

```
@Component
public class PaymentEngineMock implements PaymentEngine {
    // ...
}
```

```
@Component
public class PricingEngineMock implements PricingEngine {
    // ...
}
```

```
@Component
public class StockCheckEngineMock implements StockCheckEngine {
    // ...
}
```


Other Ways of Detecting Beans

In the last example we provided an array of class objects

- This requires that we manually track all our beans

A more convenient approach is ‘classpath scanning’

- We call ‘scan’ passing in the name of a package
- Spring finds the beans belonging to that package
- NB ‘refresh’ must be called before the definitions are used

Individual types can also be included via ‘register’

- Once again ‘refresh’ must be called to reset the context

```
private static AbstractApplicationContext buildContext() {
    AnnotationConfigApplicationContext context =
           new AnnotationConfigApplicationContext();
    String currentPackageName = Program.class.getPackage().getName();
    context.scan(currentPackageName);
    context.refresh();
    return context;
}
```

```
private static AbstractApplicationContext buildContext() {
    AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
    context.register(PaymentEngineMock.class);
    context.register(PricingEngineMock.class);
    context.register(StockCheckEngineMock.class);
    context.register(SampleShop.class);
    context.refresh();
    return context;
}
```


Introducing Config Classes

A '`@Configuration`' class can contain bean provider methods

- Which are annotated with '`@Bean`' and should return an interface type

Classpath scanning can still also be enabled

- Via the '`@ComponentScan`' annotation

'`@Bean`' methods should only be put in config classes

- If they are put elsewhere they are handled in 'lite' mode

Using Configuration Classes

```
private static AbstractApplicationContext buildContext() {
    AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext();
    context.register(SampleConfig.class);
    context.refresh();
    return context;
}
```

```
@Configuration
@ComponentScan(basePackages = "demos.spring.notes.v4")
public class SampleConfig {
}
```


41

Creating Configuration Classes

```
@Configuration
public class SampleConfig {
    @Bean
    public PricingEngine foo() {
        return new PricingEngineMock();
    }

    @Bean
    public PaymentEngine bar() {
        return new PaymentEngineMock();
    }

    @Bean
    public StockCheckEngine zed() {
        return new StockCheckEngineMock();
    }
}
```

v5/SampleConfig.java

42

Autowiring By Name

Annotations can be used to name beans in two ways:

- Via the 'name' element in '@Bean' on provider methods
- Via the 'value' element of annotations for declaring beans
  - So '@Component("fred")' or '@Component(value="fred")'

Injection is done by using '@Qualifier' on parameters

- E.g. 'public void setThing(@Qualifier("fred") Thing t)'
- '@Qualifier' can be used any number of times

An alternative to '@Qualifier' is '@Resource'

- NB this is a JEE annotation that Spring supports

Autowiring By Name

```
@Configuration
public class SampleConfig {
    @Bean(name = "pricing")
    public PricingEngine foo() {
        return new PricingEngineMock();
    }

    @Bean(name = "payment")
    public PaymentEngine bar() {
        return new PaymentEngineMock();
    }

    @Bean(name = "stock")
    public StockCheckEngine zed() {
        return new StockCheckEngineMock();
    }
}
```

v6/SampleConfig.java

```java
@Configuration
@ComponentScan(basePackages = "demos.spring.notes.v9")
public class SampleConfig { }
```

```java
@Component("payment")
public class PaymentEngineMock implements PaymentEngine {
    // ...
}

@Component("pricing")
public class PricingEngineMock implements PricingEngine {
    // ...
}

@Component("stock")
public class StockCheckEngineMock implements StockCheckEngine {
    // ...
}
```


Autowiring By Name Via @Autowired and @Qualifier

```java
@Service("shop")
public class SampleShop implements Shop {
    @Autowired
    public void setPaymentEngine(
                    @Qualifier("payment") PaymentEngine paymentEngine) {
        this.paymentEngine = paymentEngine;
    }

    @Autowired
    public void setPricingEngine(
                    @Qualifier("pricing") PricingEngine pricingEngine) {
        this.pricingEngine = pricingEngine;
    }

    @Autowired
    public void setStockCheckEngine(
                    @Qualifier("stock") StockCheckEngine stockCheckEngine) {
        this.stockCheckEngine = stockCheckEngine;
    }

    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // As in previous examples
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v6/SampleShop.java

Autowiring By Name Via Single Mentod

```java
@Service("shop")
public class SampleShop implements Shop {
    @Autowired
    public void startup(
            @Qualifier("payment") PaymentEngine paymentEngine,
            @Qualifier("pricing") PricingEngine pricingEngine,
            @Qualifier("stock") StockCheckEngine stockCheckEngine) {
        this.paymentEngine = paymentEngine;
        this.pricingEngine = pricingEngine;
        this.stockCheckEngine = stockCheckEngine;
    }

    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // As in previous examples
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v7/SampleShop.java

INSTIL

47

Autowiring By Name via @Resource

```java
@Service("shop")
public class SampleShop implements Shop {
    @Resource(name = "payment")
    public void setPaymentEngine(PaymentEngine paymentEngine) {
        this.paymentEngine = paymentEngine;
    }

    @Resource(name = "pricing")
    public void setPricingEngine(PricingEngine pricingEngine) {
        this.pricingEngine = pricingEngine;
    }

    @Resource(name = "stock")
    public void setStockCheckEngine(StockCheckEngine stockCheckEngine) {
        this.stockCheckEngine = stockCheckEngine;
    }

    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // As in previous examples
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v8/SampleShop.java

INSTIL

48

Wiring Via Constructors

So far we have always used property injection
- But the annotations work equally well with constructors

Constructor injection is more succinct
- It helps to ensure that all the dependencies of the type are satisfied

Property injection is more explicit
- Plus easier when you have multiple dependencies of the same type

Sometimes the choice is made for you
- E.g. when the dependency is only available at a later time

Autowiring By Type Via Constructors

```
@Service("shop")
public class SampleShop implements Shop {
    @Autowired
    public SampleShop(PricingEngine pricingEngine,
                      PaymentEngine paymentEngine,
                      StockCheckEngine stockCheckEngine) {
        super();
        this.pricingEngine = pricingEngine;
        this.paymentEngine = paymentEngine;
        this.stockCheckEngine = stockCheckEngine;
    }

    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // As in previous examples
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v10/SampleShop.java

Autowiring By Name Via Constructors

```java
@Service("shop")
public class SampleShop implements Shop {
    @Autowired
    public SampleShop(@Qualifier("pricing") PricingEngine pricingEngine,
                      @Qualifier("payment") PaymentEngine paymentEngine,
                      @Qualifier("stock") StockCheckEngine stockCheckEngine) {
        super();
        this.pricingEngine = pricingEngine;
        this.paymentEngine = paymentEngine;
        this.stockCheckEngine = stockCheckEngine;
    }


    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // As in previous examples
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

v11/SampleShop.java

Inserting Additional Behaviour

Your beans may need extra ‘hooks’

- To know when they have just been created
- Or when they are about to be destroyed

Spring supports the JSR250 lifecycle annotations

- Methods annotated with ‘`@PostConstruct`’ are called after Spring has initialized the instance of your bean
- Those annotated with ‘`@PreDestroy`’ are called before an instance is released for Garbage Collection

Note there are more powerful built in abstractions

- Spring lets you register both pre and post processors
- At the level of both bean instances and bean factories

Support For Lifecycle Mewthods

```java
@Service("shop")
public class SampleShop implements Shop {
    @Autowired
    public SampleShop(@Qualifier("pricing") PricingEngine pricingEngine,
                      @Qualifier("payment") PaymentEngine paymentEngine,
                      @Qualifier("stock") StockCheckEngine stockCheckEngine) {
        // As in previous examples
    }

    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // As in previous examples
    }

    @PostConstruct
    public void begin() {
        System.out.println("Spring context just created a shop");
    }

    @PreDestroy
    public void end() {
        System.out.println("Spring context just released a shop");
    }

    // ...
}
```

v12/SampleShop.java

The Lifecycle of a Bean

By default Spring creates singletons

- The same instance is returned, no matter how many times ask for it

But other scopes are supported

- A bean in prototype scope has a new instance created every time
- Spring MVC supports '`request`', '`session`' and '`global session`' scopes

The annotation for managing this is '`@Scope`'

- Note an error when injection requires an impossible mix of scopes

Demonstrating Bean Scopes

```
@Service("shop")
@Scope("prototype")
public class SampleShop implements Shop {
    // ...
}
@Component("payment")
public class PaymentEngineMock implements PaymentEngine {
    // ...
}
@Component("pricing")
@Scope("prototype")
public class PricingEngineMock implements PricingEngine {
    // ...
}
@Component("stock")
public class StockCheckEngineMock implements StockCheckEngine {
    // ...
}
```


Demonstrating Bean Scopes

```
private static void testDependencies(SampleShop shop1, SampleShop shop2) {
    PrintStream out = System.out;
    if (shop1 == shop2) {
        out.println("There is a single shop");
    } else {
        out.println("There are multiple shops");
    }
    if (shop1.getPricingEngine() == shop2.getPricingEngine()) {
        out.println("There is a single pricing engine");
    } else {
        out.println("There are multiple pricing engines");
    }
    if (shop1.getPaymentEngine() == shop2.getPaymentEngine()) {
        out.println("There is a single payment engine");
    } else {
        out.println("There are multiple payment engines");
    }
    if (shop1.getStockCheckEngine() == shop2.getStockCheckEngine()) {
        out.println("There is a single stock check engine");
    } else {
        out.println("There are multiple stock check engines");
    }
}
```

```
There are multiple shops
There are multiple pricing engines
There is a single payment engine
There is a single stock check engine
```

v13/Program.java

Introducing Spring EL

Spring 3 introduced an expression language
- To enable you to inject arbitrary values into beans

This 'Spring EL' can be used in three ways
- Within the XML bean description file
- Via the '@Value' annotation within code
- Dynamically via the 'SpelExpressionParser'

The EL syntax allows you to
- Perform arithmetic and use the ternary conditional
- Call properties, regular methods, and static functions
- Apply pattern matching and manipulate collections

57

The Basic Syntax of Spring EL

Demo	Explanation
`#{ 5 }, #{ 6.7 }, #{ 'dave' }`	Specifying literal values in EL
`#{ myBean }`	Referencing a Spring bean by name
`#{ myBean.address }`	Calling 'getAddress()' on the bean called 'myBean'
`#{ myBean.size * 12 }`	Perform arithmetic on the value of a property
`#{ myBean.size gt 50 }`	Performing comparisons on the value of a property
`#{ myBean.func() }`	Calling 'func()' on the bean called 'myBean'
`#{ myBean.op1().op2().op3() }`	Making chains of method calls
`#{ myBean.op1()?.op2() }`	Making method calls safely by testing for 'null'
`#{ T(com.abc.MyClass).VAL }`	Access a static field of a class
`#{ T(com.abc.MyClass).func() }`	Call a static function of a class
`#{ systemProperties['abc.def']}`	Access the value of a system property

58

The Basic Syntax of Spring EL

Demo	Explanation
`#{ mb.owner == 'dave' and mb.size gt 200 }`	Combine different expressions
`#{ not myBean.inStock }`	Negate an expression
`#{ myBean.size == 20 ? "ab" : "cd" }`	Use the ternary conditional operator
`#{ mb.owner != null ? mb.owner : "cd" }`	Supplying a replacement for a property
`#{ mb.owner != null ?: "cd" }`	A shortcut for the above
`#{ myBean.owner matches '[a-z]{8}' }`	Perform pattern matching
`#{ myList [0] }, #{ myMap['someKey'] }`	Access values in collections
`#{ staff.?[dept == 'IT'] }`	Select items based on a predicate
`#{ staff.^[dept == 'IT'] }`	Select the first item that matches
`#{ staff.$[dept == 'IT'] }`	Select the last item that matches
`#{ staff.![employeeId] }`	Perform a projection on a collection

Demonstrating Spring EL

```
@Component("pricing")
public class PricingEngineMock implements PricingEngine {
    @Value("#{dataSource.discountAmount}")
    public void setMinimumDiscountAmount(double minimumDiscountAmount) {
        this.minimumDiscountAmount = minimumDiscountAmount;
    }

    @Value("#{dataSource.percentageDiscount * 2}")
    public void setPercentageDiscount(int percentageDiscount) {
        this.percentageDiscount = percentageDiscount;
    }

    @Value("#{dataSource.prices}")
    public void setPrices(List<Double> prices) {
        this.prices = prices;
    }

    // ...
}
```

v14/PricingEngineMock.java

Support for JSR 330

For many years there was no formal support for DI in JEE

- In V5 the '@Resource' annotation was added for built in types
- In JSR 330 a new set of annotations were added for general purpose DI

Spring supports these annotations since version three

- '`@Named`' can be used on types to replace '`@Component`'
- Plus also on parameters to replace '`@Qualifier`'
- '`@Inject`' can be used to replace '`@Autowired`'

Support for CDI Annotations

```java
@Named("shop")
public class SampleShop implements Shop {
    @Inject
    public void setPaymentEngine(
            @Named("payment") PaymentEngine paymentEngine) {
        this.paymentEngine = paymentEngine;
    }

    @Inject
    public void setPricingEngine(
            @Named("pricing") PricingEngine pricingEngine) {
        this.pricingEngine = pricingEngine;
    }

    @Inject
    public void setStockCheckEngine(
            @Named("stock") StockCheckEngine stockCheckEngine) {
        this.stockCheckEngine = stockCheckEngine;
    }

    @Override
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        // ...
    }

    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```

cdi.v1/SampleShop.java

Support for CDI Annotations

```
@Named("payment")
public class PaymentEngineMock implements PaymentEngine {
    // ...
}

@Named("pricing")
public class PricingEngineMock implements PricingEngine {
    // ...
}

@Named("stock")
public class StockCheckEngineMock implements StockCheckEngine {
    // ...
}
```

INSTIL

63

Spring AOP

- Aspect Oriented Programming

INSTIL

64

Introducing AOP

Aspect Oriented Programming is a relatively recent idea
- It complements existing paradigms like OO and FP

An **Aspect** is a type of behaviour within your system
- Such as transactions, security checks, logging or thread safety
- This needs to be implemented across many unrelated classes
- Hence Aspects are also described as 'Cross-Cutting Concerns'

Typically, the set of classes/methods needing the aspect keeps changing
- This rules out any solution based around interfaces or base classes

65

The Terminology of AOP

The **Advice** is the implementation of the Aspect
- I.E the code to be run to implement the cross-cutting concern

The **Join Point** is where the Advice is applied
- In AOP join points include method calls, the creation of objects, fields being accessed, and exception handlers being triggered
- In the Spring implementation only method calls are supported

A **Pointcut** is an expression that selects join points
- Just as a regular expression determines matches in a text file or an XPath expression selects nodes in an XML document
- E.g. '`execution(public * *(..))`' selects all public method calls in any class, regardless of the declared parameters

66

Controversy Surrounding AOP

AOP is controversial because of the Weaving
- The mechanism by which the advice is added at the join point

Within the VM the possibilities for weaving are unattractive
- Your source code could be rewritten before compilation
- Your bytecode could be rewritten after compilation
- Proxies for your classes could be created via:
  - Bytecode generation libraries that write classes in the fly
  - Custom class-loaders that supply proxy classes to the VM

Weaving done badly will harm traceability and debugging
- This is the major factor preventing AOP going mainstream

67

How Spring Implements Aspects

Spring is in an ideal position to implement AOP safely
- The whole point of Dependency Injection is that by working through interfaces the client code remains ignorant of the implementation

Spring creates proxies for your bean as required
- Via the support for dynamic proxies within the Reflection API
- These proxies call into the advice before forwarding the method call to the intended target (aka the Advised Object)

No esoteric programming hacks are required
- Making AOP a much less scary prospect

68

Why Spring Implements Aspects

Spring has always supported AOP
- AOP and Dependency Injection are the foundations of the other frameworks

AOP is used for declarative transaction management
- Say one or more of your beans needs a transaction policy
- Spring generates a proxy which is substituted for your bean
  - Calls made against the proxy go through a transaction manager
  - Before the matching call is made against your bean

In Spring 2.0 the AOP support was merged with AspectJ
- This is the leading AOP tool, with excellent Eclipse support
- It provides the standard mini-language for describing pointcuts

The Different Kinds of Advice

Spring supports five kinds of advice
- The difference being when the advice is applied

Before advice is applied before the method call

After advice is applied after the method call
- Regardless of how the method completed
- **AfterReturning** advice is applied if the method returns normally
- **AfterThrowing** advice is applied if an exception is thrown

Around advice wraps around the method call
- Your method takes a '`ProceedingJoinPoint`' parameter
- You use this to signal when the target should be called

Testing in Spring

Part 1: Unit Testing Spring Beans

INSTIL

71

Testing Components in Spring

One of Springs benefits is that it facilitates testing

- Agile processes reply on continuous automated testing

Springs use of DI means that beans are pre-configured for unit testing

- Dummy implementations of dependencies can be injected to separate the class under test from the rest of the system
- The framework itself is not needed during this testing

The framework has built in support for integration testing

- An application context can be used in both JUnit and TestNG
- Special declarations can be created to test groups of beans

INSTIL

72

Integration Testing in Spring

For testing Spring provides the 'TestContext' framework
- This allows a test tool to use a context with minimal coding
- At present JUnit and TestNG are supported

Spring makes use of JUnit's extension points
- '`@RunWith`' allows tests to be run using a custom '`Runner`'
- Spring provides the 'SpringJUnit4ClassRunner' class

Injection is enabled via '`@ContextConfiguration`'
- This instructs Spring to load bean definitions from a file
  - By default this is '`/PackagePath/TestName-Context.xml`'
  - Other locations can be specified via the 'locations' element
- Spring 2.5 annotations are then used to insert dependencies

Integration Testing in Spring

```
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertTrue;
import javax.annotation.Resource;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"/springTesting.xml"})
public class IntegrationTesting {
    @Test
    public void sanity() {
        assertTrue(shop.makePurchase("AB123", 20, "DEF456GHI78"));
    }

    @Resource(name = "shop")
    public void setShop(Shop shop) {
        this.shop = shop;
    }

    // ...
}
```

Introducing Spring Boot

- Simplifying Your Spring Projects

INSTIL

75

Introducing Spring Boot

The breadth of Spring became a major issue

- Spring was supposed to reduce complexity in Enterprise Java
- But the number of libraries and their dependencies increases it

Developers had to play ‘whack a mole’ with JAR files

- Maven and Gradle help with this, but only to a certain extent

Spring Boot exists to solve this problem

- By making it simple to configure Spring based apps

INSTIL

76

Introducing Spring Boot

Spring Boot has two primary goals

- To make it easy to build projects that use a variety of Spring frameworks
- To deploy projects as standalone, self contained systems

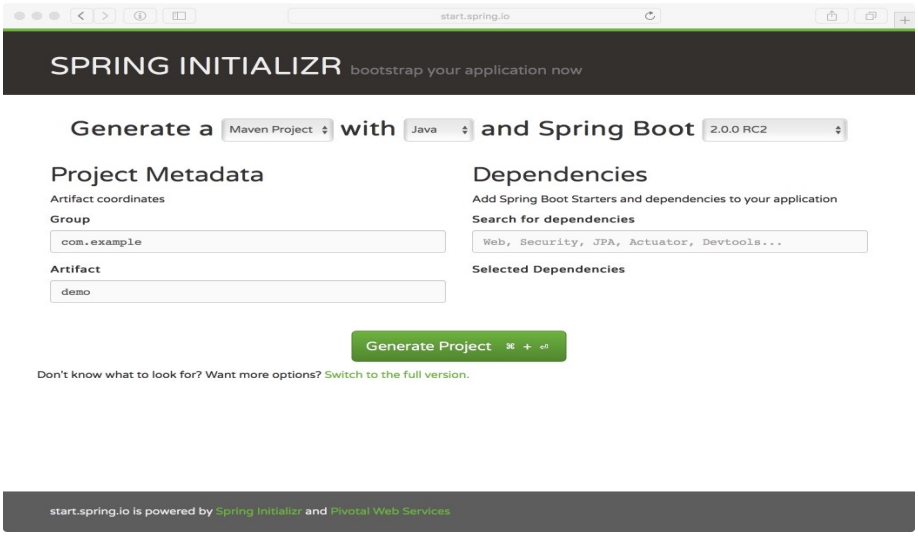
This aligns with the movement away from monoliths

- Towards cloud hosted micro-services
- Sometimes expressed as ‘make JAR not WAR’ ☺

Most people get started via the ‘Spring Initializr’

- An online wizard producing a build file, starter code and config files
- IntelliJ will run this without you needing to leave the IDE

The Spring Initializr



Introducing Spring Boot

Spring Boot makes Spring opinionated

- The framework makes assumptions and uses 'magic values'
- This is sometimes called 'convention over configuration'

You either know the conventions or you don't

- E.g. with Web Applications it looks for static content in the '`public`', '`static`', '`resources`' or '`WEB-INF/resources`' folders

Spring Boot creates an Application Context on your behalf

- An '`AnnotationConfigEmbeddedWebApplicationContext`' is used if it thinks you are creating a Web Application / Microservice
- Again this is inferred based on the modules in use

Working With Spring Boot

A Spring Application can be started by:

- Creating a '`SpringApplication`' object
  - And then invoking its 'run' method with or without args
  - A bean wiring file can be passed in the constructor
- Annotating a class with '`@SpringBootApplication`'
  - And passing its class object to '`SpringApplication.run`'
  - The annotation combines the meanings of '`@Configuration`', '`@EnableAutoConfiguration`' and '`@ComponentScan`'

The application can be administered remotely

- Setting a property causes a built in JMX server to start
- This hosts a instance of '`SpringApplicationAdminMXBean`'

Bootstrapping Spring Boot

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

```
public class Application {
    private static String fileName = "wiring.xml";

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext ctx
                = new SpringApplication(fileName).run(args);
        System.out.println("Hit return to end");
        System.in.read();
        ctx.close();
    }
}
```

INSTIL

81

Traditional Spring MVC

Creating Regular Web Applications

INSTIL

82

Traditional Vs. Modern Spring MVC

Spring MVC was invented before frameworks like Angular and React
- It predates, by many years, the concept of a Single Page Application
- It also predates Cloud Computing and the concept of a Microservice

Originally Spring MVC applications used Server Pages
- To generate the HTML that would be displayed in the browser

Today we typically use Spring MVC to generate RESTful services
- We do not need the parts of Spring MVC supporting Server Pages

Hence the material in this chapter may not be relevant
- It is included so you can appreciate the underlying design
- There are still a lot of legacy Spring MVC apps being maintained

83

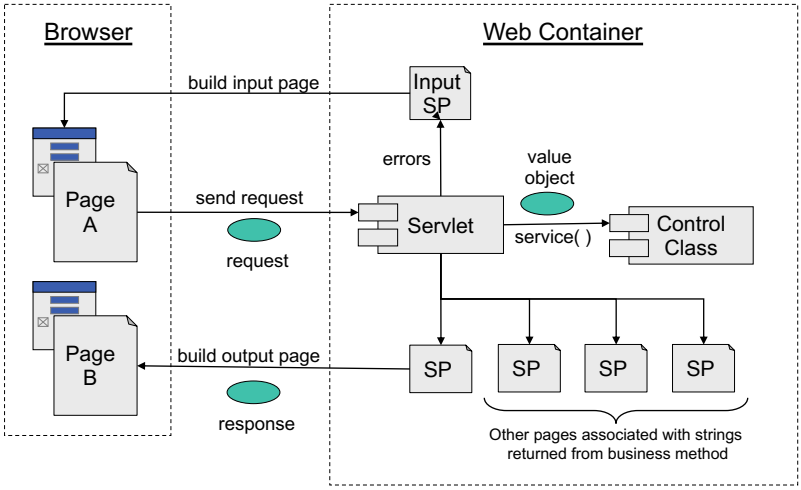
The Spring MVC Framework

Spring MVC is based around the 'Model 2 Architecture'
- Evolved by Sun to combine the best of Servlets and JSP's

In a Model 2 based Web Application:
- A Servlet receives the request and generates a JavaBean
  - Validating and converting the parameters as it does so
- The JavaBean is passed to a controller class
  - This could be an EJB or a plain Java object (POJO)
- The controllers output is passed with the JavaBean to a JSP
  - This takes care of generating the response for the client
  - There may be one JSP for each type of controller response
  - The original JSP may be called if the page contained errors

84

Standard JEE Model 2 Design



85

Model 2 Design and Frameworks

Following Model 2 manually quickly becomes tedious

- You are continually writing Servlets to route requests
- For each request all that changes is:
  - The names of the parameters in the request
  - The name of the JavaBean to be created
  - The name of the controller to pass the bean to
  - The names of the JSP's that build the response

However in reality only a single Servlet is needed

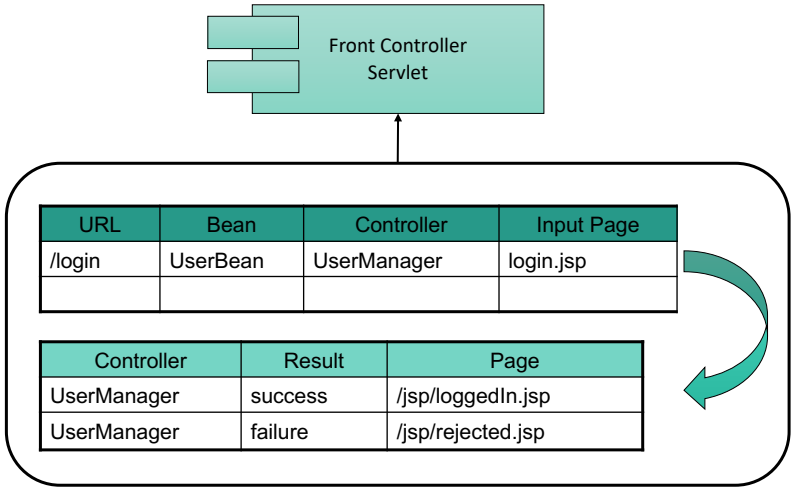
- For each request this 'Master Servlet' reads the name of the JavaBean, Controller etc… from its own special config file

This is known as a 'Front Controller' Servlet

- Struts 1, Struts 2, Spring MVC and JSF all provide one

86

The JEE Front Controller Pattern

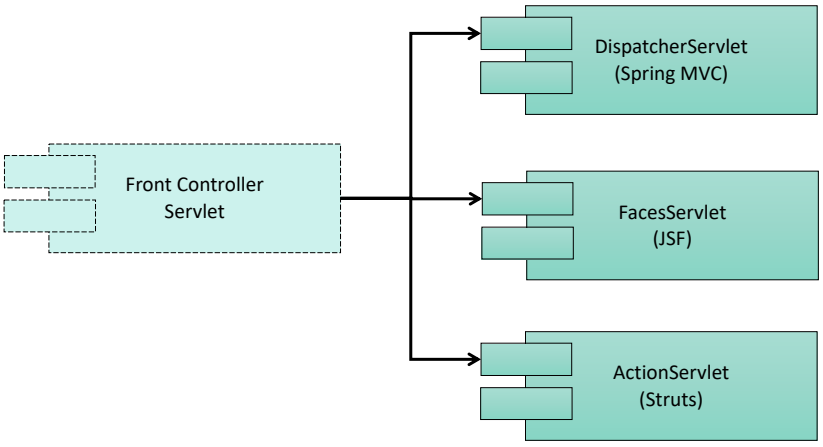


URL	Bean	Controller	Input Page
/login	UserBean	UserManager	login.jsp

Controller	Result	Page
UserManager	success	/jsp/loggedIn.jsp
UserManager	failure	/jsp/rejected.jsp

87

The JEE Front Controller Pattern



88

The Spring MVC Front Controller

Spring MVC provides a 'Dispatcher Servlet'

- This acts as a 'Front Controller' for incoming requests
- It is configured in 'web.xml' in the normal way

The servlet name you assign is very important

- It will be used to determine the path to the config file
- E.g. if you configure the servlet with the name 'fred' then on startup it will look for '/WEB-INF/fred-servlet.xml'

Often you don't want Spring to handle all requests

- By convention the Dispatcher Servlet is mapped to '*.htm'
- In a RESTful app you would choose something less intrusive to enable meaningful URL's (e.g. /app/*)

How the Dispatcher Servlet Works

The 'Dispatcher Servlet' does very little work

- Instead it delegates to other components

There are three main types of component:

- A 'Handler Mapping' associates URL's with controllers
  - The input is a URL and the output is a class name
- A 'View Resolver' associates the return values of business methods with paths to view pages (JSP, FreeMarker etc…)
  - The input is a string and the output a path to a file
- An 'Exception Resolver' handles uncaught exceptions

Other component types exist

- E.g. for selecting a locale and applying UI themes

How the Dispatcher Servlet Works

Components needed by Spring MVC are autowired

- When the dispatcher servlet starts up it examines the configuration file for types that implement certain interfaces
  - As this is 'autowiring by type' you don't need to use id's
- At a minimum you must provide an implementation of the handler mapping and view resolver interfaces

This is how Spring MVC provides flexibility

- Each component type has many built-in implementations
- Some implementations integrate other frameworks
  - E.g. the JSTL, Tiles, FreeMarker and Velocity
- You can mix-and-match or write your own as required

Basic Annotations for Spring MVC

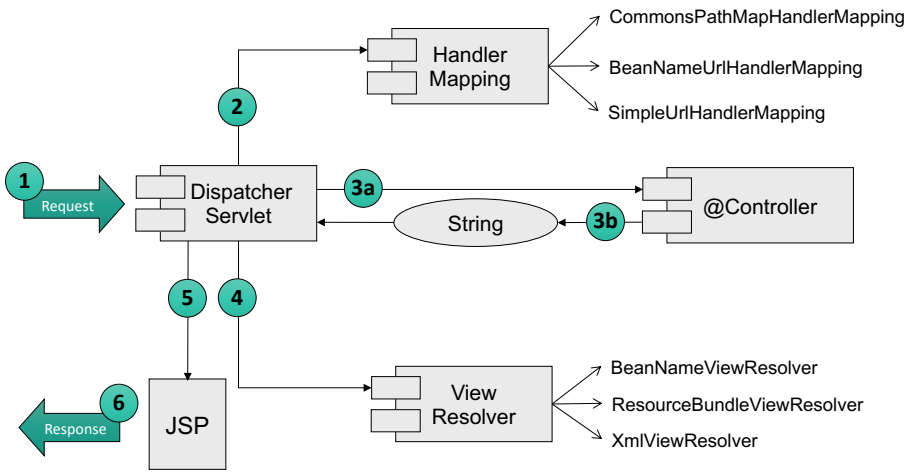
Annotation	Meaning
`@Controller`	Spring will use the class as a handler for one or more HTTP requests when the component-scan schema extension is used
`@RequestMapping`	Specifies the URL and/or HTTP request type NB: Can be attached to the class or individual methods
`@ModelAttribute`	Signifies that a JavaBean method parameter is a 'command object' and hence should be loaded with the requests data
`@RequestParam`	Signifies that a method parameter should be initialized based on the value of a single parameter in the HTTP request
`@SessionAttributes`	Specifies which entries in the 'Model And View' should be added to the HTTP Session, in addition to being added to the view
`@ResponseBody`	Signifies that the string returned from a controller method should be used directly as the body of the response (instead of being converted into the path to a view page)

The Lifecycle of a Request in MVC

1. The 'Dispatcher Servlet' receives most requests
   - As in other frameworks it acts as a Front Controller
   - By convention the '.htm' postfix is used
2. The URL is passed to a 'Handler Mapping'
   - This associates the URL of the request with a controller
3. The controllers business method is triggered
   - This traditionally returns a 'Model And View' object
   - In the new implementation it returns a string
4. The view name is passed to a 'View Resolver'
   - This associates the view name with the path to a server page
5. The request is sent on to the server page
   - Where the model data from the 'Model And View' is available

93

The Lifecycle of a Request in MVC



94

A Modern Spring MVC Controller

The next slide shows a Spring MVC Controller

- '`@Controller`' marks the class as an MVC controller
  - It will be picked up automatically via component scanning
- '`@RequestMapping`' specifies the URL
  - This can be the entire URL or just the base part
- '`@SessionAttributes`' is for state management
  - When a name listed in the annotation matches the name of an item in the model then that item is put in the session
- The constructor is '`@Autowired`'
  - Any DI technique can be used inside a MVC controller

95

An Annotation Driven Spring MVC Controller

```
@Controller
@RequestMapping("/search.htm")
@SessionAttributes({"flightsOut", "flightsBack"})
public class SearchController {
    @Autowired
    public SearchController(
               FlightSearchEngine flightSearchEngine) {
        this.flightSearchEngine = flightSearchEngine;
    }
    @SuppressWarnings("unchecked")
    @RequestMapping(method = RequestMethod.POST)
    public String op1(SearchBean bean,
                      BindingResult result,
                      Map model) throws Exception {
        // Rest of method omitted
        return "outwardsFlightSelection";
    }

    private FlightSearchEngine flightSearchEngine;
}
```

SearchController.java

96

More Detail on @RequestMapping

'`@RequestMapping`' can be used in several ways
- Depending on how complex the controller will be

Option 1 (URL on class):
- Annotate the class with '`@RequestMapping("/someURL")`'
- Annotate the method(s) with '`@RequestMapping(method=GET)`'

Option 2 (two part URL):
- Annotate the class with '`@RequestMapping("/part1")`'
- Annotate the methods with '`@RequestMapping(value="/part2")`'

Option 3 (URL on methods):
- Annotate the methods with '`@RequestMapping("/someURL")`'

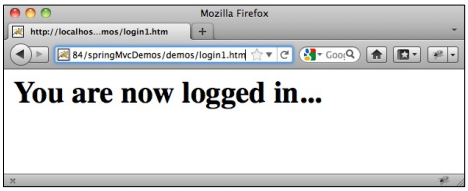
Controller Methods in Detail

There are many ways of writing a controller method
- The parameters and return types are recognized via annotations and 'convention over configuration'
- There is no definitive set of supported method signatures
  - This can be confusing until you get used to it

The most basic takes nothing and returns a string
- By default the string will be mapped to a path to a view
  - Using the view resolver(s) declared in the configuration file
- If you annotate the string with '`@ResponseBody`' then its content will be used directly as the body of the response
  - We will revisit this concept when talking about REST

```
<a href="demos/login1.htm">
    The 'Hello World' controller method
</a>
```

```
@Controller
@RequestMapping("/demos")
public class LoginController {
    @RequestMapping(value = "/login1.htm", method = GET)
    public @ResponseBody String demo1() {
        System.out.println("Demo 1 method called");
        return "<h2>You are now logged in...</h2>";
    }
}
```



INSTIL

99

Capturing Individual Parameters

It is possible to store each parameter from the request inside its own parameter in the controller method

- E.g. '`public String func(@RequestParam("foo") String foo)`'

There are several 'gotchas' to watch out for:

- You can often omit the element on the annotation
  - In which case the method parameter name is used instead
  - This only works when your classes contain debugging info
    - Since parameter names are not available via reflection
- You can omit the annotation entirely
  - Spring defaults to matching 'foo' against a request param
  - It is probably best not to rely on this and be explicit instead

INSTIL

100

```
<a href="demos/login2.htm?username=Dave&password=wn1hgb">
    Reading parameters individually from the URL
</a>
```

```
@RequestMapping(value = "/login2.htm", method = GET)
public @ResponseBody
    String demo2(@RequestParam("username") String username,
                 @RequestParam("password") String password) {
    System.out.println("Demo 2 method called");
    return "<h2>Welcome back: " + username +
           " with password " + password + "</h2>";
}
```

Making Use of the Servlet API

Spring will inject JEE web classes for you

- E.g. if your method declares an '`HttpServletRequest`' parameter then Spring points it at the current request
- This also applies to the response and session objects

It is a best practice to avoid doing so

- In order to enable your controllers to support unit testing

At the lowest level you can work with streams

- An '`InputStream`' or 'Reader' lets you read from the request
- An '`OutputStream`' or 'Writer' lets you write to the response

```
<a href="demos/login3.htm?username=Dave&password=wn1hgb">
    Using the raw <i>request</i> and <i>response</i> objects
</a>
```

```
@RequestMapping(value = "/login3.htm", method = GET)
public void demo3(HttpServletRequest request,
                  HttpServletResponse response) {
    System.out.println("Demo 3 method called");
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    response.setContentType("text/html");
    PrintWriter writer;
    try {
        writer = response.getWriter();
        writer.println("<h2>Welcome back: " + username +
                       " with password " + password + "</h2>");
    } catch (IOException ex) {
        throw new IllegalStateException("IO Failure: " + ex);
    }
}
```


Working With JavaBeans

The most common first parameter is a JavaBean

- Spring will assume you want the bean populated
- By matching properties to parameters from the request

This applies to both GET and POST requests

- Typically, the input will be from a form which is being POST'ed

There are several options for validation

```
<a href="demos/login4.htm?username=Dave&password=wn1hgb">
    Loading parameters into a JavaBean
</a>
```

```
@RequestMapping(value = "/login4.htm", method = GET)
public @ResponseBody String demo4(User theUser) {
    System.out.println("Demo 4 method called");
    return "<h2>Welcome back: " + theUser.getUsername() +
           " with password " + theUser.getPassword() +
           "</h2>";
}
```

INSTIL

105

A JavaBean To Hold Request Parameters

```
public class User {
    public User() {
        super();
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    private String username;
    private String password;
}
```

User.java

INSTIL

106

```
<div id="loginForm1" style="margin-left: 4em;">
    <form action="demos/login5.htm" method="POST">
        Username:<input name="username" type="text“
                        size="10" value="Dave"/><br>
        Password:<input name="password" type="text“
                        size="10" value="wn1hgb"/><br>
        <input type="submit" value="Login"/>
    </form>
</div>
```

```
@RequestMapping(value = "/login5.htm", method = POST)
public @ResponseBody String demo5(User theUser) {
    System.out.println("Demo 5 method called");
    return "<h2>Welcome back: " + theUser.getUsername() +
           " with password " + theUser.getPassword() +
           "</h2>";
}
```

INSTIL

107

Support for Path Parameters

Spring MVC 3 supports ‘Path Parameters’

This is where components of the URL itself contain data
- E.g. ‘`http://megacorp.com/newOrder/AB12/belfast`’

Path parameters are described using ‘URL Templates’
- E.g. ‘`/newOrder/{client_id}/{client_office_location}`’

To use path parameters in your method
- Specify the URL Template in ‘`@RequestMapping`’

Annotate one or more parameters with ‘`@PathVariable`’
- The value of ‘`@PathVariable`’ must match an id in the URL
- As with request parameters you can rely on the variable name as long as you will never compile without debug information

INSTIL

108

```
<a href="demos/dave/wn1hgb/login6.htm">
    Reading parameters from the URL path
</a>
```

```
@RequestMapping(value = "/{username}/{password}/login6.htm",
                method = GET)
public @ResponseBody String demo6(
          @PathVariable("username") String username,
          @PathVariable("password") String password) {
    System.out.println("Demo 6 method called");
    return "<h2>Welcome back: " + username +
           " with password " + password + "</h2>";
}
```

Other Options for Parameters

It is possible to capture headers from the request

- Using the '`@RequestHeader`' annotation
- This is important in RESTful applications

Cookie values can also be extracted

- Using the '`@CookieValue`' annotation

The entire body of the request can be captured

- By using '`@RequestBody`' on a string parameter
- E.g. '`String op1(@RequestBody String body)`'

In V3 you can use converters to process the body

- Your parameter could be a byte array or an XML document

```
<a href="demos/login7.htm">
    Reading headers from the request
</a>
```

```
@RequestMapping(value = "/login7.htm", method = GET)
public @ResponseBody String demo7(
            @RequestHeader("Accept") String acceptHeader) {
    System.out.println("Demo 7 method called");
    return "<h2>Value of accept header is: " +
           "<span style='color:blue'>" + acceptHeader +
           "</span></h2>";
}
```


```
<a href="demos/login8.htm">
    Reading cookies from the request
        (relies on session existing)
</a>
```

```
@RequestMapping(value = "/login8.htm", method = GET)
public @ResponseBody String demo8(
            @CookieValue("JSESSIONID") String sessionCookie) {
    System.out.println("Demo 8 method called");
    return "<h2>Value of session ID cookie is: " +
           "<span style='color:blue'>" + sessionCookie +
           "</span></h2>";
}
```

```
<div id="loginForm2" style="margin-left: 4em;">
    <form action="demos/login9.htm" method="POST">
        Username:<input name="username" type="text“
                        size="10" value="Dave"/><br>
        Password:<input name="password" type="text“
                        size="10" value="wn1hgb"/><br>
        <input type="submit" value="Login"/>
    </form>
</div>
```

```
@RequestMapping(value = "/login9.htm", method = POST)
public @ResponseBody String demo9(
            @RequestBody String theBody) {
    System.out.println("Demo 9 method called");
    return "<h2>Value of body is: <span style='color:blue'>" +
           theBody + "</span></h2>";
}
```

INSTIL

113

Introduction to Microservices

- Why adopt Microservices?

INSTIL

114

The Move to Microservices

Enterprise Architecture is in transition
- Change is occurring across several dimensions

We no longer host our own servers
- Instead we make use of Cloud Providers

We no longer deploy applications as a unit
- Instead we break then into distinct services

Non-functional requirements are ever more important
- As wider access increases scalability and security concerns

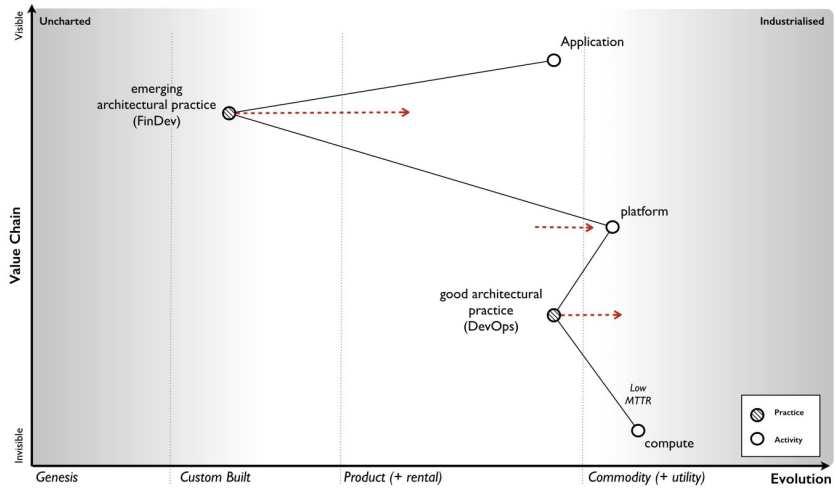
We are exploring Serverless Architectures

115

Evolving Architectures

116

Evolving Architectures



Source: *https://medium.com/hackernoon/why-the-fuss-about-serverless-4370b1596da0*

Breaking Up Monoliths

A Reactive approach implies a non-monolithic architecture

- It's hard to grow and shrink different aspects of the system when all the aspects are built and deployed together as an indivisible unit

So we may wish to break up our monolith into microservices

- As ever if it isn't broken why try to fix it?

Doing this incorrectly is guaranteed to cause disaster

- We cannot partition a codebase into arbitrary chunks
- Coupling and cohesion always need to be considered

The partitioning strategy we use must be effective

- This is where Domain Driven Design comes in

Breaking Up Monoliths

Logically we want to:

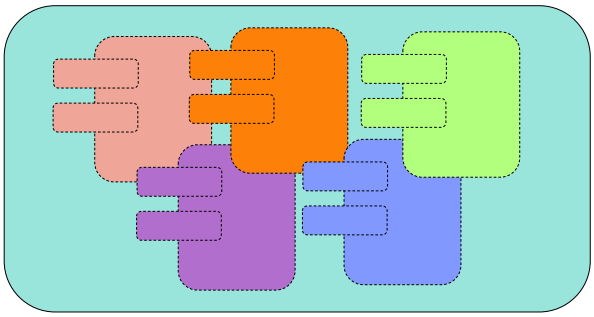
- Identify the different subsystems within the codebase
- Separate these out so they could be deployed separately
- Incrementally give each subsystem its own existence

There are many ways to achieve this:

- Do we rewrite the subsystems or copy them from the monolith?
- Do we keep the functionality in the monolith as a fallback?

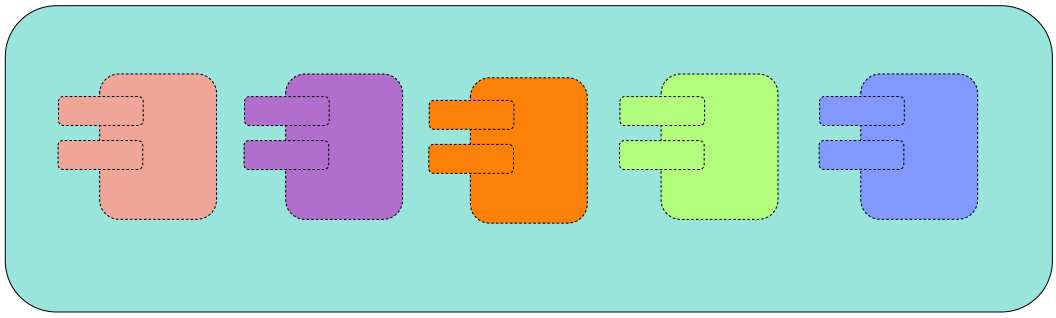
119

Breaking Up Monoliths – Stage 1



120

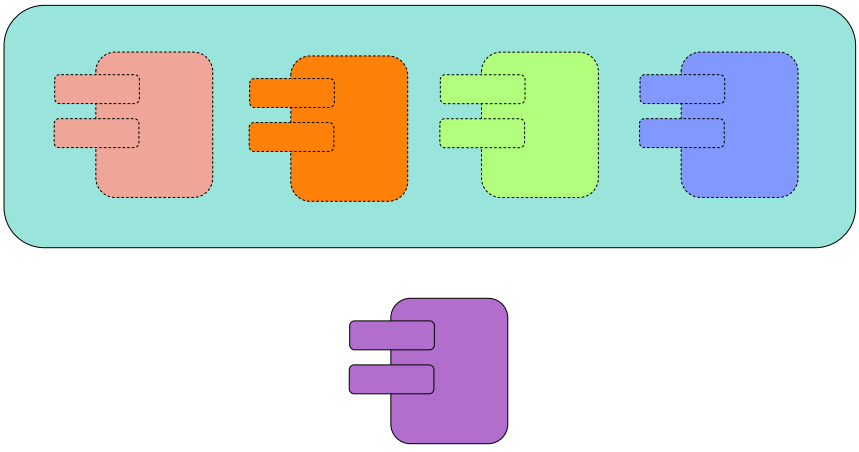
Breaking Up Monoliths – Stage 2



INSTIL

121

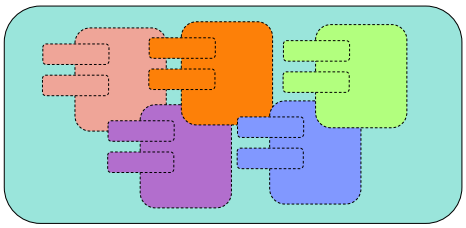
Breaking Up Monoliths – Stage 3

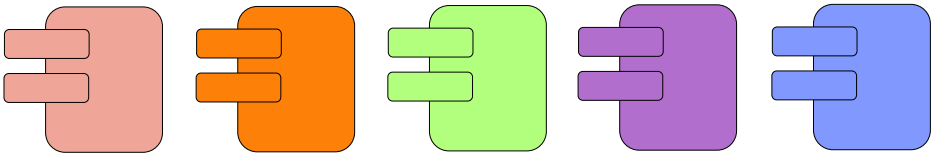


INSTIL

122

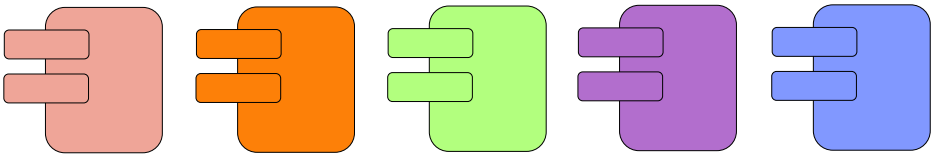
Breaking Up Monoliths – Stage 4





123

Breaking Up Monoliths – Stage 5



124

The problem with Microservices

https://www.youtube.com/watch?v=V3NQt5KP7EM

125

Current Microservice Frameworks

What is the correct size for a Microservice?

- This is a famous (and typically unproductive) talking point

The original vision for Microservices was not always realized

- Frameworks like Spring Boot encouraged a 'kitchen sink' approach
- As opposed to the leaner philosophy represented by Dropwizard

Newer frameworks have reversed this trend

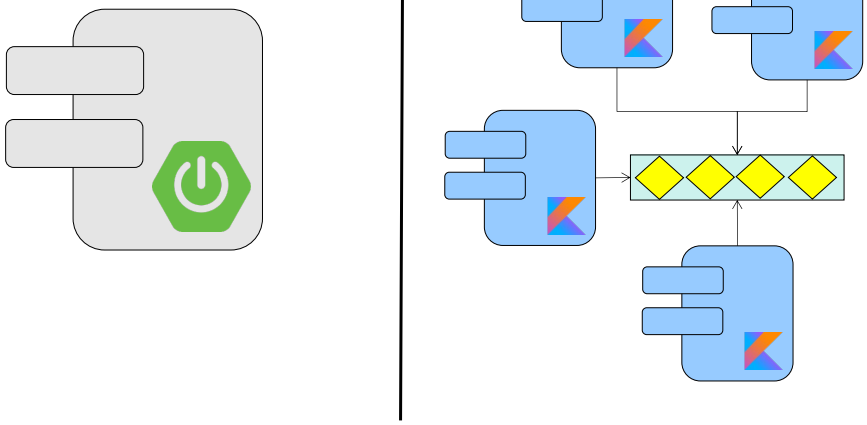
- Ktor, HTTP4K etc... are designed for Micro-Microservices ☺
- Part of this is to enable them to be combined with Serverless

There are now many Microservices frameworks

- Each with their own unique selling points

126

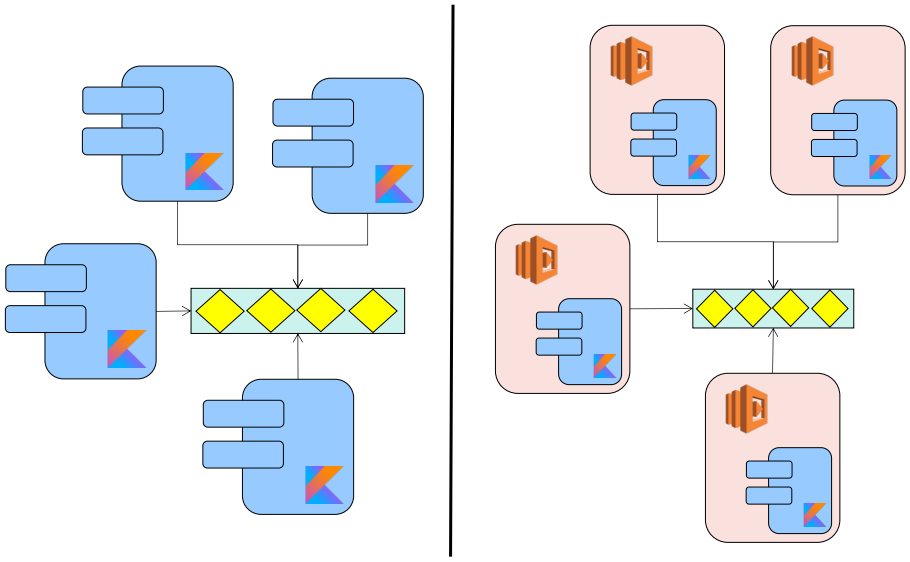
Current Microservice Frameworks



INSTIL

127

Current Microservice Frameworks



INSTIL

128

Current Microservice Frameworks

Dropwizard is a Java framework for developing ops-friendly, high-performance, RESTful web services.

Dropwizard pulls together **stable**, **mature** libraries from the Java ecosystem into a **simple**, **light-weight** package that lets you focus on *getting things done*.

Dropwizard has *out-of-the-box* support for sophisticated **configuration**, **application metrics**, **logging**, **operational tools**, and much more, allowing you and your team to ship a *production-quality* web service in the shortest time possible.

General

- Javadoc
- Release Notes
- Security
- Frequently Asked Questions

Getting Started

- Getting Started

Spring Boot 2.2.4

Overview | Learn | Samples

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

You can also join the Spring Boot community on Gitter!

Current Microservice Frameworks

Ktor

Easy to use, fun and asynchronous.

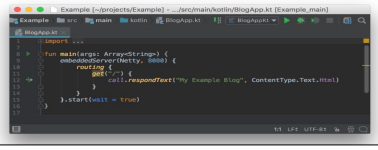
Home / Quick Start / Servers / Clients / Kotlinx / Samples / Advanced

Connected Applications with Kotlin

Ktor is a framework for building asynchronous servers and clients in connected systems using the powerful Kotlin programming language. This website provides a complete reference to the Ktor application structure and programming interface. And how to approach particular tasks.

Here is the place to find the answers you are looking for and learn all about how it works. Search for information or explore the sections below to get started.

Search... (press 's' to focus, or '#' for sections)



HTTP4K

http4k is a lightweight but fully-featured HTTP toolkit written in pure Kotlin that enables the serving and consuming of HTTP services in a functional and consistent way. http4k applications are just Kotlin functions which can be mounted into a running backend. For example, here's a simple echo server.

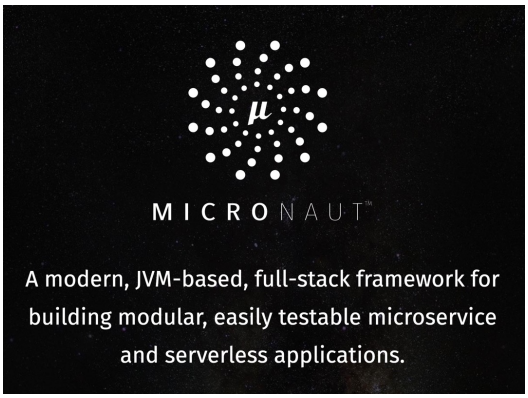
```
val app: HttpHandler = { request: Request -> Response(OK).body(request.body) }
val server = app.asServer(SunHttp(8000)).start()
```

http4k consists of a core library, `http4k-core`, providing a base HTTP implementation + a number of capability abstractions (such as servers, clients, templating, websockets etc). These capabilities are then implemented in add-on modules.

The principles of http4k are:

- **Application as a Function:** Based on the Twitter paper "Your Server as a Function", all HTTP services can be composed of 2 types of simple function:
  - *HttpHandler*: `(Request) -> Response` - provides a remote call for processing a Request.
  - *Filter*: `(HttpHandler) -> HttpHandler` - adds Request/Response pre/post processing. These filters are composed to make stacks of reusable behaviour that can then be applied to an `HttpHandler`.
- **Immutability:** All entities in the library are immutable unless their function explicitly disallows this.
- **Symmetric:** The `HttpHandler` interface is identical for both HTTP services and clients. This allows for simple offline testability of applications, as well as plugging together of services without HTTP container being required.
- **Dependency-lite:** Apart the from Kotlin StdLib, `http4k-core` module has ZERO dependencies and weighs in at ~700kb. Add-on modules only have dependencies required for specific implementation.
- **Testability** Built by **TDD** enthusiasts, so supports **super-easy** mechanisms for both In and Out of Container testing of:
  - individual endpoints
  - applications
  - websockets
  - full suites of microservices
- **Modularity:** Common behaviours are abstracted into the `http4k-core` module. Current add-on modules provide:

Current Microservice Frameworks



131

Microservices and Events

Software Engineering is increasingly focused around events
- But there is confusion about events vs. messages vs. actors vs. services

Both *Event Based* and *Message Driven* designs are asynchronous
- Messages are sent to a clearly identified destination
- Events originate from a clearly identified source
- A message can contain an event as its payload

Event Storming is a process for discovering the Domain Model
- But this does not constrain the technologies selected for the design

Event Sourcing is a specific type of architecture
- Where all state is derived from the sequence of events that occurred

132

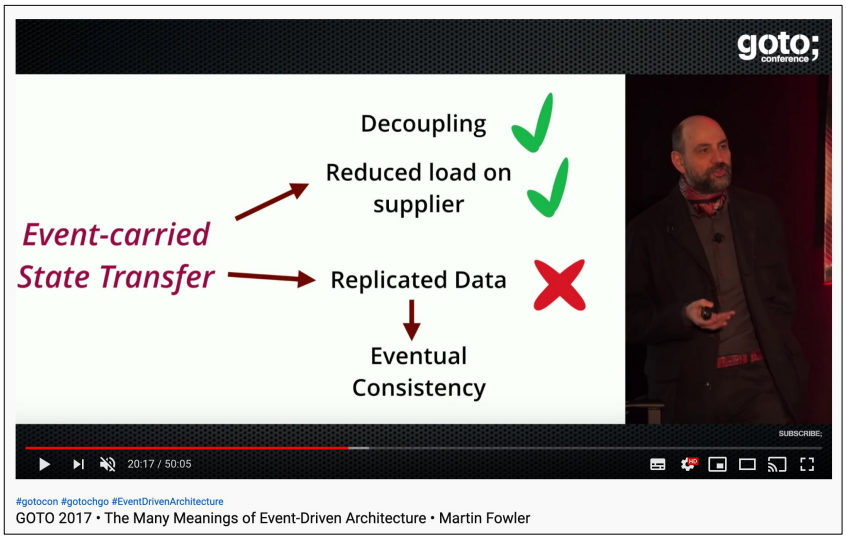
Microservices and Events

Event Sourcing is a specific type of architecture
- Where all state is derived from the sequence of events that occurred
- E.g. your bank balance is derived from all the deposits and withdrawals

There are no updates or deletes in event sourcing
- Instead we append new events into our database
- We store all the facts and derive state from them

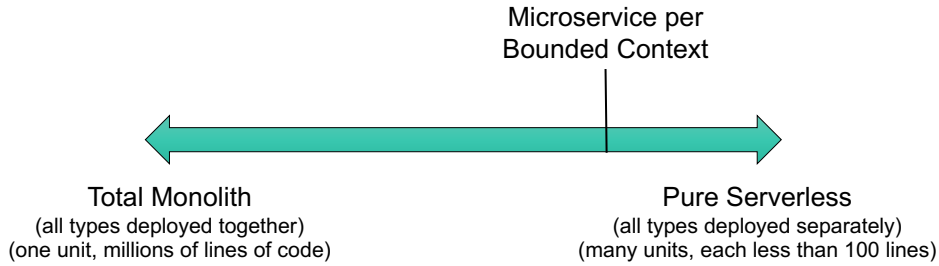
Event sourcing relies on the falling cost of data storage
- Most companies entire data storage needs can fit on a single drive

Microservices and Events



https://www.youtube.com/watch?v=STKCRSUsyP0

Cloud Deployment vs. Cloud Native



135

Principles of Serverless Development

Configuration is everything
- The platform handles scaling

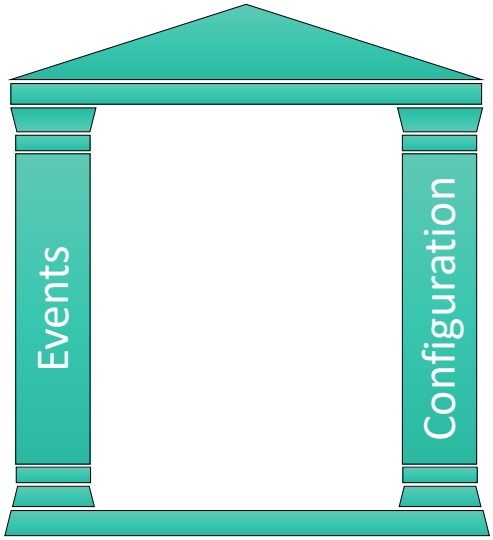
Events are the new objects
- The core building blocks

Code becomes just glue
- That sticks together services

Functions are always stateless
- Making FP the new paradigm

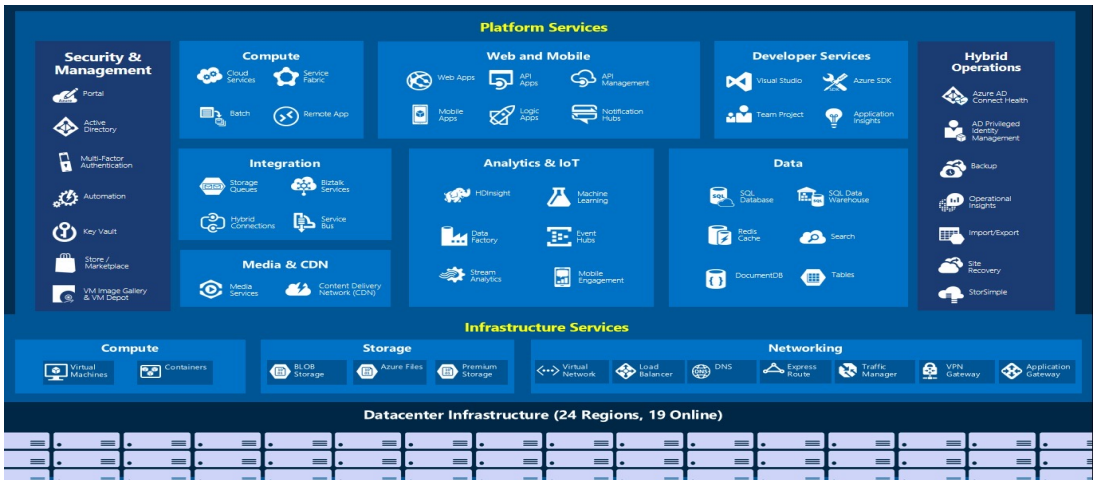
136

Principles of Serverless Development



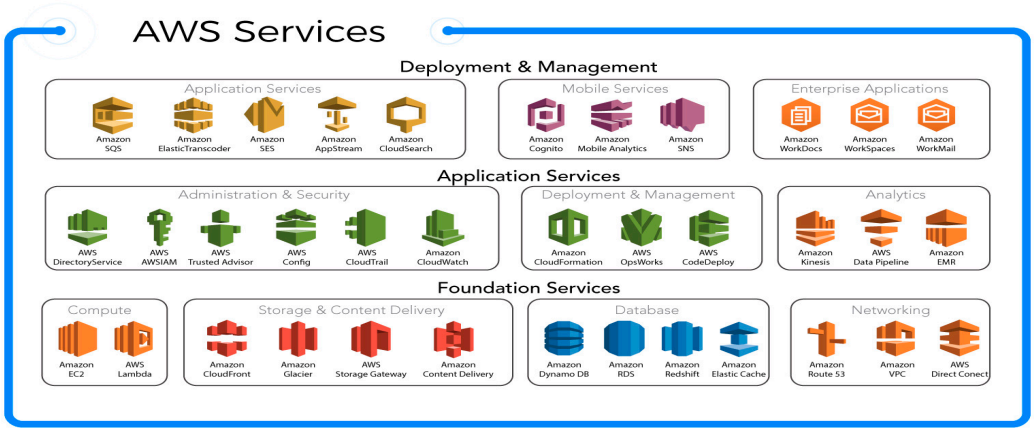
137

Cloud Deployment vs. Cloud Native



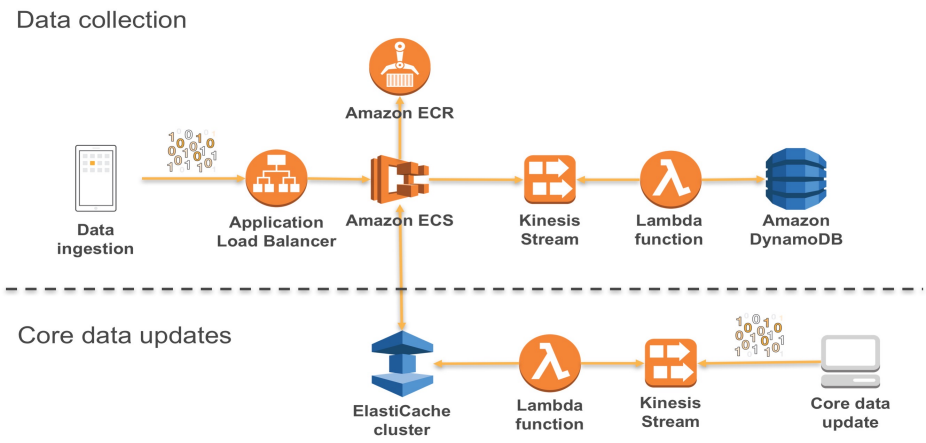
138

Cloud Deployment vs. Cloud Native



139

AWS Architecture for an Ad-Tracking System



140

REST Services

- Building Microservices with Spring MVC

INSTIL

141

Applying REST with Spring MVC

Spring MVC 3 was designed to enable REST

- The framework supports the full range of HTTP Verbs

Objects are automatically converted for you

- If your method returns '`@ResponseBody Course`' then it will be encoded as JSON (via Jackson) or XML (via JAXB) based on the 'Accept' header
- The same applies to method parameters
  - E.g. if it takes a param of type '`@RequestBody` Course'

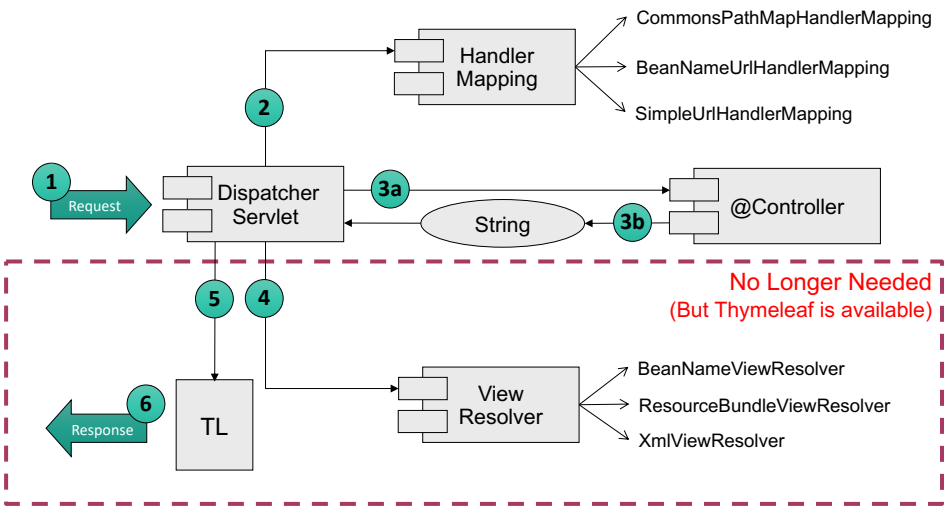
The framework has been improved incrementally

- '`@RestController`' was added for convenience
- As were annotations for each HTTP verb (e.g. '`@GetMapping`')

INSTIL

142

The Lifecycle of a Request in MVC



143

Creating A Sample Service

Lets create a service in three iterations:

- Iteration one will be a basic, but complete, service
- Iteration two will add support for errors and exceptions
- Iteration three will add some more modern features
  - Which became available with Spring MVC 4.x

The service provides CRUD operations in training courses

- We will use a plain JavaScript client to demo the functionality
- You can also use a universal REST client like 'Postman'

144

The AngularJS Front End

145

The Design of the RESTful Service

We want to be able to retrieve all courses:

- A request sent to '`/courses`' should retrieve either HTML or JSON depending on the '`Accept`' header

We want to retrieve full details for a single course

- A request sent to '`courses/{id}`' should retrieve either HTML or JSON depending on the value of 'Accept'

We want to be able to delete courses

- Via a DELETE request sent to '`/courses/{id}`'

We want to be able to create and update courses

- Via a PUT request sent to '/courses'
- The body of the request should be JSON

146

Creating a Basic RESTful Service

Our service is just a class decorated with '@RestController'

- This is equivalent to '@Controller', except that return values will automatically be marshalled and sent to the client via Jackson or JAXB
- The type of marshalling will depend on the 'Accept' header

```
@RestController
@RequestMapping("/courses")
public class CoursesController  {

}
```


147

Creating a Basic RESTful Service

Web methods are again decorated with '@RequestMapping'

- This has a number of different elements you can set
- Most important is 'method', which maps to the HTTP verb

The 'produces' element corresponds to the 'Accept' header

- You use this to distinguish between (for example) methods tailored to produce XML and those tailored for JSON

The 'consumes' element corresponds to 'Content-Type'

- So you can have one method that receives an XML payload, one for JSON, one for form parameters etc…

148

Creating a Basic RESTful Service

Note that '`@RequestMapping`' also has a 'header' element

- Allowing you to invoke a particular method based on an arbitrary combination of headers and header values

'`@RequestBody`' marks a parameter as special

- The body of the request will be un-marshalled and inserted
- Once again via Jackson, JSONP or JAXB

'`@PathVariable`' also marks a parameter as special

- A part of the URL will be extracted and inserted

149

The Iteration 1 Controller

```
@RestController
@RequestMapping("/courses")
public class CoursesController  {
    @RequestMapping(method = RequestMethod.GET, produces="application/xml")
    public CourseList viewAllAsXml() {
        return new CourseList(portfolio.values());
    }
    @RequestMapping(method = RequestMethod.GET, produces="application/json")
    public Collection<Course> viewAllAsJson() {
        return portfolio.values();
    }
    @RequestMapping(value="/{id}", method = RequestMethod.PUT,
                    consumes="application/json")
    public Course addOrUpdateCourseViaJson(@RequestBody Course newCourse) {
        portfolio.put(newCourse.getId(),newCourse);
        return newCourse;
    }
    @RequestMapping(value="/{id}", method = RequestMethod.PUT,
                    consumes="application/xml")
    public String addOrUpdateCourseViaXml(@RequestBody Course newCourse) {
        portfolio.put(newCourse.getId(),newCourse);
        return newCourse.getId();
    }
```

Marshalled to XML via JAXB

Marshalled to JSON via Jackson

Marshalled from JSON via Jackson

Marshalled from XML via JAXB

CoursesController.java

150

The Iteration 1 Controller

```
    @RequestMapping(value="/{id}", method = RequestMethod.GET,
                    produces="application/json")
    public Course fetchCourseDetailsAsJson(@PathVariable("id") String id) {
        return portfolio.get(id);
    }
    @RequestMapping(value="/{id}", method = RequestMethod.GET,
                    produces="application/xml")
    public Course fetchCourseDetailsAsXml(@PathVariable("id") String id) {
        return portfolio.get(id);
    }
    @RequestMapping(value="/{id}", method = RequestMethod.DELETE)
    public void deleteACourse(@PathVariable("id") String id) {
        portfolio.remove(id);
    }
    @Resource(name="portfolio")
    public void setPortfolio(Map<String,Course> portfolio) {
          this.portfolio = portfolio;
    }
    private Map<String,Course> portfolio;
}
```

Path variable automatically extracted from the URL

CourssController.java

Extending Our Service To Manage Errors

Our first iteration ignored situations where:

- We only want to return a status code like '204 No Content'
- We need to return an error code and custom message
- We need to handle exceptions thrown by web methods

Methods returning void can use '`@ResponseStatus`'

- This specifies the return code to be used in all circumstances

Other methods can return a '`ResponseEntity`'

- This is a generic type that wraps the original return value
- It can be created with a response code, a list of headers and a body

Extending Our Service To Manage Errors

One controller can act as an advisor to others
- In a manner similar to advice in Spring AOP

The controller must be decorated with '`@ControllerAdvice`'
- So it can intercept requests and responses from others

Error handers are decorated with '`@ExceptionHandler`'
- Where the 'value' element gives the type of the exception
- The input to the method should be the exception that was thrown
- The return value is marshalled as normal

The Iteration 2 Controller

```
@RestController
@RequestMapping("/courses")
public class CoursesController  {

    @RequestMapping(method = RequestMethod.GET, produces="application/xml")
    public ResponseEntity<CourseList> viewAllAsXml() {
          if(portfolio.isEmpty()) {
             return new ResponseEntity<>(NOT_FOUND);
          } else {
             return new ResponseEntity<>(new CourseList(portfolio.values()), OK);
          }
    }

    @RequestMapping(method = RequestMethod.GET, produces="application/json")
    public ResponseEntity<Collection<Course>> viewAllAsJson() {
          if(portfolio.isEmpty()) {
         return new ResponseEntity<>(NOT_FOUND);
          } else {
             Collection courses = portfolio.values();
             HttpHeaders headers = new HttpHeaders();
             headers.set("NumCourses", String.valueOf(courses.size()));
             return new ResponseEntity<>(courses, headers, OK);
          }
    }
```

Manually setting the HTTP response code

Specifying custom headers

CoursesController.java

The Iteration 2 Controller

```
@ResponseStatus(NO_CONTENT)
@RequestMapping(value="/{id}", method = RequestMethod.PUT,
                consumes="application/json")
public void addOrUpdateCourseViaJson(@RequestBody Course newCourse) {
    portfolio.put(newCourse.getId(),newCourse);
}

@ResponseStatus(NO_CONTENT)
@RequestMapping(value="/{id}", method = RequestMethod.PUT,
                consumes="application/xml")
public void addOrUpdateCourseViaXml(@RequestBody Course newCourse) {
    portfolio.put(newCourse.getId(),newCourse);
}

@RequestMapping(value="/{id}", method = RequestMethod.GET,
                produces="application/json")
public ResponseEntity<Course> fetchCourseDetailsAsJson(@PathVariable("id")
                String id) {
  if(portfolio.containsKey(id)) {
     return new ResponseEntity<>(portfolio.get(id), OK);
  } else {
     return new ResponseEntity<>(NOT_FOUND);
  }
}
```

Directly specifying response code

CoursesController.java

155

The Iteration 2 Controller

```
@RequestMapping(value="/{id}", method = RequestMethod.GET,
                produces="application/xml")
public ResponseEntity<Course> fetchCourseDetailsAsXml(
             @PathVariable("id") String id) {
  if(portfolio.containsKey(id)) {
     return new ResponseEntity<>(portfolio.get(id), OK);
  } else {
     return new ResponseEntity<>(NOT_FOUND);
  }
}

@RequestMapping(value="/{id}", method = RequestMethod.DELETE)
public ResponseEntity<String> deleteACourse(@PathVariable("id")
                                            String id) {
      if(portfolio.containsKey(id)) {
         if(portfolio.get(id).getTitle().contains("Scala")) {
            throw new DeletionException("Cannot remove Scala courses!");
         }
         portfolio.remove(id);
         return new ResponseEntity<>("[\"Removed " + id + "\"]", OK);
      } else {
         return new ResponseEntity<>(NOT_FOUND);
      }
}
```

Unhandled exception to be caught by the ExceptionResolver

CoursesController.java

156

The Iteration 2 Controller

```
    @Resource(name="portfolio")
    public void setPortfolio(Map<String,Course> portfolio) {
          this.portfolio = portfolio;
    }
    private Map<String,Course> portfolio;
}
```

```
@ControllerAdvice
@RestController
public class ExceptionResolver {
   @ResponseStatus(BAD_REQUEST)
   @ExceptionHandler(value = DeletionException.class)
   public String deleteError(Exception ex) {
      return String.format("[\"Root cause was - %s\"]",
                           ex.getMessage());
   }
}
```

Custom exception handler method

CoursesController.java

Features Added In 4.x

For our third iteration we can modernise the code

- Numerous minor additions were made from 4.0 to 4.3

Separate annotations were added for the HTTP verbs

- '`@RequestMapping(method = PUT)`' is now '`@PutMapping`'
- This enhances readability and avoids copy and paste errors

A builder API was added to the 'ResponseEntity' type

- '`new ResponseEntity<>(portfolio.get(id), OK)`' can be replaced with '`ResponseEntity.ok(portfolio.get(id))`'
- You can call the builder API directly via static imports

MVC now recognises Java 8 'Optional' objects

- These can be used for parameters that may not exist

The Iteration 3 Controller

```
@RestController("modernCoursesController")
@RequestMapping("/modern/courses")
public class CoursesController {

    @GetMapping(produces="application/xml")
    public ResponseEntity<CourseList>
                viewAllAsXml(@RequestParam("type") Optional<CourseDifficulty> type)
{

          if(portfolio.isEmpty()) {
             return ResponseEntity.notFound().build();
          } else {
             Collection<Course> courses = new HashSet<>(portfolio.values());
             type.ifPresent(t -> courses.removeIf(c ->
                                !c.getDifficulty().equals(t)));
             return ResponseEntity.ok(new CourseList(courses));
          }
    }
```

Support for Java 8 'Optional' type

GET specific annotation

Builder API

Builder API

CoursesController.java

159

The Iteration 3 Controller

```
@GetMapping(produces="application/json")
public ResponseEntity<Collection<Course>>
           viewAllAsJson(@RequestParam("type") Optional<CourseDifficulty> type) {
      if(portfolio.isEmpty()) {
         return ResponseEntity.notFound().build();
      } else {
         Collection<Course> courses = new HashSet<>(portfolio.values());
         type.ifPresent(t -> courses.removeIf(c -> !c.getDifficulty().equals(t)));
         HttpHeaders headers = new HttpHeaders();
         headers.set("NumCourses", String.valueOf(courses.size()));
         return new ResponseEntity<>(courses, headers, OK);
      }
}

@ResponseStatus(NO_CONTENT)
@PutMapping(value="/{id}", consumes="application/json")
public void addOrUpdateCourseViaJson(@RequestBody Course newCourse) {
    portfolio.put(newCourse.getId(),newCourse);
}
```

GET specific annotation

Builder API

Builder API

PUT specific annotation

CoursesController.java

160

The Iteration 3 Controller

```
@ResponseStatus(NO_CONTENT)
@PutMapping(value="/{id}", consumes="application/xml")
public void addOrUpdateCourseViaXml(@RequestBody Course newCourse) {
    portfolio.put(newCourse.getId(),newCourse);
}
@GetMapping(value="/{id}", produces="application/json")
public ResponseEntity<Course> fetchCourseDetailsAsJson(@PathVariable("id") String id) {
  if(portfolio.containsKey(id)) {
     return ResponseEntity.ok(portfolio.get(id));
  } else {
     return ResponseEntity.notFound().build();
  }
}
@GetMapping(value="/{id}", produces="application/xml")
public ResponseEntity<Course> fetchCourseDetailsAsXml(@PathVariable("id") String id) {
  if(portfolio.containsKey(id)) {
     return ResponseEntity.ok(portfolio.get(id));
  } else {
     return ResponseEntity.notFound().build();
  }
}
```

PUT specific annotation

Builder API

Builder API

GET specific annotation

Builder API

Builder API

CoursesController.java

INSTIL

161

The Iteration 3 Controller

```
@DeleteMapping(value="/{id}")
public ResponseEntity<String>
                    deleteACourse(@PathVariable("id") String id) {
      if(portfolio.containsKey(id)) {
         if(portfolio.get(id).getTitle().contains("Scala")) {
            throw new DeletionException("Cannot remove Scala courses!");
         }
         portfolio.remove(id);
         return ResponseEntity.ok("[\"Removed " + id + "\"]");
      } else {
         return ResponseEntity.notFound().build();
      }
}

@Resource(name="portfolio")
public void setPortfolio(Map<String,Course> portfolio) {
      this.portfolio = portfolio;
}
private Map<String,Course> portfolio;
}
```

DELETE specific annotation

Builder API

CoursesController.java

INSTIL

162

Testing MVC Controllers

- The MVC Test Framework

INSTIL

163

Using The MVC Test Framework

We have already seen the Spring / JUnit test runner

- Which can load in one or more bean wiring files and perform dependency injection on unit tests

Spring MVC extends this for testing controllers

- You can create a '`MockMvc`' object which can route requests to controllers via the '`DispatcherServlet`'
  - Mock objects are also provided for types in the Servlet API
- This allows you to test your controllers realistically
  - The requests you send will go through the same marshalling and validation as requests from browsers

The framework is typically combined with '`JsonPath`'

- Allowing you to validate the results of requests

INSTIL

164

The MVC Test Framework

```java
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration("/sample-config.xml")
public class TrainingCoursesServiceTest {
    private static final MediaType JSON_CONTENT_TYPE =
                MediaType.parseMediaType("application/json;charset=UTF-8");

    @Autowired
    private WebApplicationContext wac;
    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
    }
    @Test
    public void allCoursesCanBeFound() throws Exception {
        mockMvc.perform(get("/courses").accept(JSON_CONTENT_TYPE))
               .andExpect(jsonPath("$").isArray())
               .andExpect(jsonPath("$", hasSize(12)))
               .andExpect(jsonPath("$[?(@.difficulty=='BEGINNER')]", hasSize(4)))
               .andExpect(jsonPath("$[?(@.difficulty=='INTERMEDIATE')]", hasSize(4)))
               .andExpect(jsonPath("$[?(@.difficulty=='ADVANCED')]", hasSize(4)));
    }
```

TrainingCoursesServiceTest.java

165

The MVC Test Framework

```java
    @Test
    public void coursesCanBeFoundByID() throws Exception {
        mockMvc.perform(get("/courses/AB12").accept(JSON_CONTENT_TYPE))
                .andExpect(status().isOk())
                .andExpect(content().contentType(JSON_CONTENT_TYPE))
                .andExpect(jsonPath("$.id").value("AB12"))
                .andExpect(jsonPath("$.title").value("Intro to Scala"))
                .andExpect(jsonPath("$.difficulty").value("BEGINNER"))
                .andExpect(jsonPath("$.duration").value(4));
    }

    @Test
    @DirtiesContext
    public void coursesCanBeRemoved() throws Exception {
        mockMvc.perform(delete("/courses/AB12"))
                .andExpect(status().isOk());
        mockMvc.perform(get("/courses").accept(JSON_CONTENT_TYPE))
                .andExpect(jsonPath("$").isArray())
                .andExpect(jsonPath("$", hasSize(11)))
                .andExpect(jsonPath("$[?(@.title=='Intro to Scala')]", hasSize(0)));
    }
```

TrainingCoursesServiceTest.java

166

The MVC Test Framework

```
    @Test
    @DirtiesContext
    public void coursesCanBeAdded() throws Exception {
        String content = "{\"id\":\"YZ98\",\"title\":\"Extra Hard Haskell\",
                          \"difficulty\":\"ADVANCED\",\"duration\":5}";

        mockMvc.perform(put("/courses/YZ98").contentType(JSON_CONTENT_TYPE)
                                            .content(content))
                .andExpect(status().isOk());
        mockMvc.perform(get("/courses").accept(JSON_CONTENT_TYPE))
                .andExpect(jsonPath("$").isArray())
                .andExpect(jsonPath("$", hasSize(13)))
                .andExpect(jsonPath("$[?(@.title=='Extra Hard Haskell')]",
                                    hasSize(1)));
        mockMvc.perform(get("/courses/YZ98").accept(JSON_CONTENT_TYPE))
                .andExpect(status().isOk())
                .andExpect(content().contentType(JSON_CONTENT_TYPE))
                .andExpect(jsonPath("$.id").value("YZ98"))
                .andExpect(jsonPath("$.title").value("Extra Hard Haskell"))
                .andExpect(jsonPath("$.difficulty").value("ADVANCED"))
                .andExpect(jsonPath("$.duration").value(5));
    }
}
```

TrainingCoursesServiceTest.java

INSTIL

167

Introduction to Reactive

- What's wrong with lists?

INSTIL

168

```
List<Item>

Flux<Item>
Flow<Item>
Observable<Item>
```


Introducing Event Streams

Reactive Programming builds on Functional Programming

- To process streams of events instead of in-memory collections
- Conceptually we define a ‘pipeline’ for how items are processed

Streams of events are everywhere:

- User interface events like clicks, key presses and submits
- Network events when fetching resources via AJAX / REST
- Alerts from data generators (e.g. bit-coin mining) and timers

Disambiguating Streams

‘java.io.(InputStream|OutputStream|Reader|Writer)’
- You wish to do I/O (for some bizarre reason)

‘java.util.stream.Stream|IntStream|LongStream|DoubleStream’
- You wish to take advantage of multi-core hardware (intended purpose)
- You wish to do functional programming on the JVM (adopted purpose)

‘reactor.core.(Flux|Mono)’
- You wish to do reactive coding with event streams and flow control

Introducing Event Streams

Reactive Streams are defined as observable sequences
- Components act as publishers and/or subscribers

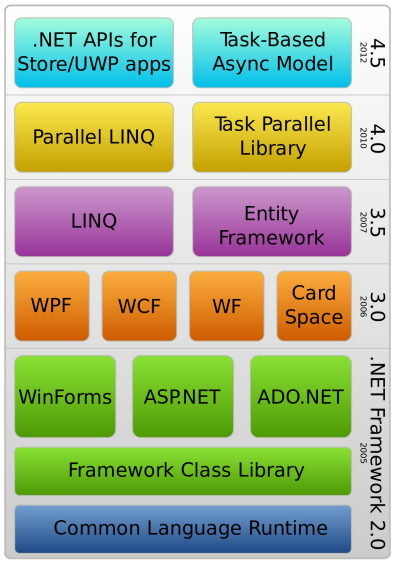
Observables share traits with both Iterators and Promises
- Iterators pull values synchronously (via ‘next’)
- Promises are pushed a single value (via ‘then’)
- Observables are pushed many items (via ‘subscribe’)

	Single Invocation	Multiple Invocations
Pulling Value(s) (Synchronous)	Option (aka Maybe) Try in Scala etc…	Iterator
Pushing Value(s) (Usually Asynchronous)	Promise (aka Future & IOU)	**Observable**

Introducing Event Streams

This is all Microsoft's fault!

- LINQ added to .NET in 2007
- Parallel LINQ emerged soon after
- LINQ Rx invented by Eric Meijer
- Async streams added to C# 8



Source: *https://commons.wikimedia.org/w/index.php?curid=2953328*

173

Introducing Event Streams

Advantages of event streams are:

- Producers and consumers can be running in different threads
- Multiple event sources can be combined in a variety of ways
- The anti-pattern of 'Callback Hell' can be avoided

The main drawback is the 'viral nature' of event streams

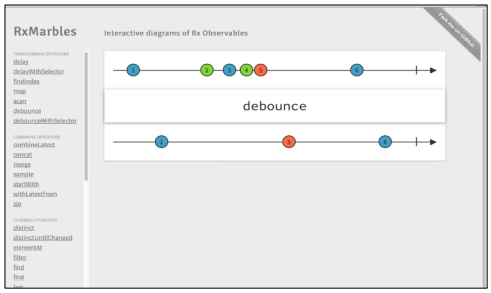
- You can't contain them to one part of your application
- Component A can't receive a *Flux* and return a *List*
  - Unless it blocks till all the items have been obtained
- It's therefore tempting to make Event Streams the default
  - But this adds complexity where streams are not required

174

Introducing Marble Diagrams

Marble diagrams are used to describe observable sequences

- Horizontal arrows denote streams over time
- Marbles denote data flowing through the stream
- Applying an operator produces a new stream

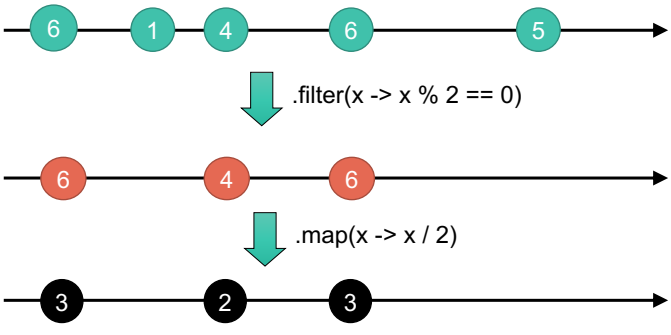


https://rxmarbles.com/

INSTIL

175

Introducing Marble Diagrams



INSTIL

176

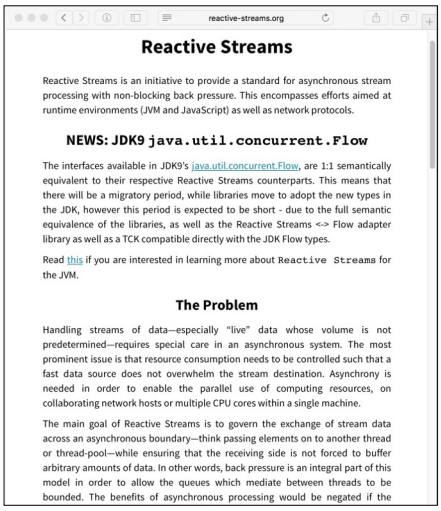
The Reactive Streams Specification

Reactive libraries are categorised

- Into a maturity model based around 4 generations

The Reactive Streams Spec defines principles and interfaces

- Shared by all implementations in order to enable interoperability



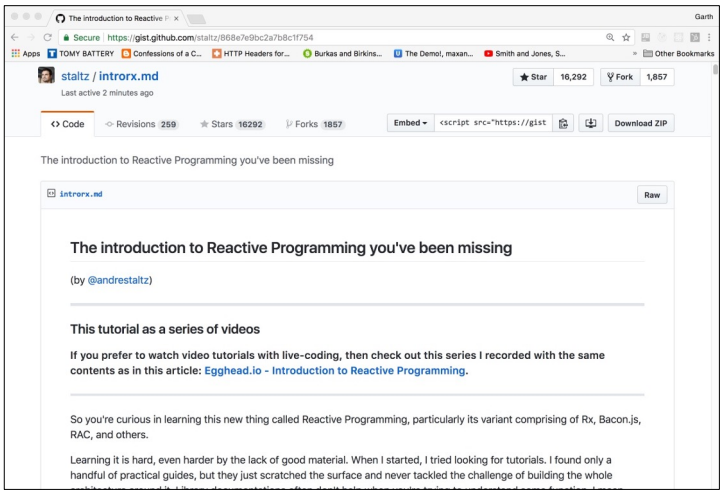
177

The Reactive Streams Specification

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}

public interface Subscription {
    void request(long n);
    void cancel();
}

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}
```


178

A Great Reactive Streams Tutorial



https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Reactive Design vs. Reactive Streams

Reactive Design and Reactive Streams are not the same

- Reactive Design is a way of architecting entire applications
- Reactive Streams are a way of passing data between layers

You don’t have to do a Reactive Design via Reactive Streams

- Other approaches, like Actors and Coroutines, are available

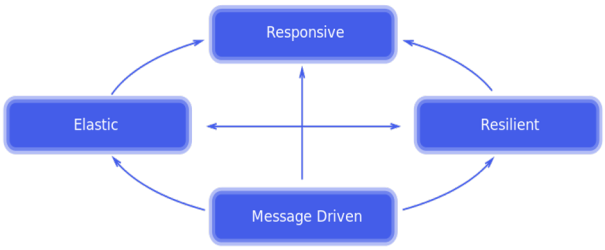
Reactive Design ≠ Reactive Streams

What is Reactive Design?

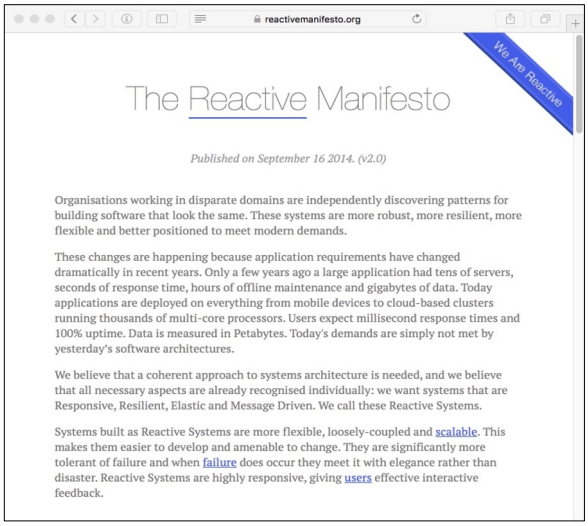
Reactive Design is summed up by the Reactive Manifesto

- Discrete components communicate via message passing
- Resilience is achieved by restarting components as needed
- Elasticity is provided by the hosting infrastructure
  - We can create new instances of components as required
  - A routing layer distributes requests to spread the load
- Coupling between components is focused on responsiveness
  - The aim is to prevent one component ever waiting on another
  - Global transactions are sacrificed in terms of eventual consistency

The Reactive Manifesto



https://reactivemanifesto.org



Introducing Project Reactor

Project Reactor is produced by Pivotal

- It is a library for supporting Event Streams
- Based on the Reactive Streams Spec

It is integrated into the String ecosystem

- It is fundamental to String from V5 onward
- The WebFlux framework builds on top of it
- As do emerging standards like R2DBC

183

Introducing Project Reactor

Reactor is based around two key types

- A '`Flux<T>`' is a asynchronous sequence of 0..N items
- A '`Mono<T>`' is an asynchronous sequence of 0..1 items
  - It supports a more limited set of operations

The '`Mono`' type is confusing at first

- Think of it as a pipe down which only one ball will roll

184

Hello Project Reactor

Let's use Reactor to build a frequency table from sample text:

1. Create a '`Flux<String>`' made up of the words in the text
2. Convert each word to lowercase via the 'map' operation
3. Convert each word to a '`Flux<Char>`' and join the streams
   - This can be achieved via the '`flatMap`' operation
   - NB we need to go from a String to an Array to a Flux
4. Use 'filter' to keep only those chars that are letters
5. Group the characters via the '`collectMultimap`' operation
   - The keys will be chars and the values a list of instances
6. Convert the values in the table to a simple count of instances
7. Print out the resulting table

185

Hello Project Reactor

```
fun main(args: Array<String>) {
    val loremIpsum = """Lorem ipsum dolor sit amet, consectetur adipiscing
                        elit, sed do eiusmod tempor incididunt ut labore et
                        ...
                        eu fugiat nulla pariatur. Excepteur sint occaecat
                        cupidatat non proident, sunt in culpa qui officia
                        deserunt mollit anim id est laborum."""

    val flux = Flux.fromIterable(loremIpsum.split(" "))
    flux.map { it.toLowerCase() }
        .flatMap { word -> Flux.fromArray(word.toCharArray().toTypedArray()) }
        .filter { theChar -> Character.isLetter(theChar) }
        .collectMultimap { it }
        .map { table -> table.mapValues { entry -> entry.value.size } }
        .subscribe(::printTable)
}
```

Program.kt

186

Hello Project Reactor

```
fun printTable(input: Map<Char, Int>) {
    println("The frequency count of letters in 'lorem ipsum' is:")
    input.forEach { entry ->
        val msgWord = "instance" + if (entry.value > 1) "s" else ""
        println("${entry.value} $msgWord of ${entry.key}")
    }
}
```

```
The frequency count of letters in 'lorem ipsum' is:
29 instances of a
3 instances of b
16 instances of c
19 instances of d
38 instances of e
3 instances of f
3 instances of g
1 instance of h
42 instances of i
...
```

INSTIL

187

Spring Boot and WebFlux

- Creating Reactive Microservices

INSTIL

188

Introducing Spring 5 and WebFlux

The Spring MVC framework was based on the Servlet API
- Which dates from when blocking networking was the only model available

Asynchronous networking is now available and popular
- On the JVM this is commonly done via the 'Netty' server

The Spring team decided to create a brand new framework
- Rather than have Spring MVC support two kinds of I/O

This new framework is known as 'WebFlux'
- Both MVC and WebFlux can exist in the same app

Introducing Spring 5 and WebFlux

WebFlux supports two coding models
- The first reuses the standard MVC annotations
- The second is based around functional concepts

Annotation based controllers look reassuringly familiar
- The only real difference is in the input and output types

Annotation Based Controllers in WebFlux

```
@RestController
@RequestMapping("/courses")
class CoursesController {

    @Resource(name = "portfolio")
    private Map<String, Course> portfolio;

    @GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public ResponseEntity<Flux<Course>> allCourses() {
        if (portfolio.isEmpty()) {
            return notFound().build();
        } else {
            Flux<Course> coursesFlux = Flux
                    .fromIterable(portfolio.values())
                    .delayElements(Duration.ofMillis(250));
            return ok(coursesFlux);
        }
    }
```

Same data source as Spring MVC

Returning values incrementally via SSE

Data returned via a Flux

Adding a delay for the demo

CoursesController.java

INSTIL

191

Annotation Based Controllers in WebFlux

```
@GetMapping(value = "/byDifficulty/{difficulty}",
            produces = MediaType.TEXT_EVENT_STREAM_VALUE)
   ResponseEntity<Flux<Course>> coursesByDifficulty(
     @PathVariable("difficulty") CourseDifficulty difficulty) {

   if (portfolio.isEmpty()) {
            return notFound().build();
        } else {
            Flux<Course> coursesFlux = Flux
                    .fromIterable(portfolio.values())
                    .filter(c -> c.getDifficulty() == difficulty)
                    .delayElements(Duration.ofMillis(250));
            return ok(coursesFlux);
        }
    }
```

CoursesController.java

INSTIL

192

Annotation Based Controllers in WebFlux

```
@PutMapping(value = "/{id}",
            consumes = "application/json",
            produces = MediaType.TEXT_PLAIN_VALUE)
   Mono<String> addOrUpdateCourse(@RequestBody Mono<Course> mono) {
       return mono.map(course -> {
           portfolio.put(course.getId(), course);
           return "Course updated";
       });
   }

   @GetMapping(value = "/{id}", produces = "application/json")
   ResponseEntity<Mono<Course>> singleCourse(
       @PathVariable("id") String id) {
       Course course = portfolio.get(id);
       if (course != null) { return ok(Mono.just(course)); }
       else { return notFound().build(); }
   }
```

CoursesController.java

193

Annotation Based Controllers in WebFlux

```
@DeleteMapping(value = "/{id}",
               produces = MediaType.TEXT_PLAIN_VALUE)
   ResponseEntity<Mono<String>> deleteById(
       @PathVariable("id") String id) {
       Course course = portfolio.get(id);
       if (course != null) {
           if (course.getTitle().contains("Scala")) {
               throw new DeletionException("Can't remove Scala!");
           }
           portfolio.remove(id);
           return ok(Mono.just("Removed " + id));
       } else {
           return notFound().build();
       }
   }
}
```

CoursesController.java

194

The WebFlux Functional API

WebFlux also offers a functional API where:

- Request handlers are represented as functions
  - The functional interface is '`HandlerFunction`'
- Routes are also represented as functions
  - The functional interface is '`RouterFunction`'
  - Routes are built via '`RouterPredicate`' objects
    - Which in turn come from '`RouterPredicates`'
    - E.g. `RequestPredicates.path("/foo")`
- All contracts are based around immutability
  - E.g. the '`ServerRequest`' and '`ServerResponse`' interfaces

The WebFlux Functional API

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) { ... }

    @Resource(name = "portfolio")
    private Map<String, Course> portfolio;

    @Bean
    public RouterFunction<ServerResponse> routes() {
        return route().path("/courses", builder -> builder
                .GET("/", accept(TEXT_EVENT_STREAM), this::allCourses)
                .GET("/{id}", accept(APPLICATION_JSON), this::singleCourse)
                .GET("/byDifficulty/{difficulty}",
                      accept(TEXT_EVENT_STREAM),
                       this::coursesByDifficulty)
                .PUT("/{id}", this::addOrUpdateCourse)
                .DELETE("/{id}", this::deleteCourse)
        ).build();
    }
```

Same data source as before

Using the DSL to create a table of routes

Application.java

The WebFlux Functional API

```
private Mono<ServerResponse> allCourses(ServerRequest request) {
    if (portfolio.isEmpty()) {
        return notFound().build();
    } else {
        Flux<Course> coursesFlux = Flux
                .fromIterable(portfolio.values())
                .delayElements(Duration.ofMillis(250));
        return ok().contentType(TEXT_EVENT_STREAM)
                .body(fromPublisher(coursesFlux, Course.class));
    }
}
```

Application.java

197

The WebFlux Functional API

```
private Mono<ServerResponse> coursesByDifficulty(
        ServerRequest request) {
    String difficulty = request.pathVariable("difficulty");
    if (portfolio.isEmpty()) {
        return notFound().build();
    } else {
        Flux<Course> coursesFlux = Flux
                .fromIterable(portfolio.values())
                .filter(c ->
                        c.getDifficulty()
                                .toString()
                                .equalsIgnoreCase(difficulty))
                .delayElements(Duration.ofMillis(250));
        return ok().contentType(TEXT_EVENT_STREAM)
                .body(fromPublisher(coursesFlux, Course.class));
    }
}
```

Application.java

198

The WebFlux Functional API

```
private Mono<ServerResponse> addOrUpdateCourse(
        ServerRequest request) {
    return request
            .bodyToMono(Course.class)
            .flatMap(course -> {
                portfolio.put(course.getId(), course);
                return ok().body(fromObject("Course Updated"));
            });
}

private Mono<ServerResponse> singleCourse(ServerRequest request) {
    String id = request.pathVariable("id");
    Course course = portfolio.get(id);
    if (course != null) {
        return ok().body(fromObject(course));
    } else {
        return notFound().build();
    }
}
```

Application.java

199

The WebFlux Functional API

```
private String removeCourse(Course course) {
        portfolio.remove(course.getId());
        return String.format("Removed %s", course.getId());
    }

    private Mono<ServerResponse> deleteCourse(ServerRequest request) {
        String id = request.pathVariable("id");
        Course course = portfolio.get(id);
        if (course != null) {
            return ok().body(fromObject(removeCourse(course)));
        } else {
            return notFound().build();
        }
    }
}
```

Application.java

200

Creating WebSockets in Spring MVC / WebFlux

Spring MVC and WebFlux both support WebSockets

- Although the API is not very deeply integrated

The event generator implements '`WebSocketHandler`'

- This defines a 'handle' method that takes a '`WebSocketSession`'
- You create a publisher object using the standard Reactor methods and then pass it to the session via the 'send' method

This generator is then registered with WebFlux by:

- Mapping an instance to a URL via a '`HandlerMapping`' object
  - The '`SimpleUrlHanderMapping`' is most commonly used
- Making the '`HandlerMapping`' available to the framework
  - Typically by returning it from a bean provider method
- Also registering a bean of type '`WebSocketHandlerAdapter`'

WebSockets in WebFlux

```
public class PairedQueueAndFlux<T> {
    private BlockingQueue<T> incomingItems;
    private Flux<T> broadcaster;
    private ExecutorService executor;

    public PairedQueueAndFlux() {
        incomingItems = new LinkedBlockingQueue<>();
        executor = Executors.newSingleThreadExecutor();
        broadcaster = Flux.create(this::bindQueueToFlux).share();
    }

    public void enqueue(T item) {
        incomingItems.add(item);
    }
```

PairedQueueAndFlux.java

WebSockets in WebFlux

```
    public Flux<T> getBroadcaster() {
        return broadcaster;
    }

    private void bindQueueToFlux(FluxSink<T> sink) {
        var submit = executor.submit(() -> {
            while (true) {
                try {
                    sink.next(incomingItems.take());
                } catch (InterruptedException ex) {
                    throw new RuntimeException(ex);
                }
            }
        });
        sink.onCancel(() -> submit.cancel(true));
    }
}
```

PairedQueueAndFlux.java

WebSockets in WebFlux

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) { ... }

    @Resource(name = "portfolio")
    private Map<String, Course> portfolio;

    @Resource(name="courses")
    private PairedQueueAndFlux<Course> coursesFlux;

    @Resource(name="courseUpdates")
    private PairedQueueAndFlux<CourseUpdateMessage> updatesFlux;

    @Bean
    WebSocketHandlerAdapter buildAdapter() {
        return new WebSocketHandlerAdapter();
    }
```

Application.java

WebSockets in WebFlux

```
@Bean
public HandlerMapping webSocketMappings() {
    var handler = new SimpleUrlHandlerMapping();
    handler.setUrlMap(Map.of(
            "/courses", allCoursesHandler(),
            "/courses/byId/{id}", singleCourseHandler(),
            "/courses/byDifficulty", coursesByDifficultyHandler(),
            "/courses/updates/all", courseUpdatesHandler()
    ));
    //This property must be set
    handler.setOrder(1);
    return handler;
}
```

Application.java

INSTIL

205

WebSockets in WebFlux

```
@Bean
public RouterFunction<ServerResponse> standardMappings() {
    return route().path("/courses", builder -> builder
            .PUT("/{id}", this::addOrUpdateCourse)
            .DELETE("/{id}", this::deleteCourse)
    ).build();
}

private WebSocketHandler courseUpdatesHandler() {
    return session -> {
        Flux<WebSocketMessage> data = updatesFlux.getBroadcaster()
                .map(update -> convertToMessage(session, update));
        return session.send(data);
    };
}
```

Application.java

INSTIL

206

WebSockets in WebFlux

```
private WebSocketHandler singleCourseHandler() {
    return session -> {
        String id = extractCourseId(session);
        Mono<WebSocketMessage> data = Mono.just(portfolio.get(id))
                .map(course -> convertToMessage(session, course));
        return session.send(data);
    };
}

private WebSocketHandler allCoursesHandler() {
    return session -> {
        Flux<Course> existing = Flux.fromIterable(portfolio.values());
        Flux<WebSocketMessage> data = existing
                .concatWith(coursesFlux.getBroadcaster())
                .delayElements(Duration.ofMillis(250))
                .map(course -> convertToMessage(session, course));
        return session.send(data);
    };
}
```

Application.java

207

WebSockets in WebFlux

```
private WebSocketHandler coursesByDifficultyHandler() {
    return session -> session.receive()
            .map(WebSocketMessage::getPayloadAsText)
            .map(CourseDifficulty::valueOf)
            .flatMap(difficulty -> {
                Stream<Course> courses =
                   filterCoursesByDifficulty(difficulty);
                Flux<WebSocketMessage> data = Flux.fromStream(courses)
                    .delayElements(Duration.ofMillis(250))
                    .map(course -> convertToMessage(session, course));
                return session.send(data);
            }).then();
}
```

Application.java

208

WebSockets in WebFlux

```
private Stream<Course> filterCoursesByDifficulty(
        CourseDifficulty difficulty) {
    return portfolio.values()
            .stream()
            .filter(c -> c.getDifficulty() == difficulty);
}

private String extractCourseId(WebSocketSession session) {
    var nettySession = (ReactorNettyWebSocketSession) session;
    var template = new UriTemplate("/courses/byId/{id}");
    var parameters = template.match(nettySession.getHandshakeInfo()
                                                .getUri()
                                                .getPath());
    return parameters.get("id");
}
```

Application.java

INSTIL

209

WebSockets in WebFlux

```
private Mono<ServerResponse> addOrUpdateCourse(
        ServerRequest request) {
    return request
            .bodyToMono(Course.class)
            .flatMap(course -> {
                String id = course.getId();
                if(!portfolio.containsKey(id)) {
                    coursesFlux.enqueue(course);
                    updatesFlux.enqueue(
                        new CourseUpdateMessage(INSERTED, id));
                } else {
                    updatesFlux.enqueue(
                        new CourseUpdateMessage(UPDATED, id));
                }
                portfolio.put(course.getId(), course);
                return ok().body(fromValue("Course Added / Updated"));
            });
}
```

Application.java

INSTIL

210

WebSockets in WebFlux

```
    private String removeCourse(Course course) {
        String id = course.getId();

        portfolio.remove(id);
        updatesFlux.enqueue(new CourseUpdateMessage(DELETED, id));
        return String.format("Removed %s", id);
    }

    private Mono<ServerResponse> deleteCourse(ServerRequest request) {
        String id = request.pathVariable("id");
        Course course = portfolio.get(id);
        if (course != null) {
            return ok().body(fromValue(removeCourse(course)));
        } else {
            return notFound().build();
        }
    }
```

Application.java

211

WebSockets in WebFlux

```
    private <T> WebSocketMessage convertToMessage(
                WebSocketSession session,
                T input) {
        var data = convertToJson(input);
        return session.textMessage(data);
    }

    private <T> String convertToJson(T input) {
        var mapper = new ObjectMapper();
        var result = "";
        try {
            result = mapper.writeValueAsString(input);
        } catch (JsonProcessingException e) {
            throw new IllegalStateException("Unable to convert" + e);
        }
        return result;
    }
}
```

Application.java

212

Spring Data

- Automating your DB access

INSTIL

213

The Object Relational Mismatch

Software and DB development are traditionally separated

- Databases tend to be used by many diverse applications
- Hence they last longer than the systems that use them

Database administrators have their own agenda

- They are guardians of the integrity of business data

Consequently two different skill sets emerged

- Software developers think of data in OO terms
- Database developers think of data in relational terms

It is unusual to find people truly skilled in both areas

- Each group underestimates the complexity of the others job

INSTIL

214

The Object Relational Mismatch

	Developers	Database Admins
Data Format	Trees of Objects (Hierarchical)	Tables with Keys (Relations)
Links	References (Unidirectional)	Primary / Foreign Keys (Bidirectional)
Primary Focus	Behavior (Methods)	Data (Tuples)
Product Lifespan	2-20 years	10-40 years
Technology Set	Rapidly Evolving (OO, VM's, AOP, FP etc…)	Slowly Evolving (SQL and Stored Procs)
Development Model	Increasingly Iterative / Agile	Entrenched Waterfall

The Object Relational Mismatch

The lack of common ground between relational and OO modelling adversely affects most software projects

- Many projects have well defined database schemas and OO designs but no clear idea of how to connect them
- This leads to a layer of 'bridge code' in the architecture which is inelegant and inefficient (aka the data access layer)

Most database access API's simply wrap up SQL

- Objects represent connections, queries, results etc…
- But developers must load and save the data into their objects

This problem is called the Object Relational Mismatch

- Possibly the most intractable problem in software design

The Object Relational Mismatch

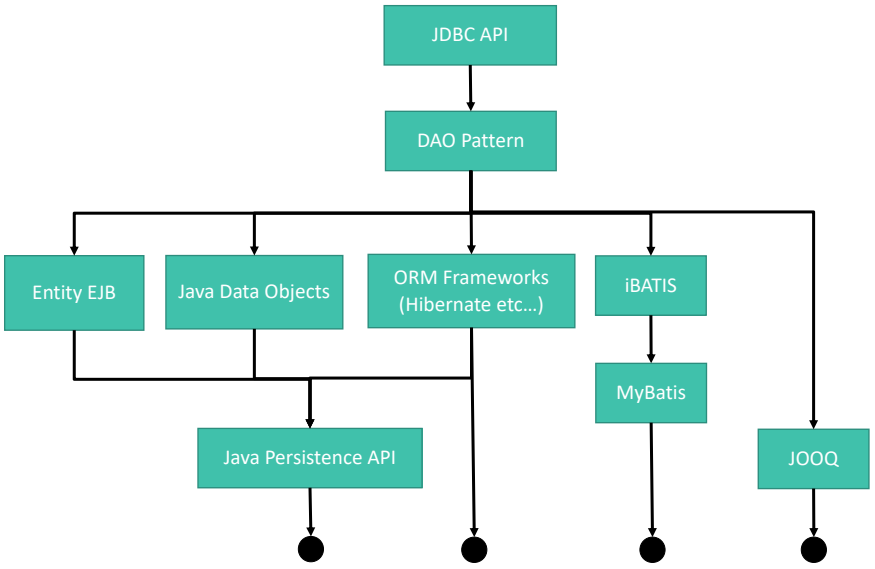
Solving the Object Relational Mismatch

A variety of different solutions have been proposed
- All of them imperfectly bridge the OO / DB divide

JDBC simply wraps DB abstractions in OO types
- You must map the data from result sets to your own types

MyBatis requires you write your own SQL statements
- And embed them via config files or annotations
- But it takes away a lot of the remaining pain

JOOQ takes a unique approach
- It uses a code generator to create a DSL from your DB schema
- You end up writing SQL like code which is fully type checked



Solving the Object Relational Mismatch

Object Relational Mappers are the most common solution
- Hibernate popularised the concept in the Java community

All ORM's work the same way:
- You map types to tables and fields to columns
- You are given an API for loading and saving objects
- The framework generates the SQL to persist data
  - Based on the mappings you provided

Originally each ORM has its own mapping syntax and API
- E.g. in Hibernate you used XML files and the 'Session' type

The Java Persistence API standardised all Java ORMs
- It provides a standard API based on the 'EntityManager' type
- It defines annotations and formats or configuration files

Solving the Object Relational Mismatch

ORM frameworks aren't a panacea

- They are built on the assumption that it is worthwhile to take your data out of the DB and covert it from to an object based format

This assumption is not always correct

- E.g. when performing anything that counts as 'batch processing'

If you apply an ORM to a 'data first' problem then:

- You incur a performance hit without any benefit
- You complicate your design for no good reason

No single approach should be adopted exclusively

Characteristics of the JPA

An implementation of the JPA must:

- Maintain a cache of references to persistent objects
- Detect when an objects state changes (becomes 'dirty')
- Generate SQL as required to save these changes

These features must be applied to trees of objects

- Each persistent object many have many associations

There are several approaches to dirty checking

- The impl could save a snapshot of each objects state
  - As it was after the last transaction committed
- The implementation could inject bytecode into objects
  - To set flags when their state changes

The Lifecycle of a Persistent Object

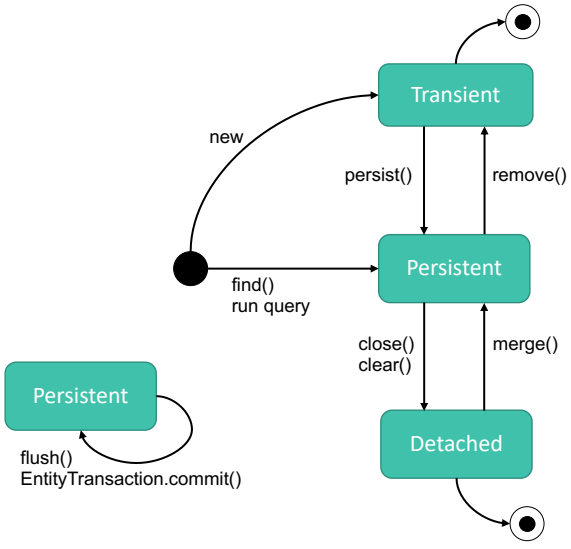
A persistent object can be in one of three states
- Known as transient, persistent and detached

A transient object is not associated with a DB record
- If the object is deleted then the information is lost

A persistent object is in the entity manager cache
- It can remain there for any number of synchronizations

A detached object has left the entity manager cache
- It still represents an existing record in the database
- But nothing is keeping the two in sync

223



224

The Lifecycle of a Persistent Object

An object becomes transient when:

- It is created via 'new' to hold a new record
- The record it represents is deleted

Transient objects do not appear in the EM cache

- In the former case because they have yet to be added and in the latter because they are automatically evicted

An object becomes persistent when:

- A transient object is passed to the EM via 'persist'
- A detached object is passed to the EM via 'merge'
- The EM creates the object itself
  - Note that this may be due to an association

225

The Lifecycle of a Persistent Object

Persistent objects are synchronized periodically

- When and why this occurs can be very complex
- The implementation must detect when an object is dirty

Writes do not occur only when transactions commit

- E.g. if pending changes would affect the results returned by an impending query then those changes must be written early

An object becomes detached when:

- The EM in which it was referenced is closed or emptied
- It is evicted from the EM cache individually

Detached objects can become persistent again

- This is a critical aspect of JEE architecture…

226

Annotating Persistent Classes

JPA developers can place mapping info in two places
- In mapping files referenced from 'persistence.xml'
- In annotations decorating the persistent classes

The annotation approach is the most convenient
- But the XML method remains the most flexible

Mappings in the XML can override those in annotations
- If the 'xml-mapping-metadata-complete' element is used then all mapping information provided via annotations is ignored

Note that the JPA assumes sensible defaults
- E.g. unless otherwise specified the class and table names are assumed to be identical, as are column / field names
- Relationships between classes need to be explicitly mapped

Annotations Within JPA

Annotation	Description
@Transient	Specifies that a property or field is not persistent
@Table @SecondaryTable @SecondaryTables	Specifies the primary and (optionally) secondary tables into which an entity will be saved. By default the table is assumed to have the same name as the class.
@JoinColumn @JoinColumns	Specifies a foreign key relationship between primary and secondary tables for use with associations between beans
@Id	Specifies the identifier of the bean which maps to a primary key
@IdClass	Specifies a class holding multiple fields which, when taken together, map to the composite primary key used by an entity bean
@UniqueConstraint	Specifies a constraint of a primary or secondary table
@Column	Specifies the column into which a persistent property or field is stored. By default the names are assumed to be identical

Annotations Within JPA

Annotation	Description
@Basic	Specifies that a mapping involves a standard Java type
@Lob	Specifies that a field or property should be mapped into a CLOB or BLOB
@Serialized	Specifies that a field or property should be mapped using serialization
@Embeddable	Specifies that an object can be saved as part of a larger entity
@Embedded	Shows that an entity bean embeds an embeddable object
@EmbeddedId	Specifies that an embedded class is being used as a composite primary key
@OneToOne @OneToMany @ManyToOne @ManyToMany	Specify the different types of association that can exist between beans
@AssociationTable	Specifies the table required for a many to many association

229

Annotations Within JPA

Annotation	Description
@Inheritance	Defines the mapping strategy used when one bean inherits from another
@InheritanceJoinColumn @InheritanceJoinColumns	Specifies the column(s) that should be used to join the table of a derived bean with the table of its superclass
@AttributeOverride	Overrides mappings of properties or fields (e.g. during inheritance)
@DiscriminatorColumn	May be required by the container in order to work out which objects data is being stored in a particular row (e.g. when multiple beans are being stored in the same table)
@GeneratedIdTable	Defines a table used to store generated id values for beans
@SequenceGenerator @TableGenerator	Define generators that can be used to create bean id's / primary keys (the generator can then be used in @Id)
@Version	Used for controlling concurrency

230

Annotations Within JPA

Annotation	Description
@PrePersist	Marks a method as a callback which will be triggered at a particular point of the objects lifecycle. They may be defined inside the Entity Bean or in a separate listener class
@PostPersist	
@PreRemove	
@PostRemove	
@PreUpdate	
@PostUpdate	
@PostLoad	

Annotation	Description
@FlushMode	Specifies if an objects data should be flushed as part of a method or query
@NamedQuery	Specifies that a method is associated with a query

Understanding JPA-QL / JQL

Both the JPA and the (now long deprecated) Entity EJB type required a mini-language for running queries

- This allows a developer to find multiple records without needing to know SQL, the mappings or the DB schema
- This began as EJB-QL evolved into JPA-QL and is not JQL

A JQL query mixes relational and OO concepts

- The syntax will be familiar to anyone who has seen SQL
- However navigation is performed using OO associations

Queries can be placed in XML or annotations

- The 'EntityManager' interface lets you create and run Query objects, with support for parameters and paging

Introducing Spring Data

The JPA may well be suitable for your persistence needs
- But should all your developers learn the JPA in depth?

There is a lot of benefit to wrapping up your use of the JPA
- In types known as 'Data Access Objects' or 'Repositories'
- This abstracts the details of the ORM from most developers
- Repositories can be injected into controllers and services

The good news is that this process can be automated
- We can define a generic interface for all repositories
- With type parameters for the entity and primary key

This is the service provided by Spring Data
- It creates DAOs using JPA, MyBatis etc. as the underlying API

233

A Spring Data Interface for DAO's

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S var1);
    <S extends T> Iterable<S> saveAll(Iterable<S> var1);

    Optional<T> findById(ID var1);
    boolean existsById(ID var1);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> var1);

    long count();

    void deleteById(ID var1);
    void delete(T var1);
    void deleteAll(Iterable<? extends T> var1);
    void deleteAll();
}
```

CrudRepository.java

234

Creating Custom Finder Methods

A DAO / Repository interface cannot be fully generic

- Because each type will require its own finder methods
- E.g. a 'Customer' type might need 'findByCreditRating'

Spring Data manages this via 'convention over configuration'

- You add custom finder methods to the Repository interface and they are implemented for you based on naming conventions
- E.g. 'findByFoo' would find all entities based on their 'foo' property and 'findByFooContaining' would do a partial match

The 'query' annotation is for other cases

- You use it to state a query using the language of your ORM
- Such as JQL in the case of JPA and Cypher for Neo4J

Using Spring Data to Create Repositories

```
public interface DeliveryRepository
       extends CrudRepository<Delivery, Integer> {
}

public interface CourseRepository
       extends CrudRepository<Course, String> {
    List<Course> findByType(String type);
    List<Course> findByTitle(String title);
    List<Course> findByTitleContaining(String text);
    List<Course> findByTypeAndTitleContaining(String type,
                                              String titleText);
    @Query("SELECT c.number FROM Course c")
    List<String> findByCustomQueryOne();
}
```

CourseRepository.java

Using Repositories Created By Spring Data

```
@SpringBootApplication
public class CourseBookingApplication {

    public static void main(String[] args) {
        SpringApplication.run(CourseBookingApplication.class, args);
    }

    @Bean
    public CommandLineRunner runDemo(CourseRepository courses,
                                     DeliveryRepository deliveries) {
        return new CommandLineRunner() {
            public void run(String... args) throws Exception {
                showCoreFeatures(courses, deliveries);
                showCustomFinderMethods(courses);
            }
        };
    }
```

CourseBookingApplication.java

INSTIL

237

Using Repositories Created By Spring Data

```
private void showCustomFinderMethods(CourseRepository courses) {
    findCoursesByType(courses);
    findCoursesByTitle(courses);
    findCoursesByTitleContaining(courses);
    findCoursesByTypeAndTitle(courses);
    findCourseNumbers(courses);
}

private void showCoreFeatures(CourseRepository courses,
                              DeliveryRepository deliveries) {
    printSingleCourse(courses);
    printAllCourses(courses);
    addNewCourse(courses);
    printAllCourses(courses);
    removeCourse(courses);
    printAllCourses(courses);
    printDeliveryDetails(deliveries);
}
```

CourseBookingApplication.java

INSTIL

238

Using Repositories Created By Spring Data

```
private void printSingleCourse(CourseRepository repository) {
    Optional<Course> opt = repository.findById("AB12");
    String title = opt.map(Course::getTitle).orElse("UNKNOWN");
    System.out.printf("\tCourse 'AB12' has the title: '%s'\n", title);
}

private void printAllCourses(CourseRepository repository) {
    Iterable<Course> courseList = repository.findAll();
    System.out.println("Details of all the courses are:");
    for (Course c : courseList) {
        System.out.printf("\t[%s] %s\n", c.getNumber(), c.getTitle());
    }
}

private void addNewCourse(CourseRepository repository) {
    Course course = new Course();
    course.setTitle("Advanced Scala");
    course.setType("advanced");
    course.setNumber("YZ89");

    repository.save(course);
}
```

CourseBookingApplication.java

239

Using Repositories Created By Spring Data

```
private void removeCourse(CourseRepository repository) {
    Optional<Course> opt = repository.findById("YZ89");
    opt.ifPresent(repository::delete);
}

private void printDeliveryDetails(DeliveryRepository repository) {
    Optional<Delivery> opt = repository.findById(1004);
    opt.ifPresent(delivery -> {
        String courseTitle = delivery.getCourse().getTitle();
        String programName = delivery.getTrainingProgram().getName();

        String msg = "\tDelivery 1004 is of course '%s' in program '%s'\n";
        System.out.printf(msg, courseTitle, programName);
    });
}

private void findCoursesByType(CourseRepository repository) {
    Iterable<Course> courseList = repository.findByType("beginners");
    System.out.println("Details of all beginners courses are:");
    for (Course c : courseList) {
        System.out.printf("\t[%s] %s\n", c.getNumber(), c.getTitle());
    }
}
```

CourseBookingApplication.java

240

Using Repositories Created By Spring Data

```
    private void findCoursesByTitleContaining(CourseRepository repository) {
        Iterable<Course> courses = repository.findByTitleContaining("Intro");
        System.out.println("Details of courses containing the word 'Intro' are:");
        for (Course c : courses) {
            System.out.printf("\t[%s] %s\n", c.getNumber(), c.getTitle());
        }
    }

    private void findCoursesByTypeAndTitle(CourseRepository repository) {
        Iterable<Course> courses =
                repository.findByTypeAndTitleContaining("Beginners", "Java");
        System.out.println("Details of Beginners courses containing 'Java' are:");
        for (Course c : courses) {
            System.out.printf("\t[%s] %s\n", c.getNumber(), c.getTitle());
        }
    }

    private void findCourseNumbers(CourseRepository repository) {
        Iterable<String> results = repository.findByCustomQueryOne();
        System.out.println("The numbers of all the Courses are:");
        for (String str : results) {
            System.out.printf("\t%s\n", str);
        }
    }
}
```

CourseBookingApplication.java

INSTIL

241

Spring Security – Part 1

- Core Concepts

INSTIL

242

Core Concepts of Security

Authentication involves verifying the identity of an entity

- E.g. you login to verify who you are

Authorisation involves controlling access to resources

- E.g. Once logged in, you have access to the admin page

We'll take a look at these in the context of:

- OAuth – authorisation framework
- OpenID – authentication framework
- JWT – method of transmitting secure data

Common Security Entities

Entity	Description
User	The person accessing the system
Role	A high level position for a user. A user can have many roles.
Privilege	A low level access right for a role

Introducing OAuth

OAuth is an open standard for access delegation

- Whereby a user grants access to his information for one site or application via a different site
- E.g. Allow Spotify to post to your Facebook

The site granted access never has access to the password

- They only ever have a token which grants limited access

The user can easily control access via the granting site

- They can view all applications with access, determine what privileges they have and then restrict or remove as required

Introducing OAuth2

OAuth 2.0 is the current standard

- It was released in 2012 and is not backwards compatible

It defines authorization flows for:

- Web applications
- Desktop application
- Mobile Devices
- Living room devices

Note that Transport Layer Security (TLS) is required

- OAuth2 does no encryption or signing itself
- The communication protocol must provide this

Introducing OAuth

OAuth defines four roles:

- **Resource Owner** – the user who owns the account
- **Client** – the application requesting access
- **Resource Server** – the server that hosts the user's account
- **Authorization Server** – the server that verifies identify and issues tokens

The client needs to be known to the resource owner

- It must be registered previously with:
  - Application Name
  - Application Website
  - A Redirect URI that will process issued tokens
- Often a localhost addressed developer client will be provided

247

Introducing OAuth

The flow if events is as follows:

- The resource owner wants to use a client site or application
- The client needs access to another app, the resource server
- The authorization server verifies the client and issues a token
- The token grants the client access to some information

Authentication is the most common use case

- But OAuth provides authorisation, not authentication
- Normally, authentication is part of the flow on the resource server side, but not necessarily

OAuth can be used to provide Single Sign On

248

OpenID and OpenID Connect (OIDC)

OpenID is an open standard for decentralised authentication
- Note, authentication, while OAuth is technically authorisation

OpenID Connect 1.0 is the current version of OpenID
- It is the third generation, succeeding OpenID 2.0
- It's a layer that sits on top of OAuth2
- So it provides authentication using Authorisation Servers

The terminology in OpenID is as follows:
- An Identity Provider or OpenID Provider (OP) provides the authentication service
- It asserts the identity of an End User
- Relying Parties (RP) or OpenID Acceptors utilise the service
- As well as authentication the provider supplies user attributes

JSON Web Token (JWT)

A claim is a piece of information asserted by a subject
- Represented as a name/value pair
- E.g. Username = Bob Roberts

JWT (pronounced 'jot') is a method for representing claims between parties under the open standard RFC 7519
- It is part of the (JOSE) framework
  - JavaScript Object Signing and Encryption
- A JWT contains a set of claims

JWT Use Cases

Authentication is the obvious use case:

- The client logs in to the server which generates a token
- The client uses the token for all future communication
- The token is sufficient for authentication & authorisation

However, JWT can be used to safely exchange any data

- The token is compact and easy to transmit over HTTP
- It is self-contained, preventing extra trips to the DB

JSON Web Token (JWT)

JWT is encoded (base64) but may or may not be encrypted

- It is still secure via an included signature
- You may wish to encrypt so the claims are also private

The token is transmitted as a JSON object within either:

- Payload of a Java Web Signature (JWS)
- Plaintext within a JSON Web Encryption (JWE)

The object is then represented via either:

- JWS Compact Serialisation
- JWE Compact Serialisation

Signing can be done via:

- A hash generated via Message Authentication Code (MAC)
- A public private key RSA

Spring Security Part 2

- The Framework Itself

INSTIL

253

Introducing Spring Security

Spring Security follows the pattern of other projects:

- It is feature rich
  - With an extensive list of enterprise grade features
- It is built on Aspect Oriented Programming (AOP)
  - Annotations for easy, declarative configuration
- It is configured via Inversion of Control (IoC)
  - Strategies and extension points to hook in your code

INSTIL

254

Sample Spring Security Annotations

Name	Description
EnableOAuth2Sso	Conveniently enable OAuth2 Single Sign On. Settings automatically read from spring.oauth2 in application properties
EnableOAuth2Client	Enable OAuth2 in a client web application that uses Spring Security. Uses the Application Code Grant. Requires manual configuration of OAuth2 settings and objects

Spring Boot and Spring Security

Spring Boot provides autoconfiguration for Spring Security

By simply adding the starter dependency a default login is added to your web application or service

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

By adding beans, annotations etc we can customise security

The WebSecurityConfigurerAdapter

This class is extended to configure request processing

- Requests are processed by a chain of **filters**
- We can add our own filter objects to this chain
- Many standard filters exist to do common operations

The 'configure' method handles filter chain configuration

- The passed HttpSecurity object has a fluent syntax

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        //configure filters here
    }
}
```


257

HttpSecurity

The HttpSecurity fluent syntax has extensive features

- Permit or restrict access to paths
- Configure Cross-Site Request Forgery (CSRF)
- Manually add custom filters

The order of calls is significant, as it is building a chain

- Permitting access to all addresses, then restricting access to some will not work – the request will be permitted first

The syntax equates to the older XML structure

```
http.anyRequest().authenticated()
    .logout().// ...
    .csrf().// ...
    .addFilterBefore(...);
```


258

Sample HttpSecurity Configuration

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {
    private static final Logger logger =
            LoggerFactory.getLogger(ApplicationSecurity.class);

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        logger.info("Configuring HTTP security");

        http.antMatcher("/**")
                .authorizeRequests()
                    .antMatchers("/", "/login**", "/webjars/**")
                        .permitAll()
                    .anyRequest().authenticated()
                    .and()
                .logout()
                    .logoutSuccessUrl("/")
                    .permitAll()
                    .and()
                .csrf()
                    .csrfTokenRepository(
                        CookieCsrfTokenRepository.withHttpOnlyFalse())
                    .and()
                .addFilterBefore(combinedFilters(),
                                 BasicAuthenticationFilter.class);
    }
}
```

ApplicationSecurity.java

259

Common HttpSecurity Methods

Method	Description
authorizeRequests	Configure access based on the request e.g. authorizeRequest().antMatchers("/foo/**").hasRole("BAR")
antMatchers, regexMatchers, mvcMatchers	Specify configuration for a subset of URIs or MVC matches
anonymous	Configure behaviour for an anonymous user
addFilter, addFilterAfter, addFilterBefore, addFilterAt	Add a filter to the request filter chain
cors, csrf	Add CORS and CSRF filters
exceptionHandling	Configure exception handling
formLogin	Configure form based authentication
openidLogin	Configure OpenID based login
logout	Configures logout processing

260

AuthenticationManager

The main authentication strategy is `AuthenticationManager`

- An interface with a single 'authenticate' method

The '`authenticate`' method either:

- Returns an '`Authentication`' object if it succeeds
- Throws an '`AuthenticationException`' if it fails
- Returns '`null`' if it can't decide

```
public interface AuthenticationManager {
    Authentication authenticate(Authentication authentication)
            throws AuthenticationException;

}
```


ProviderManager

The most common implementation of `AuthenticationManager` is `ProviderManager`

It allows us to chain `AuthenticationProviders` together

- The providers can be queried against an authentication type
- This allows multiple authentication types to be supported by a single `ProviderManager`/`AuthenticationManager`
- This is using the Chain of Responsibility pattern

```
public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication)
            throws AuthenticationException;
    boolean supports(Class<?> authentication);

}
```

Configuring the AuthenticationManager

The authentication manager is setup using a builder

- The `AuthenticationManagerBuilder`

It can be auto-wired into an initialisation method

- The method can be placed into any configuration class
- Typically added to the `WebSecurityConfigurerAdapter`

```
@Configuration
public class ApplicationSecurity extends WebSecurityConfigurerAdapter {
    @Autowired
    public initialize(AuthenticationManagerBuilder builder) {
        // ...
    }
}
```


Cross Site Request Forgery (CSRF)

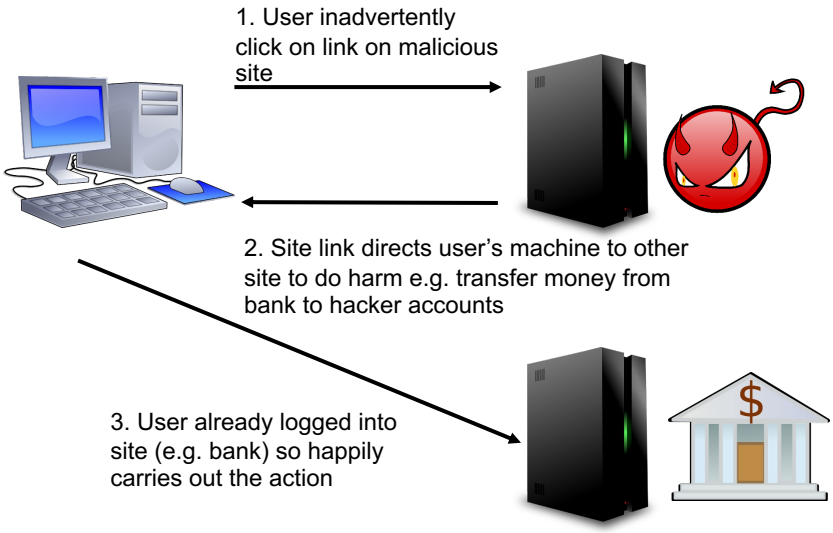
CSRF executes unwanted actions from a users machine

It exploits the fact that a user will already be authenticated

- Within the browser a user is authenticated
- A cookie stores authentication information
- All requests to a site automatically include the cookie
- Thus, any unwanted request to the site it will succeed
- CSRF simply need the user to trigger the actions

CSRF could be initiated by a link on a site or in an email

Introducing CSRF



INSTIL

265

CSRF Protection

CSRF protection involves using an additional token

- The token is stored in a cookie when logging in
- The token is added to the header in requests via JavaScript
  - A server checks for this token or else rejects the request

Browsers only allow JS to read data from their own site

- It is impossible within a browser to run JS to trigger an unwanted operation on one site, from another, malicious, site

This only needs to be supported for state change requests

- E.g. POST request to purchase an item

INSTIL

266

Spring Security and CSRF

Spring supports CSRF out of the box

- You can disable it or add your own manual filters

HttpSecurity fluent syntax supports configuration of CSRF

- E.g. '`.csrf().disable()`'

Some frameworks like Angular support this too

- Via a XSRF-TOKEN cookie and X-XSRF-TOKEN header
- The HttpOnly false allows the saved cookie to be read by JS

```
.csrf()
    .csrfTokenRepository(
        CookieCsrfTokenRepository.withHttpOnlyFalse()
    )
```