

Emergence and Overview of Microservices

Monolithic Application Architecture

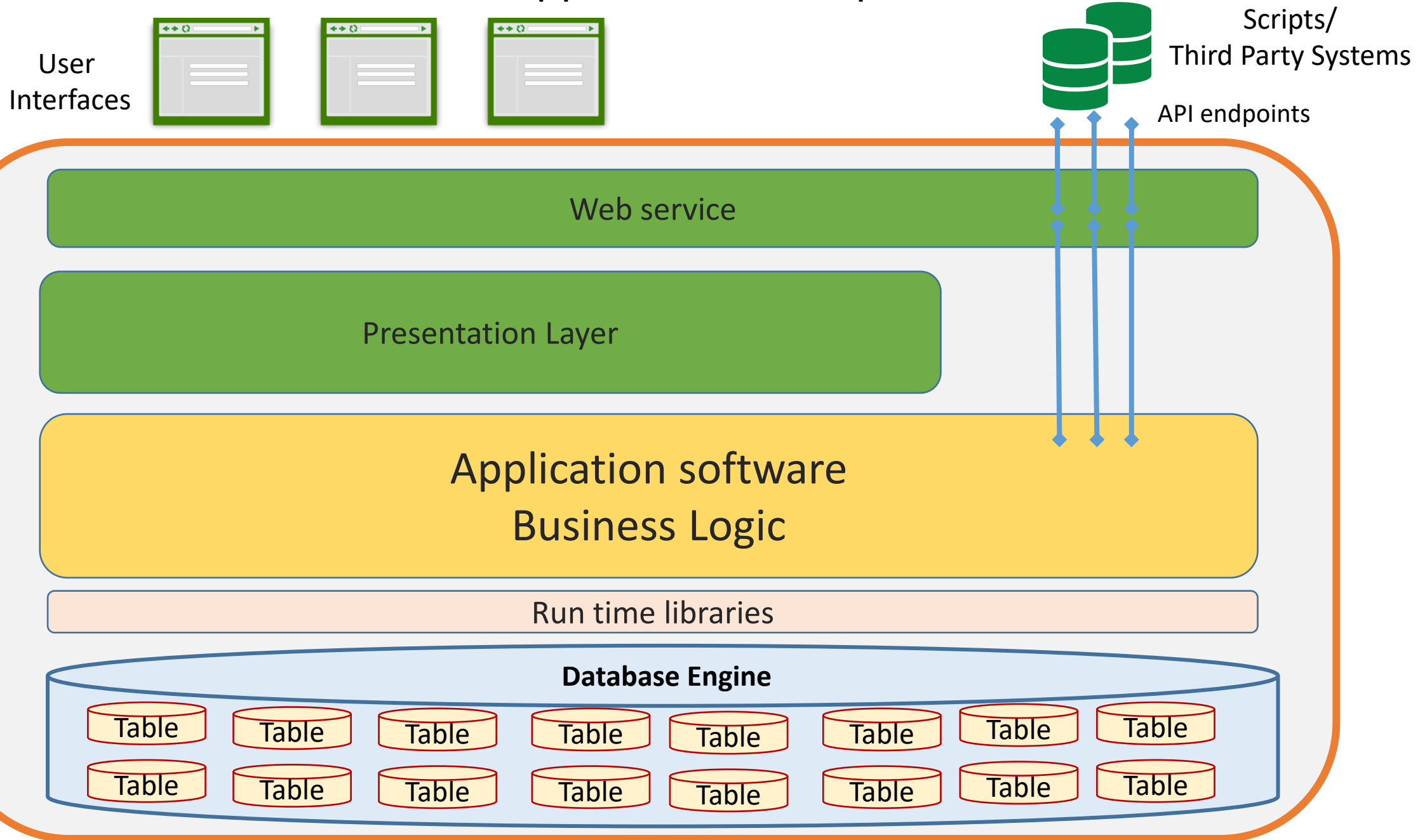
- The term "monolithic" comes from the Greek terms *monos* and *lithos*, together meaning a large stone block. The meaning in the context of IT for monolithic software architecture characterizes the uniformity, rigidity, and massiveness of the software architecture.
- A monolithic code base framework is often written using a single programming language, and all business logic is contained in a single repository.

Monolithic Application Architecture

- Typical monolithic applications consist of a single shared database, which can be accessed by different components. The various modules are used to solve each piece of business logic. But all business logic is wrapped up in a single API and is exposed to the frontend. The **user interface (UI)** of an application is used to access and preview backend data to the user. Here's a visual representation of the flow:



Monolithic Application Conceptual Model



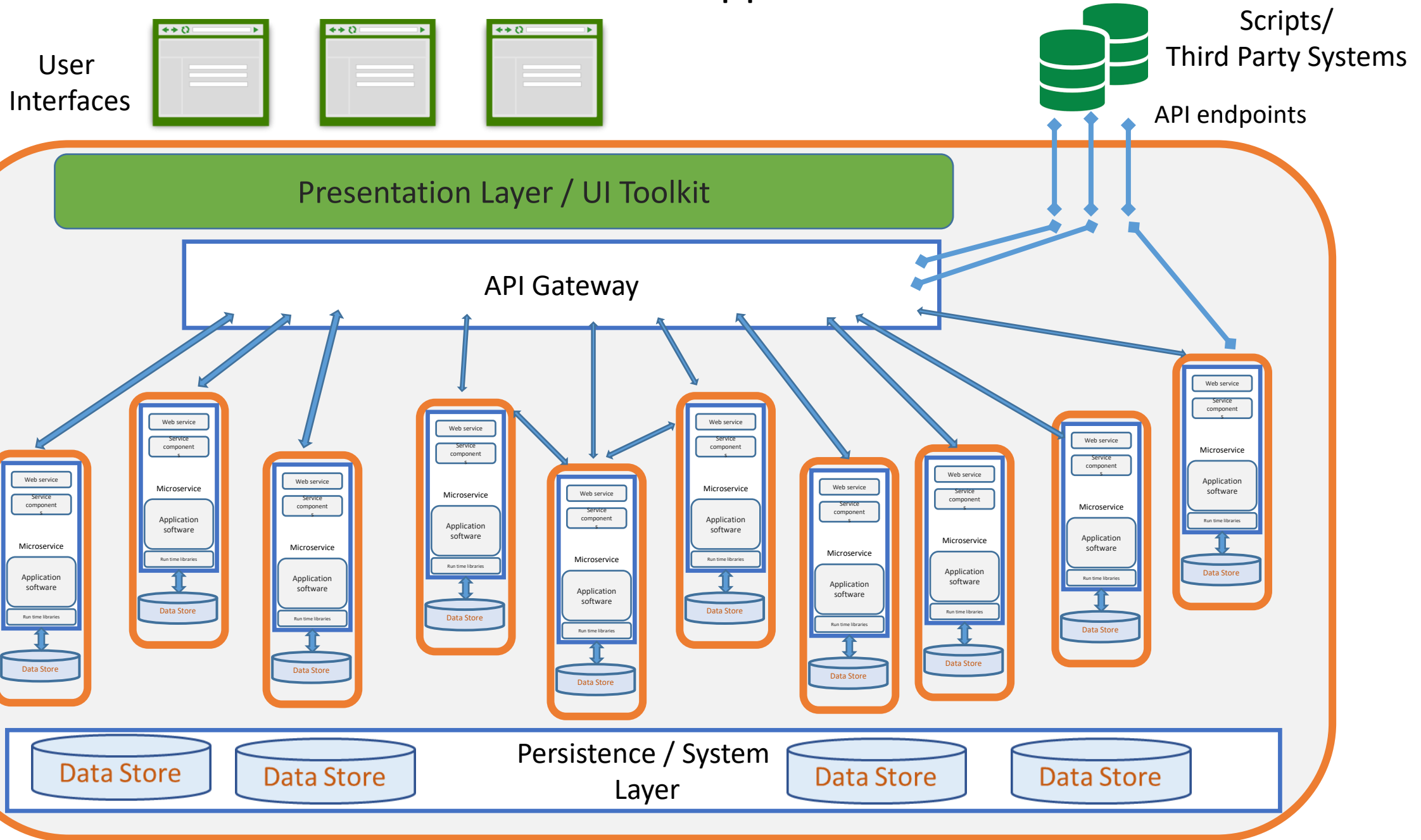
Microservices

- Microservices is an architecture style, in which large complex software applications are composed of one or more services.
- Microservice can be deployed independently of one another and are loosely coupled.
- Each of these microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability.

Microservices

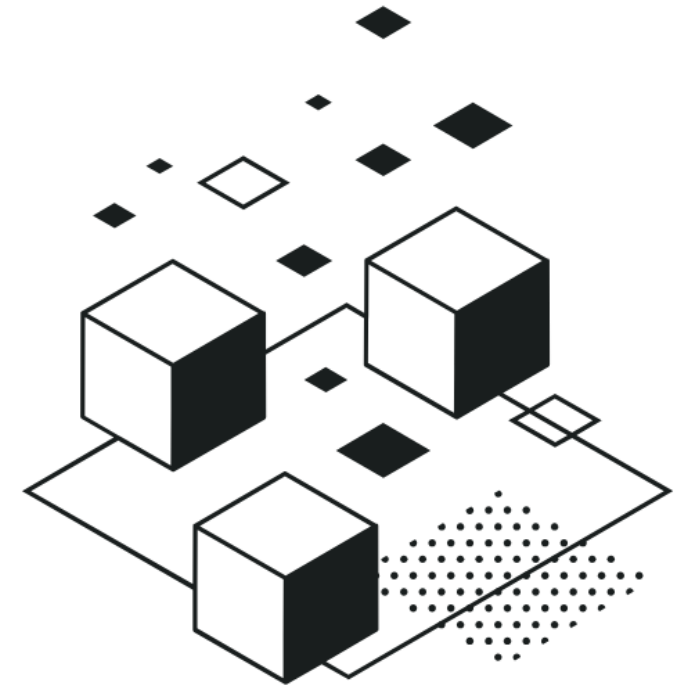
- Microservices can be developed in any programming language. They communicate with each other using language-neutral application programming interfaces (APIs) such as Representational State Transfer (REST).
- Microservices also have a bounded context. They don't need to know anything about underlying implementation or architecture of other microservices.

Microservices-based Application



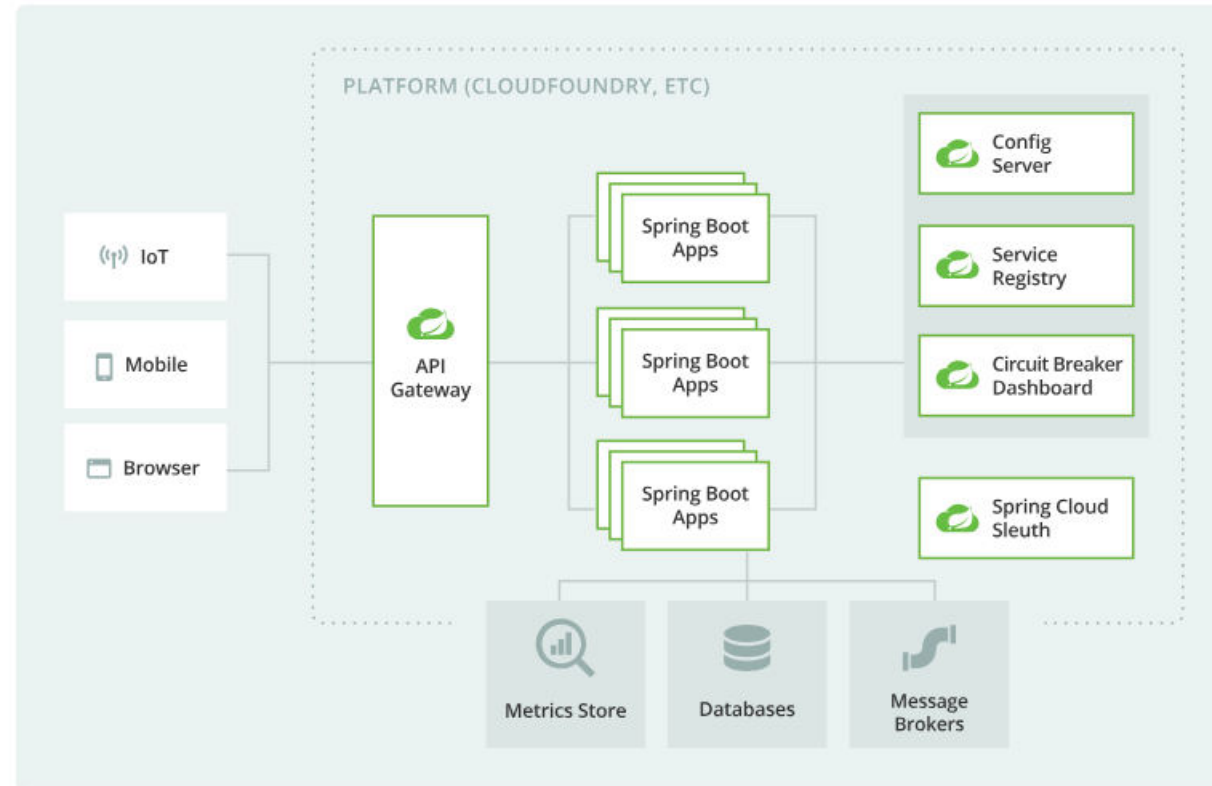
Microservices Architecture

- Microservice architectures are the ‘new normal’. Building small, self-contained, ready to run applications can bring great flexibility and added resilience to your code.
- Spring Boot’s many purpose-built features make it easy to build and run your microservices in production at scale.
- And don’t forget, no microservice architecture is complete without [Spring Cloud](#) – easing administration and boosting your fault-tolerance.



Microservices resilience with Spring Cloud

- The distributed nature of microservices brings challenges. Spring helps you mitigate these. With several ready-to-run cloud patterns, Spring Cloud can help with service discovery, load-balancing, circuit-breaking, distributed tracing, and monitoring. It can even act as an API gateway.

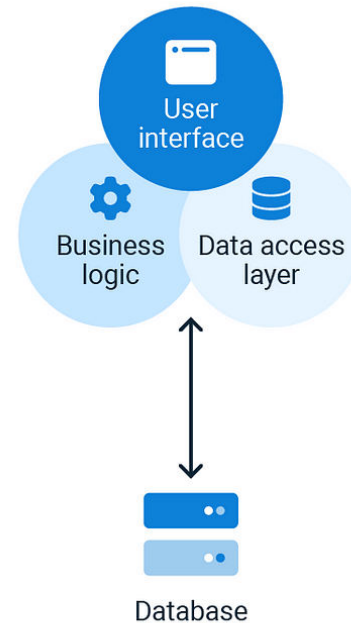


Benefits of transition from Monolithic to Microservices

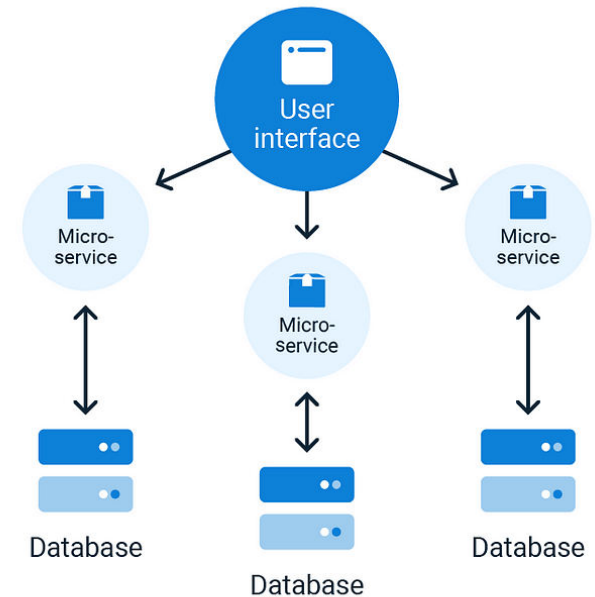
The transition from monolithic applications to microservices can bring many benefits, including:

- 1.Improved scalability
- 2.Increased flexibility
- 3.Faster delivery time
- 4.Better resource utilization
- 5.Technology diversity

Monolithic Architecture



Microservice Architecture



Monolithic vs Microservices

Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

Microservices Architectures

- **Single-Service Microservices:** Each microservice is responsible for a single, very specific task. This results in a large number of very small microservices.
- **Domain-Driven Microservices:** The services are designed based on business capabilities and priorities. This approach emphasizes understanding the business's needs and modeling the services accordingly.
- **Serverless Microservices:** This is a new type of microservice where the service is hosted on a third-party server. This approach helps businesses to scale automatically and pay only for the resources they use.

Integration patterns

- Integration patterns are a key aspect of a microservices architecture. They provide a roadmap to enable multiple microservices, possibly employing different protocols like REST or AMQP, to function in harmony.
- These patterns aim to provide an efficient way for clients to interact with individual microservices without having to handle the intricacies of various protocols.
 - API Gateway Pattern
 - Proxy Pattern
 - Gateway Routing Pattern
 - Chained Microservice Pattern
 - Branch Pattern
 - Client-Side UI Composition Pattern

Inter-Service Communication

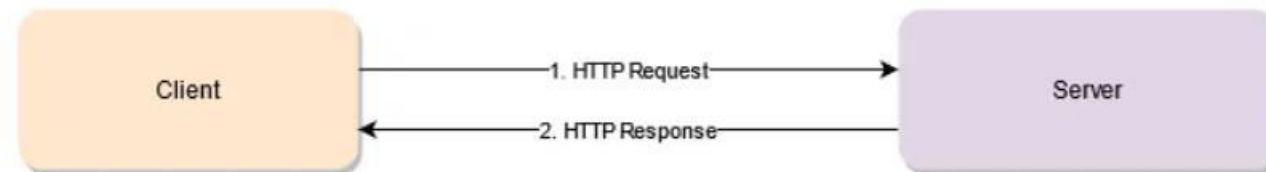
- Inter-service communication is a pivotal concept in the microservices architecture. In a microservices architecture, an application is broken down into loosely coupled, independently deployable services.
- These services need to communicate with each other to perform their tasks.
- For Example: In the Spring framework, there are several ways to handle inter-service communication, ensuring seamless operation and interaction between microservices.

Types of Inter-Service Communication

- Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.
 - Synchronous protocol
 - Asynchronous protocol

HTTP APIs

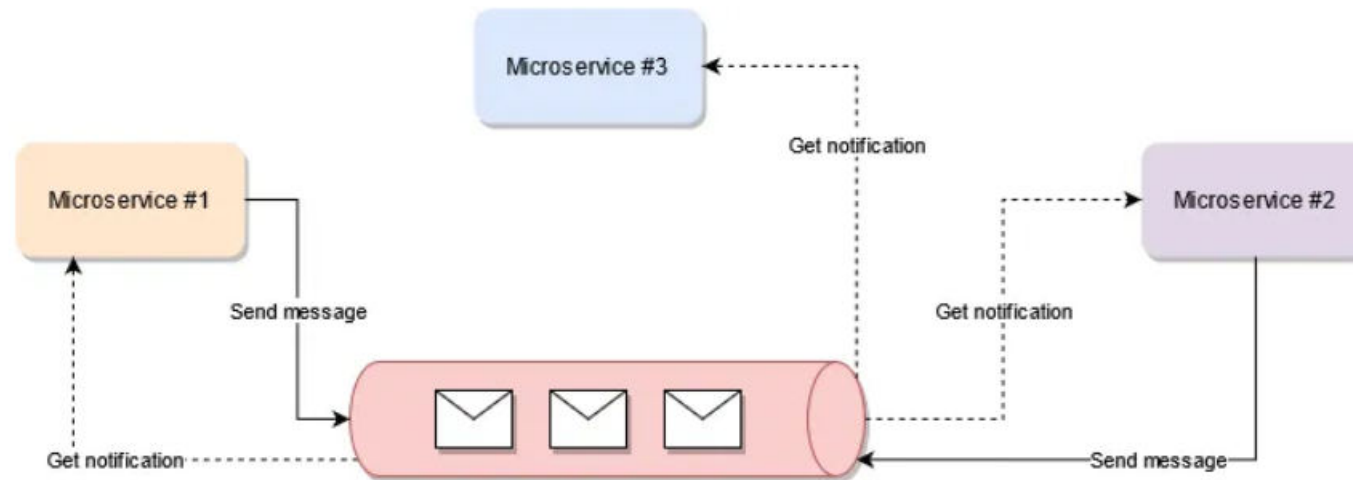
- An HTTP API essentially means having your services send information back and forth like you would through the browser or through a desktop client like Postman.
- It uses a client-server approach, which means the communication can only be started by the client. It is also a synchronous type of communication, meaning that once the communication has been initiated by the client, it won't end until the server sends back the response.



Classic Client-Server microservice communication

Asynchronous Messaging

- This pattern consists of having a message broker between the producer of the message and the receiving end.
- This is definitely one of my favorite ways of communicating multiple services with each other, especially when there is a real need to horizontally scale the processing power of the platform.

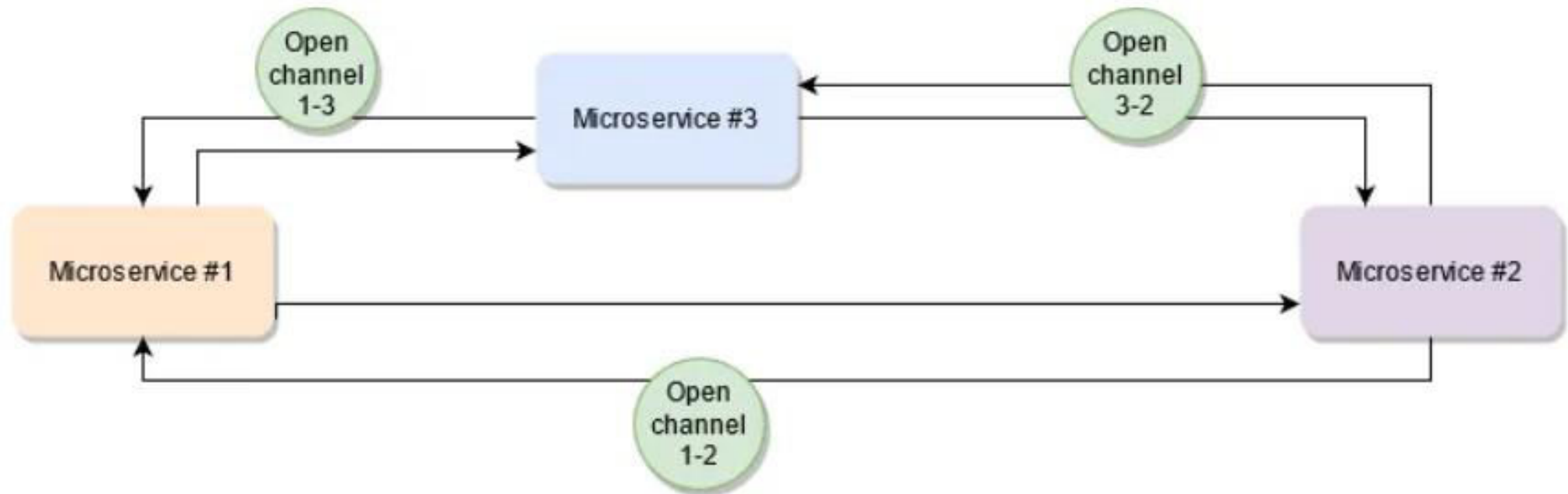


Asynchronous communication between microservices

Koenig-Solutions Pvt. Ltd.

Direct Socket Communication

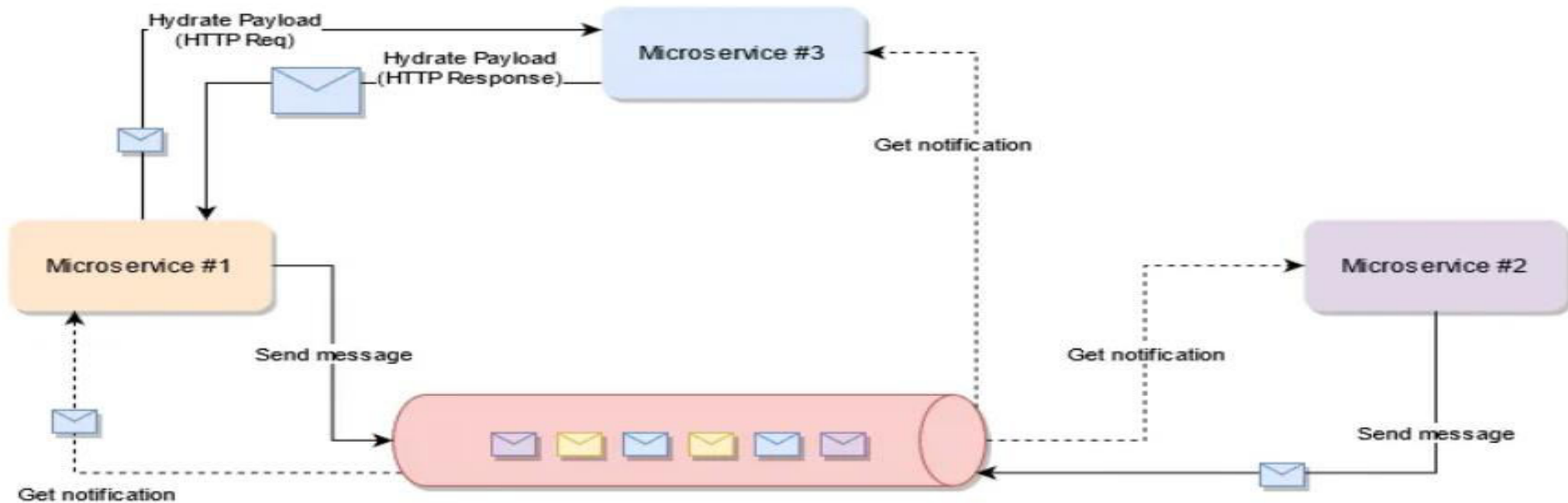
- At a first glance, the socket-based communication looks a lot like the client-server pattern implemented in HTTP, however, if you look closely there are some differences:



Open channels with sockets for microservice communication

Lightweight Events

- This pattern mixes the first two on this list. On one side, it provides a way of having multiple services communicate with each other through a message bus, thus allowing for asynchronous communication.



Lightweight events & hydration during microservice communication

Decomposing Monolith into Microservices

- The process of decomposition entails the partitioning of a monolithic application into microservices that are organized according to functional boundaries.
- The objective of this pattern is to enhance maintainability and resilience by enabling each microservice to operate autonomously.

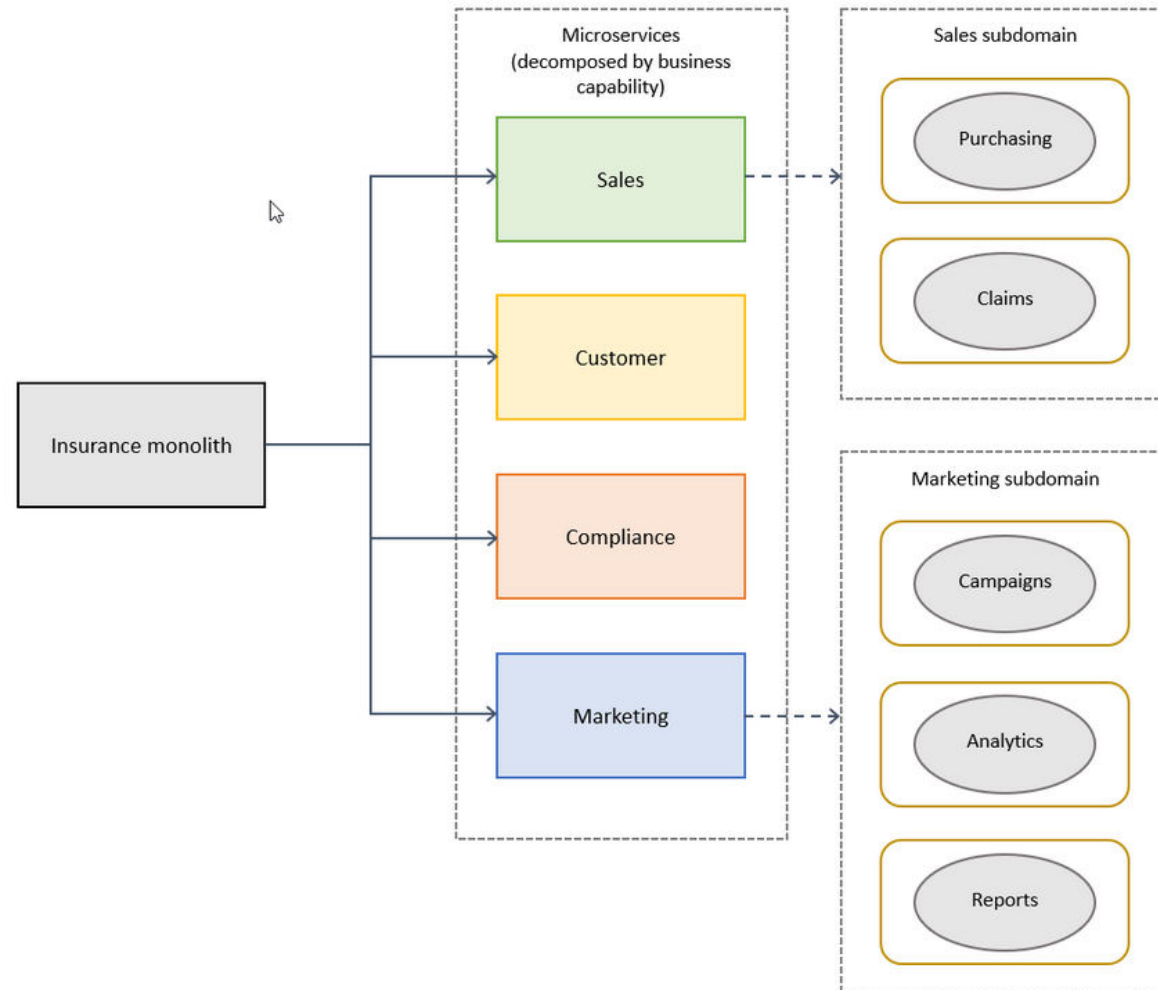
Decomposing Monolith into Microservices

- There are some patterns for decomposing the monolith to microservices:
 - Decompose by Business Capability
 - Decompose by Domain-Driven Design
 - Decompose by Transactions
 - Strangler fig Pattern
 - Service per team pattern
 - Branch by abstraction pattern

Decompose by Domain-Driven Design

- This task involves defining services that align with the subdomains of Domain-Driven Design (DDD).
- Domain-Driven Design establishes the application's domain or problem space. Domains have subdomains. Each subdomain is associated with a distinct segment of the business. Subdomain classifications include:
 - Core - The enterprise's core is the software's most valuable part.
 - Supporting- "Supporting" activities or features are related to core business operations but do not provide a competitive advantage. These solutions can be internal or outsourced.
 - Generic- Generic solutions use readily available software and are not tailored to a specific organization.

Decompose by Domain-Driven Design



Decompose by Domain-Driven Design

Advantages	Disadvantages
<ul style="list-style-type: none">• Loosely coupled architecture provides scalability, resilience, maintainability, extensibility, location transparency, protocol independence, and time independence.• Systems become more scalable and predictable.	<ul style="list-style-type: none">• Can create too many microservices, which makes service discovery and integration difficult.• Business subdomains are difficult to identify because they require an in-depth understanding of the overall business.



Understandings of Spring Framework and Spring Boot

Spring Framework

- Spring framework is an open source Java platform. It was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003.
- Spring is lightweight when it comes to size and transparency. The basic version of Spring framework is around 2MB.
- Spring is the most popular application development framework for enterprise Java. Millions of developers around the world use Spring Framework to create high performing, easily testable, and reusable code.

Intro to Spring Framework

- A Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.
- It enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs.
- This capability applies to the Java SE programming model and to full and partial Java EE.

Why I want to go to Spring Platform?

As an application developer, can use the Spring platform advantage:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

In Simple words, Spring is a

- Light Weighted
- Loosely Coupled architecture
 - Dependency Injection or Inversion of Control
- Application development framework used for J2EE to obtain it in more powerful
- Framework of Frameworks
 - Struts, Hibernate, JPA

Dependency Injection Container

- Core of the Spring Framework.
- The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction.
- The Spring container uses DI to manage the components that make up an application - Spring Beans

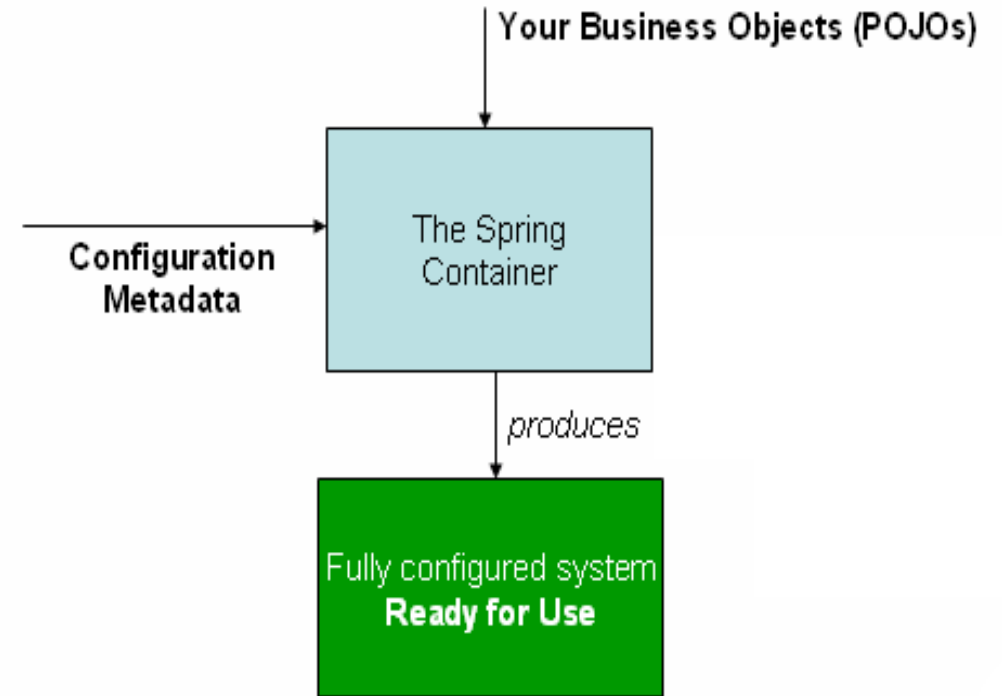
DI Container

- The container then injects those dependencies when it creates the bean.
- This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.
- Two Different Patterns:
 - Bean Factory Interface
 - `org.springframework.beans.factory.BeanFactory`
 - Application Context Interface
 - `org.springframework.context.ApplicationContext`

DI Container

Forms of metadata with the Spring container, see:

- **Annotation-based configuration:** Spring 2.5 introduced support for annotation-based configuration metadata.
- **Java-based configuration:** Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework



Configuring Metadata

- XML based metadata configuration
 - Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.
 - XML-based configuration metadata configures these beans as `<bean />` elements inside at top level `<beans />` element.
 - The `id` attribute is a string that identifies the individual bean definition.
 - The `class` attribute defines the type of the bean and uses the fully qualified classname.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

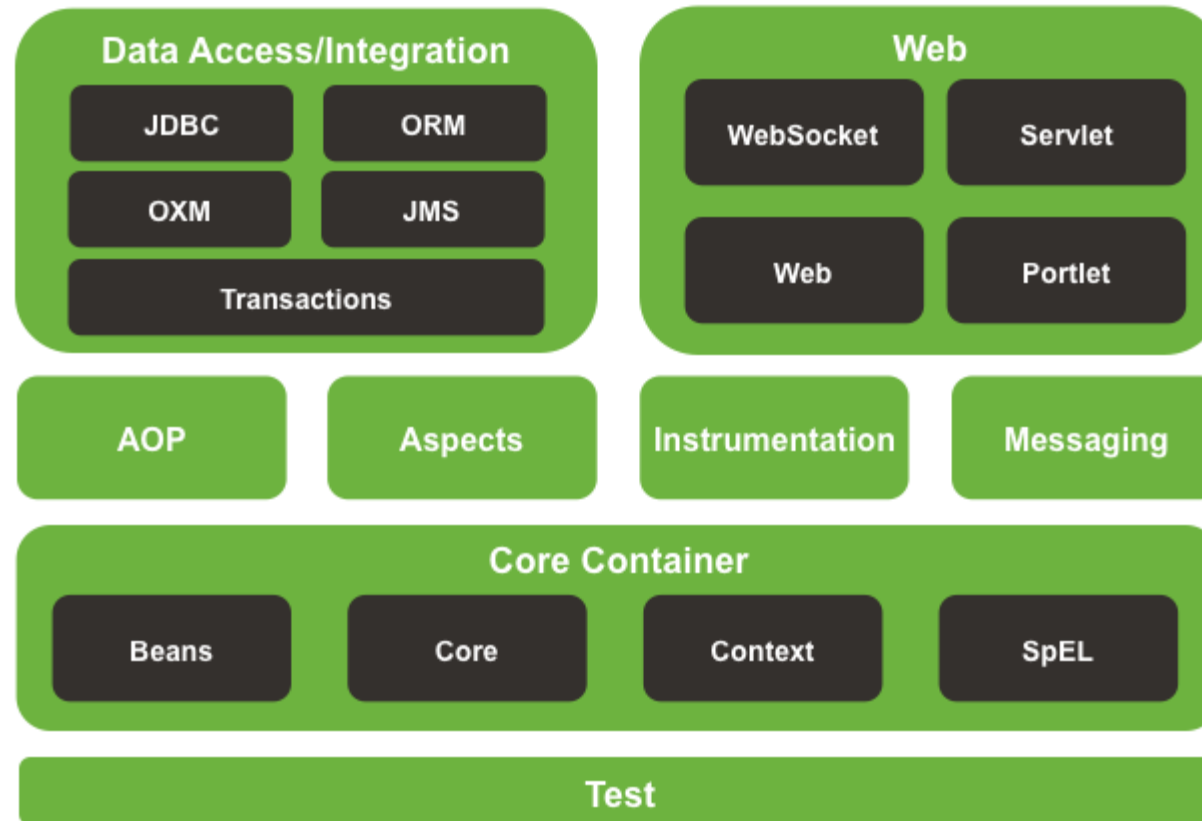
    <!-- more bean definitions go here -->

</beans>
```

Modules of Spring Framework



Spring Framework Runtime



Modules Covered On Spring Framework / Spring Core

- Spring AOP
- Spring Testing
 - jUnit5
- Spring JDBC Template
- Spring Transactions

What Problems Does AOP Solve?

- Aspect-Oriented Programming (AOP) enables modularization of cross-cutting concerns
 - The code for a cross-cutting concern is in a single place, in a module - in Java, a class represents a module
- Perform a role-based security check before every application method

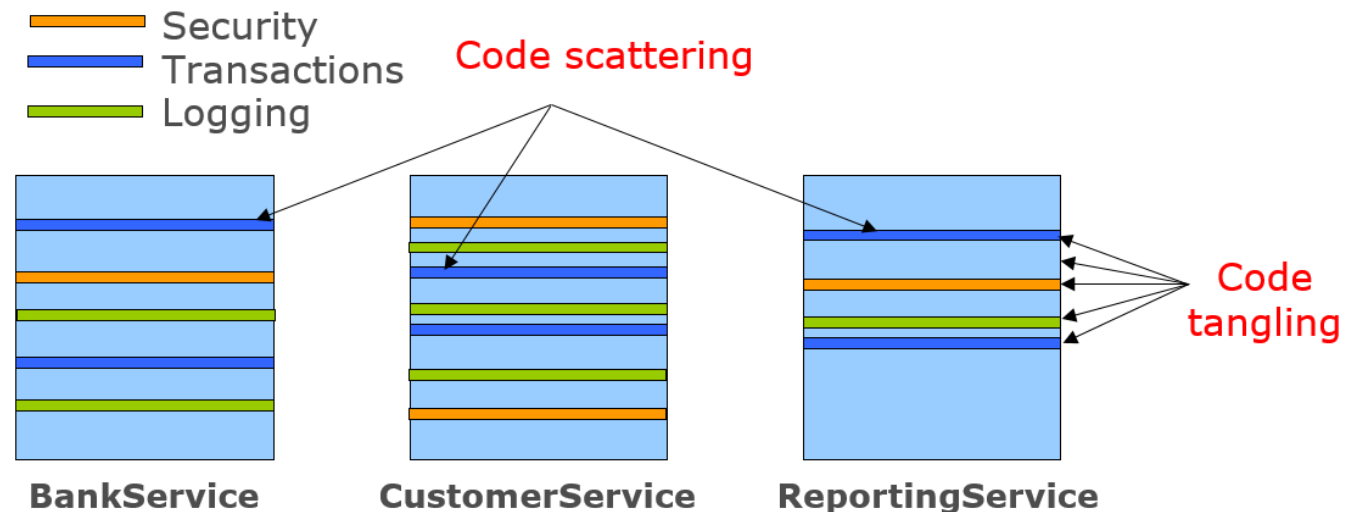
A sign this requirement is a cross-cutting concern

What are Cross-Cutting Concerns?

- Generic functionality that is needed in many places in your application
- Examples of cross-cutting concerns
 - Logging and Tracing
 - Transaction Management
 - Security
 - Caching
 - Error Handling
 - Performance Monitoring
 - Custom Business Rules

Implementing Cross Cutting Concerns Without Modularization

- Failing to modularize cross-cutting concerns leads to two problems
 - Code tangling
 - Coupling of concerns
 - Code scattering
 - The same concern spread across modules



Spring Testing – JUnit5

- JUnit 5 is composed of several different modules from three different sub-projects.

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

- The JUnit Platform serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform.
- JUnit Jupiter is the combination of the programming model and extension model for writing tests and extensions in JUnit 5.
- JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform. It requires JUnit 4.12 or later to be present on the class path or module path.

Spring's JdbcTemplate



- Greatly simplifies use of the JDBC API
 - Eliminates repetitive boilerplate code
 - Alleviates common causes of bugs
 - Handles **SQLExceptions** properly
- Without sacrificing power
 - Provides full access to the standard JDBC constructs

“Life is too short to write JDBC!”

- *Rod Johnson co-founder of Spring*

JdbcTemplate in a Nutshell

```
int count = jdbcTemplate.queryForObject(  
    "SELECT COUNT(*) FROM CUSTOMER", Integer.class);
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling exceptions
- Release of the connection

**All handled
by Spring**

Spring Boot

- Spring Boot is built on the top of the spring and contains all the features of spring.
- And is becoming favourite of developer's these days because of it's a rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and set up.
- Spring Boot is a microservice-based framework and making a production-ready application in it takes very less time
- Major Features are:
 - Dependency Management
 - Auto-Configuration
 - Package and Runtime
 - Integration Testing

Modules Covered On Spring Boot

- Spring Web MVC
- Spring Data JPA
- Spring REST
- Spring Boot Testing [Unit and Integration Testing]
- Spring Security
- Spring Actuator

Modules Covered On Spring Cloud

- Spring Cloud Configuration
- Spring Cloud Gateway – Netflix Eureka
- Spring Cloud Circuit Breaker – Resilience4J
- Spring Cloud Retry – Resilience4J
- Spring Cloud Distributed Tracing – Zipkin
- Spring Boot Messaging – Kafka / RabbitMQ
- Spring Boot Integration - Redis

Web Applications with Spring Boot

Getting Started with Spring MVC &
REST

1.18.5

KOENIG
step forward

Agenda

- **Spring Boot and Spring MVC**
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- Spring Boot Developer Tools



Web Layer Support in Spring MVC and Spring Boot

- What is Spring MVC?
 - Web framework based on:
 - *Model/View/Controller (MVC)* pattern
 - POJO programming
 - Testable components
 - Uses Spring for configuration
 - Supports Server-side web rendering & REST
- Spring Boot can help you create web applications easily
 - Based on Spring MVC

We will focus on REST.

Types of Spring MVC Applications

- Web Servlet
 - Traditional approach
 - Based on Java EE Servlet Specification
 - Servlets
 - Filters
 - Listeners
 - Etc.
- Web Reactive
 - Newer, more efficient
 - Non-blocking approach
 - Netty, Undertow, Servlet 3.1+
 - Requires knowledge of Reactive programming (see appendix)

We will focus on traditional Web Servlet-based approach.

Traditional or Embedded Servlet Container

- Spring Boot supports embedded servlet containers
 - Packaging an embedded container within a deployment artifact in a 'fat' JAR
 - Run web applications from command line!
- Spring Boot can also implement traditional structure needed by Web containers
 - WAR packaging

We will focus on embedded approach.

Spring Web “Starter” Dependencies

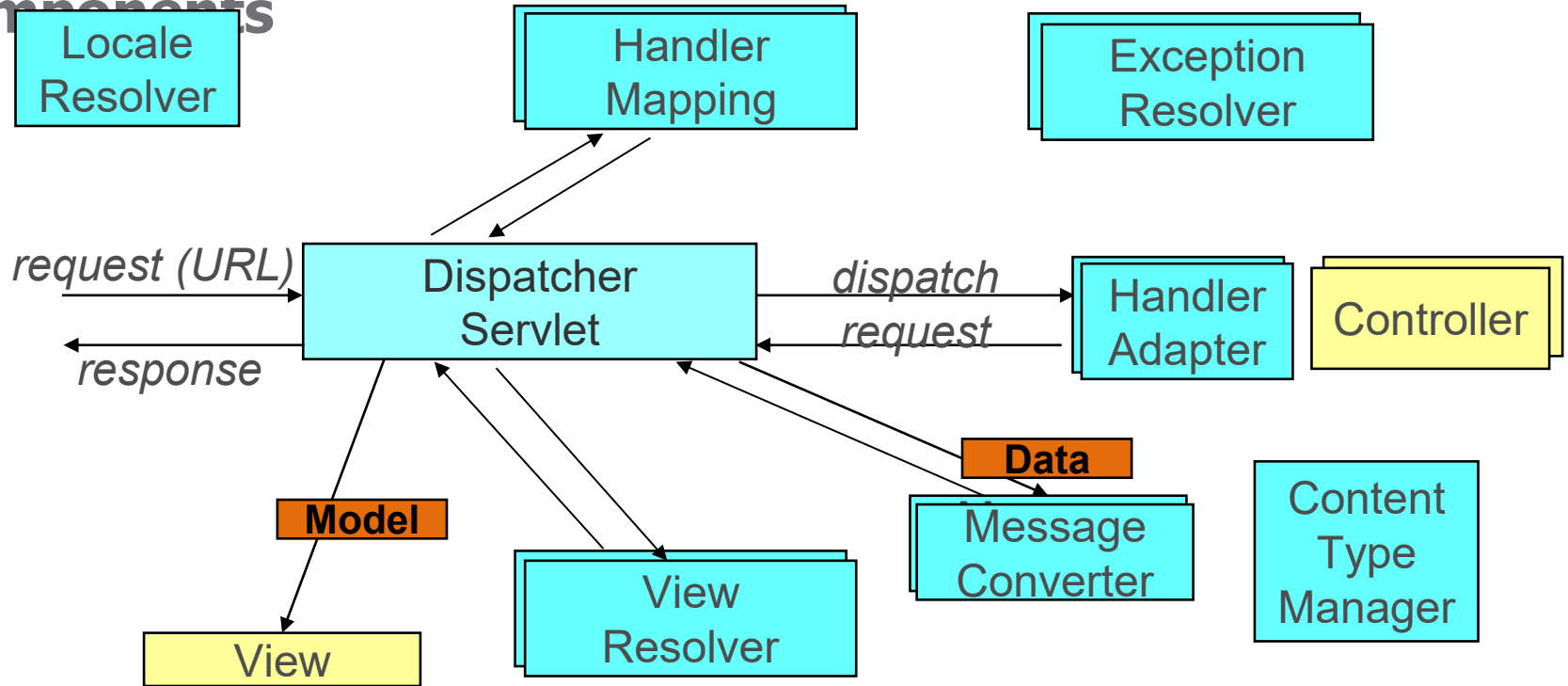
- Everything you need to develop a Spring MVC application

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

Resolves

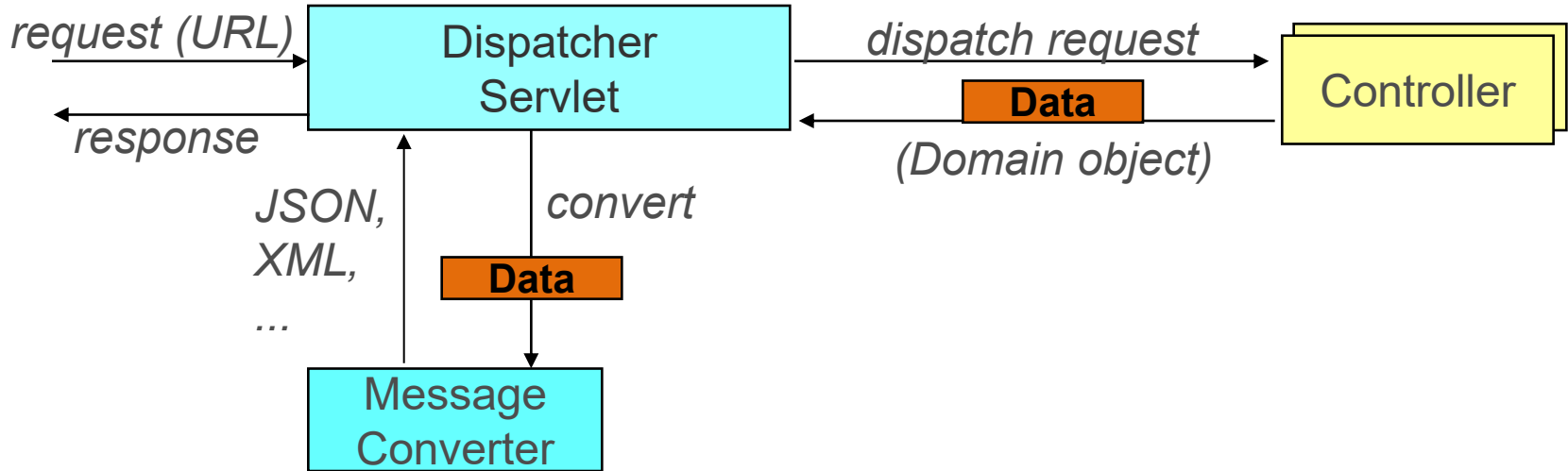
spring-web.jar
spring-webmvc.jar
spring-boot-starter.jar
jackson.jar*
tomcat-embed.jar*
...

At Startup Time, Spring Boot Creates Spring MVC Components



- Developers need to create only controllers and views (yellow-colored)

Request Processing Lifecycle - REST



- Developers need to create only controllers (yellow-colored)
- Common message-converters setup automatically – *if* found on the classpath
 - Jackson for JSON/XML, JAXB for XML, GSON for JSON ...

Controller Implementation

- Controllers are annotated with **@Controller**
 - **@GetMapping** tells Spring what method to use to process HTTP GET requests
 - **@ResponseBody** defines a *REST* response
 - Turns *off* the View handling subsystem

@Controller

```
public class AccountController {
```

```
    @GetMapping("/accounts")
```

```
    public @ResponseBody List<Account> list() {...}
```

```
}
```

@Controller is an **@Component**
so *will* be found by component-scanner

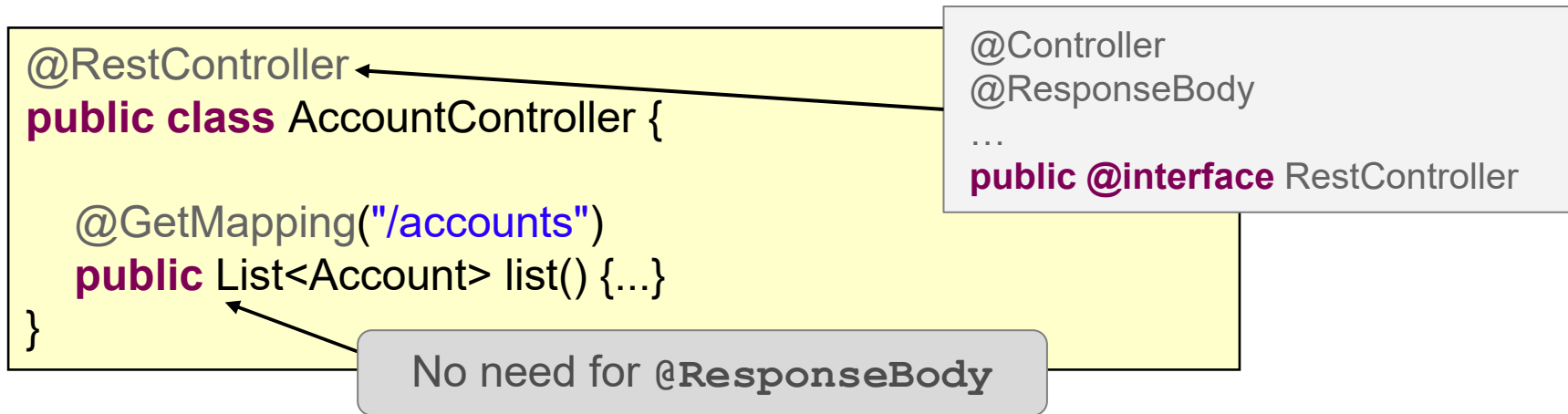
Example of calling URL: *http://localhost:8080 / accounts*

server

request mapping

@RestController Convenience

- Convenient “composed” annotation
 - Incorporates @Controller and @ResponseBody
 - Methods assumed to return REST response-data



All examples assume @RestController from now on

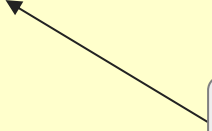
Controller Method Arguments

- You pick the arguments you need, Spring injects them
 - `HttpServletRequest`, `HttpSession`, `Principal`, `Locale`, etc.
 - See [Spring Reference, Controller Method Arguments](#)

```
@RestController
public class AccountController {

    // Retrieve accounts of currently logged-in user
    @GetMapping("/accounts")
    public List<Account> list(Principal user) {
        ...
    }
}
```

Injected by Spring



Extracting Request Parameters

- Use `@RequestParam` annotation
 - Extracts request parameters from the request URL
 - Performs type conversion

```
@RestController
public class AccountController {

    @GetMapping("/account")
    public List<Account> list(@RequestParam("userid") int userId) {
        ... // Fetch and return accounts for specified user
    }
}
```

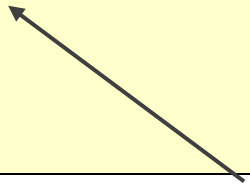
Example of calling URL:

<http://localhost:8080/account?userid=1234>

URI Templates

- Values can be extracted from request URL
 - *Based on URI Templates*
 - Use {...} placeholders and `@PathVariable`

```
@RestController
public class AccountController {
    @GetMapping("/accounts/{accountId}")
    public Account find(@PathVariable("accountId") long id) {
        ... // Do something
    }
}
```



Example of calling URL: <http://localhost:8080/accounts/98765>

Naming Conventions

- Drop annotation value ***if*** it matches method parameter name

```
// Get user's overdrawn accounts
@GetMapping("/accounts/{userId}")
public List<Account> list(@PathVariable long userId,
                        @RequestParam boolean overdrawn)
```

<http://.../accounts/1234?overdrawn=true>



<https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html>

Method Signature Examples

Example URLs

```
@GetMapping("/accounts")  
public List<Account> getAccounts()
```

<http://localhost:8080/accounts>

```
@GetMapping("/orders/{id}/items/{itemId}")  
public OrderItem item( @PathVariable("id") long orderId,  
                       @PathVariable int itemId,  
                       Locale locale,  
                       @RequestHeader("user-agent") String agent )
```

<http://.../orders/1234/items/2>

```
@GetMapping("/suppliers")  
public List<Supplier> getSuppliers(  
    @RequestParam(required=false) Integer location,  
    Principal user,  
    HttpSession session )
```

<http://.../suppliers?location=12345>

Null if not specified

HTTP GET: Fetch a Resource

- *Requirement*
 - Respond *only* to GET requests
 - Return requested data in the HTTP Response
 - Determine requested response format

```
GET /store/orders/123
Host: shop.spring.io
Accept: application/json, ...
...
```

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json

{
  "id": 123,
  "total": 200.00,
  "items": [ ... ]
}
```

Generating Response Data



- **The Problem**


- HTTP GET needs to return data in response body
 - Typically JSON or XML
- Developers prefer to work with Java objects
- Developers want to avoid manual conversion

- **The Solution**

- Object to text conversion
 - Message-Converters
- Annotate response data with **@ResponseBody**

```
HTTP/1.1 200 OK
Date: ...
Content-Length: 756
Content-Type: application/json

{
    "id": 123,
    "total": 200.00,
    "items": [ ... ]
}
```

A red arrow pointing from the right side of the slide towards the JSON body of the HTTP response.

HttpMessageConverter



- Converts HTTP request/response body data
 - XML: JAXP Source, JAXB2 mapped object*, Jackson-Dataformat-XML*
 - GSON*, Jackson JSON*
 - Feed data* such as Atom/RSS
 - Google protocol buffers*
 - Form-based data
 - **Byte[]**, **String**, **BufferedImage**
- Automatically setup by Spring Boot (except protocol buffers)
 - Manual configuration also possible

* Requires 3rd party libraries on classpath

What Return Format? *Accept* Request Header

```
@GetMapping("/store/orders/{id}")  
public Order getOrder(@PathVariable("id") long id) {  
    return orderService.findOrderById(id);  
}
```

```
GET /store/orders/123  
Host: shop.spring.io  
Accept: application/xml  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 1456  
Content-Type: application/xml
```

```
<order id="123">  
...  
</order>
```

```
GET /store/orders/123  
Host: shop.spring.io  
Accept: application/json  
...
```

```
HTTP/1.1 200 OK  
Date: ...  
Content-Length: 756  
Content-Type: application/json  
{  
    "id": 123,  
    "total": 200.00,  
    "items": [ ... ]  
}
```

Customizing GET Responses: ResponseEntity



- Build **GET** responses explicitly
 - More control
 - You can set headers or control response content

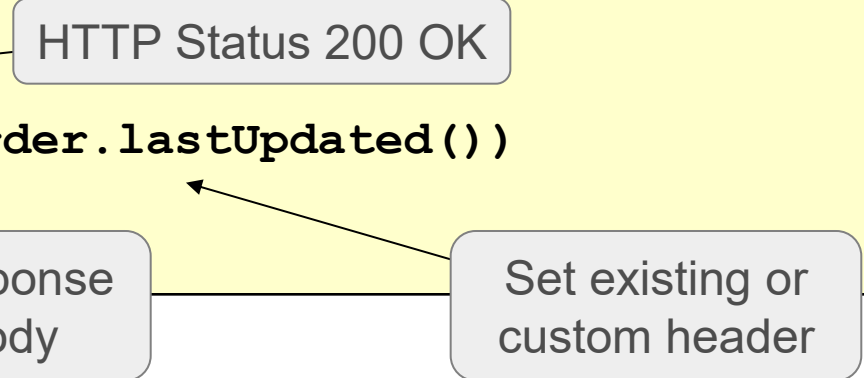
```
// ResponseEntity supports a "fluent" API for creating a  
// response. Used to initialize the HttpServletResponse.  
ResponseEntity<String> response =  
    ResponseEntity.ok()  
        .contentType(MediaType.TEXT_PLAIN)  
        .body("Hello Spring");
```



Subclasses `HttpEntity` with fluent API

Setting Response Data with Domain object

```
@GetMapping("/store/orders/{id}")  
public ResponseEntity<Order> getOrder(@PathVariable long id) {  
    Order order = orderService.find(id);  
  
    return ResponseEntity  
        .ok()   
        .lastModified(order.lastUpdated())  
        .body(order)   
};
```



HTTP Status 200 OK

Response body

Set existing or custom header

Spring Boot for Web Applications

- Use **spring-boot-starter-web**
 - Ensures Spring Web and Spring MVC are on classpath
- Spring Boot auto-configuration for Web applications
 - Sets up a **DispatcherServlet**
 - Sets up internal configuration to support controllers
 - Sets up default resource locations (images, CSS, JavaScript)
 - Sets up default Message Converters
 - And much, much more



Spring Boot Provides Web Container (Servlet Container)

- By default Spring Boot starts up an embedded web container
 - You can run a web application from the command line!
 - Tomcat is the default web container



Spring Boot's default behavior. Traditional WAR deployment available also.

Alternative Web Containers: Jetty, Undertow

- *Example:* Jetty instead of Tomcat

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Excludes Tomcat

Adds Jetty

Jetty automatically detected and used!

Running Within a Web Container (traditional)

Sub-classes Spring's *WebApplicationInitializer*
– called by the web container (Servlet 3+ required)

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

    // Specify the configuration class(es) to use
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }
}
```

Don't forget to change artifact type to war



The above requires **no** *web.xml* file

Configure for a WAR *or* a JAR

```
@SpringBootApplication
```

```
public class Application extends ServletInitializer {
```

```
    protected void configure(
        SpringApplication.Builder application) {
        return application.sources(Application.class);
    }
```

Web container
support

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

main() method
run from CLI

```
}
```

JAR or WAR

- By default, both JAR and WAR forms can run from the command line
 - Based on embedded Tomcat (or Jetty) JARs
- However, embedded Tomcat JARs can cause conflicts when running WAR inside traditional web container
 - Example: running within different version of Tomcat
- Best Practice: Mark Tomcat dependencies as *provided* when building WARs for traditional containers

Spring Boot JAR Files

- Using Boot's plugin, `mvn package` or `gradle assemble` produces *two* JAR files

22M `yourapp-0.0.1-SNAPSHOT.jar`

10K `yourapp-0.0.1-SNAPSHOT.jar.original`

“fat” JAR

Traditional JAR

- “Fat” JAR executable with embedded Tomcat
 - using `java -jar yourapp.jar`



For details: <https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/> and <https://docs.spring.io/spring-boot/docs/current/gradle-plugin/reference/htmlsingle/>

Agenda

- Spring Boot and Spring MVC
- Details
 - Request Processing Lifecycle
 - Controllers
 - Message Converters
- JAR or WAR configurations
- **Spring Boot Developer Tools**
- Quick Start
- Lab
- Spring MVC Without Boot



Spring Boot Developer Tools

- A set of tools to help make Spring Boot development easier
 - Automatic restart - any time a class file changes (on re-compile) - faster than “cold restart”
 - Automatically disabled when it considers the app is running in “production”

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
</dependencies>
```

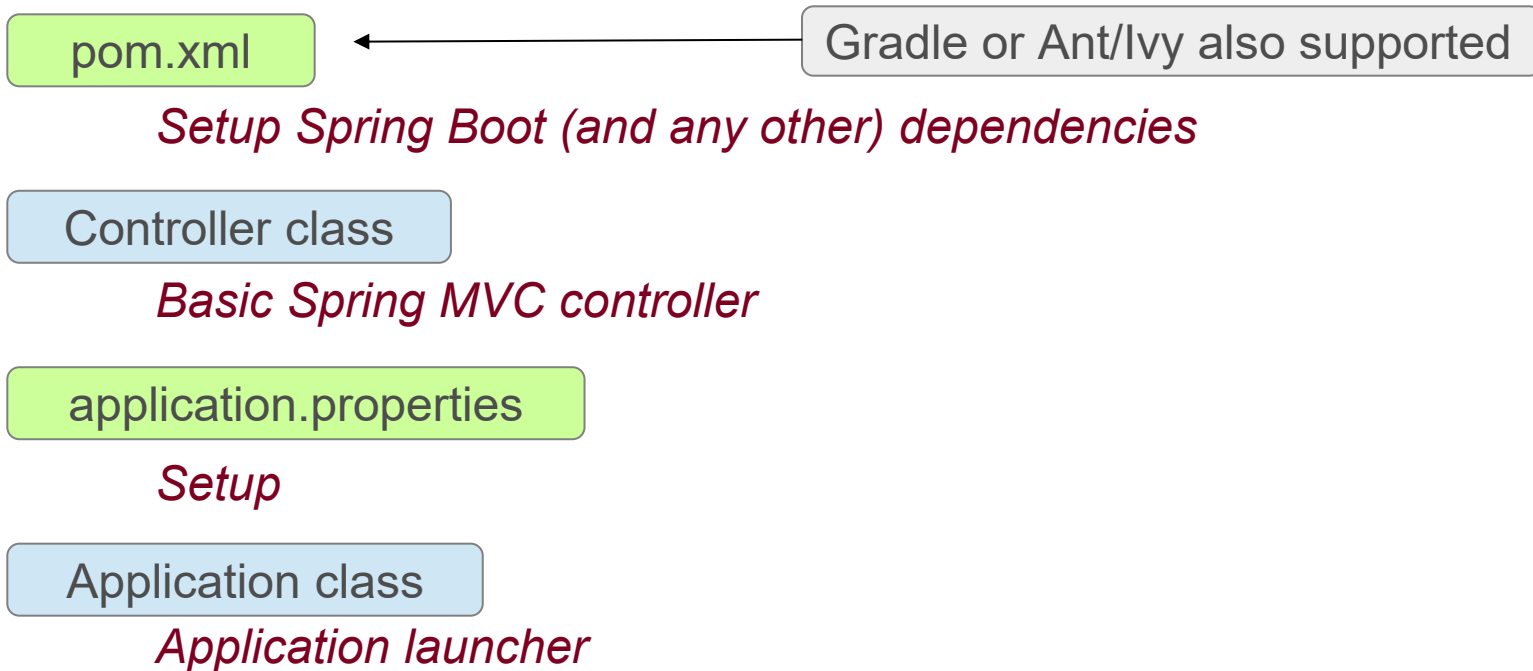
Prevents devtools from being transitively applied to other modules that use your project

Revision: What We Have Covered

- Spring Web applications have *many* features
 - The **DispatcherServlet**
 - Setup Using Spring Boot
 - Writing a Controller
 - Using Message Converters
 - JARs vs WARs
- *But you don't need to worry about **most** of this to write a simple Spring Boot Web application ...*
 - Typically you just write Controllers
 - Set some properties

Quick Start

- Only a few files to get a running Spring Web application



Testing Spring Applications

Testing in General, JUnit 5, Spring and JUnit, Profile-based Testing, Database Testing

1.18.5

Agenda

- **JUnit 5**
- Integration Testing with Spring
- Testing with Profiles
- Testing with Databases



- Labs use JUnit 5 for testing
 - JUnit 5 support is a major feature of Spring 5.3
 - JUnit 5 is the default JUnit version from Spring Boot 2.6
 - Requires Java 8+ at runtime
 - Leverages Lambda
- Components
 - **JUnit Platform**
 - A foundation for launching testing frameworks on the JVM
 - **JUnit Jupiter**
 - An extension model for writing tests and extensions in JUnit 5
 - **JUnit Vintage**
 - A *TestEngine* for running JUnit 3 & 4 tests on the platform

JUnit 5: New Programming Models

- Replaces JUnit 4 annotations

— @Before	→ @BeforeEach
— @BeforeClass	→ @BeforeAll
— @After	→ @AfterEach
— @AfterClass	→ @AfterAll
— @Ignore	→ @Disabled

JUnit 5

- Introduces new annotations

- @DisplayName
- @Nested
- @ParameterizedTest
- ...

- *Use right annotations from correct package*
- JUnit 5 ignores all JUnit 4 annotations

Writing Test – JUnit 5 Style

New package

```
import static org.junit.jupiter.api.Assertions.fail;
```

```
import org.junit.jupiter.api.AfterAll;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Disabled;  
import org.junit.jupiter.api.Test;
```

```
class StandardTests {
```

```
    @BeforeAll  
    static void initAll() {  
    }  
}
```

Replaces
@BeforeClass

```
    @BeforeEach  
    void init() {  
    }  
    ...
```

Replaces
@Before

```
    @Test  
    void succeedingTest() {  
    }
```

```
    @Test  
    void failingTest() {  
        fail("a failing test");  
    }
```

Replaces
@Ignore

```
    @Test  
    @Disabled("for demo purposes")  
    void skippedTest() {  
        // not executed  
    }  
}
```

Unit Testing

Unit Testing *Without Spring*

- Unit Testing
 - Tests one unit of functionality
 - Keeps dependencies minimal
 - Isolated from the environment (including Spring)
 - Uses “test doubles” for dependencies
 - Stubs and/or Mocks
 - *See Appendix for more details*

Integration Testing

Integration Testing *With Spring*

- Integration Testing
 - Tests the interaction of multiple units working together
 - All should work individually first (unit tests showed this)
 - Tests that involve Spring
- Tests application classes in context of their surrounding infrastructure
 - Out-of-container testing, no need to run up full App. Server
 - Infrastructure may be “scaled down”
 - Use ActiveMQ instead of commercial messaging servers
 - Test Containers
 - Use lightweight, throwaway instances of common databases or anything else that can run in a Docker container

Spring Support for Testing

- Spring has rich testing support
 - Based on **TestContext** framework
 - Defines an **ApplicationContext** for your tests
 - Use **@ContextConfiguration**
 - With a set of Spring configuration classes
- Packaged as a separate module
 - **spring-test.jar**



<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#integration-testing>
<https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#testcontext-framework>

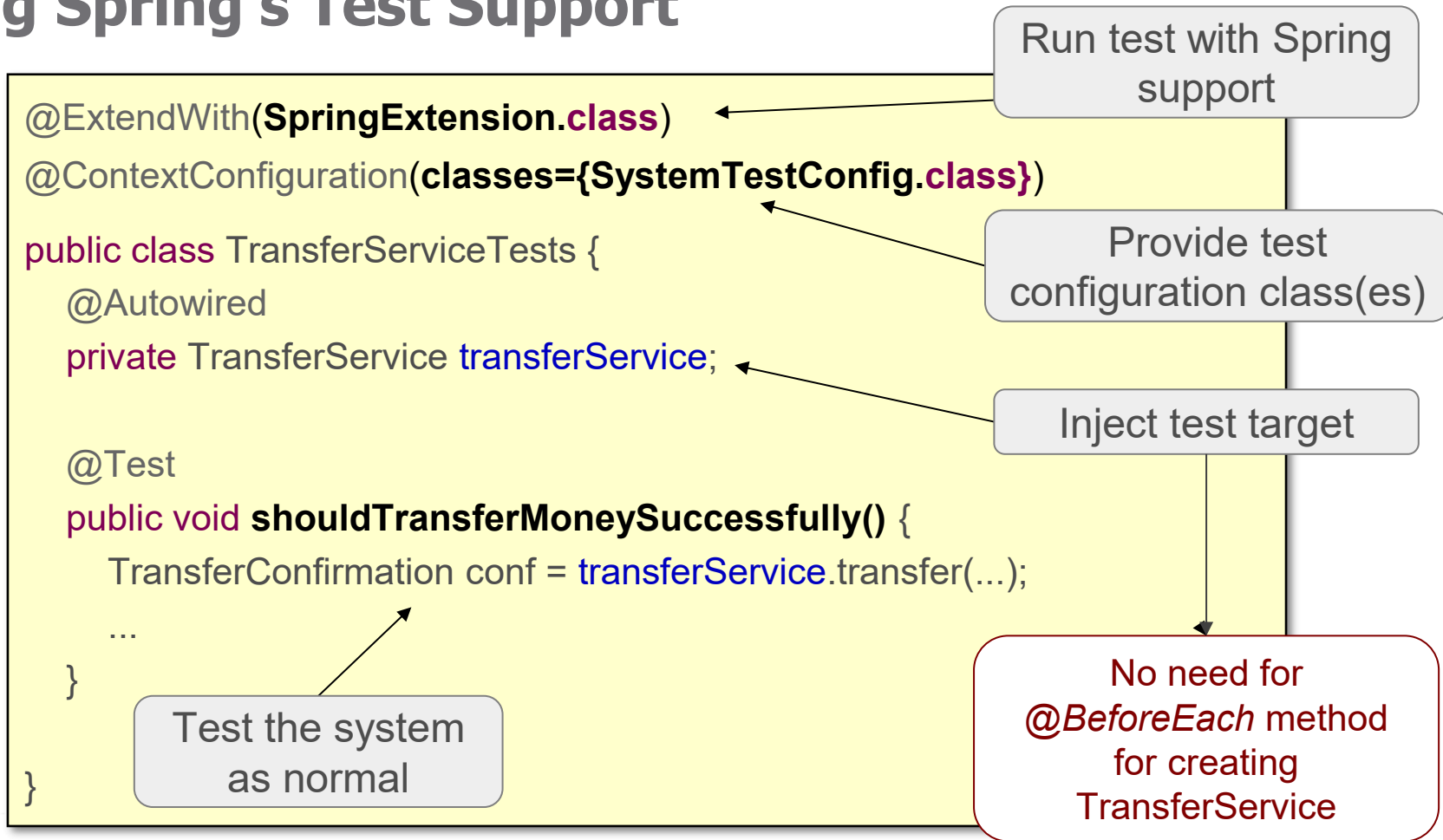
@ExtendWith in JUnit 5

- JUnit 5 has extensible architecture via @ExtendWith
 - Replaces JUnit 4's @RunWith
 - JUnit 5 supports *multiple* extensions - hence the name
- Spring's extension point is the SpringExtension class
 - A Spring aware test-runner



See *Appendix* (end of this section) for Spring's JUnit 4 support.

Using Spring's Test Support



@SpringJUnit4Config



- **@SpringJUnit4Config** is a “*composed*” annotation that combines
 - **@ExtendWith(SpringExtension.class)** from JUnit 5
 - **@ContextConfiguration** from Spring

Recommended: Use
composed annotation

```
@SpringJUnit4Config(SystemTestConfig.class)
public class TransferServiceTests {
    ...
}
```

Alternative Autowiring for Tests

```
@SpringJUnitConfig(SystemTestConfig.class)
```

```
public class TransferServiceTests {
```

```
    // @Autowired
```

```
    // private TransferService transferService;
```

```
    @Test
```

```
    public void shouldTransferMoneySuccessfully
```

```
        (@Autowired TransferService transferService) {
```

```
        TransferConfirmation conf = transferService.transfer(...);
```

```
        ...
```

```
    }
```

```
}
```

No longer required –
dependency injected as
test method *argument*

NOTES:

- Test works either way – your choice
- This use of Autowired *unique to tests*

Including Configuration as an Inner Class

```
@SpringJUnitConfig
public class JdbcAccountRepoTest {

    @Test
    public void shouldUpdateDatabaseSuccessfully() {...}

    @Configuration
    @Import(SystemTestConfig.class)
    static class TestConfiguration {
        @Bean public DataSource dataSource() { ... }
    }
}
```

Don't specify any
config classes

Looks for configuration
embedded in test class

Override a bean with
a test alternative

Multiple Test Methods Share same ApplicationContext

```
@SpringJUnitConfig(classes=SystemTestConfig.class)
public class TransferServiceTests {
    @Autowired
    private TransferService transferService;

    @Test
    public void successfulTransfer() { ... }

    @Test
    public void failedTransfer() { ... }
}
```

ApplicationContext
instantiated only *once*

Both tests share *same*
cached **Application-
Context** & use *same*
TransferService bean



Most Spring Beans are *stateless/immutable* singletons, never modified during any test. No need for a new context for each test.

@DirtyContext

- Forces context to be closed at end of test method
 - Allows testing of @PreDestroy behavior
- Next test gets a *new* Application Context
 - Cached context destroyed, new context cached instead

```
@Test
@DirtyContext
public void testTransferLimitExceeded() {
    transferService.setMaxTransfers(0);
    ... // Do a transfer, expect a failure
}
```

Context closed
and destroyed at
end of test

@TestPropertySource

- Custom properties *just* for testing
 - Specify one or more properties
 - Has higher precedence than sources
 - Specify location of one or more properties files to load
 - Defaults to looking for `[classname].properties`

```
@SpringJUnitConfig(SystemTestConfig.class)
@TestPropertySource(properties = { "username=foo", "password=bar" },
                    locations = "classpath:/transfer-test.properties")
public class TransferServiceTests {
    ...
}
```

Benefits of Testing with Spring

- No need to deploy to an external container to test application functionality
 - Run everything quickly inside your IDE or CI/CD pipeline
 - Supports *Continuous Integration* testing
- Allows reuse of your configuration between test and production environments
 - Application configuration is typically reused
 - Infrastructure configuration is environment-specific
 - DataSources
 - JMS Queues

Activating Profiles For a Test

- **@ActiveProfiles** inside the test class
 - Define one or more active profiles
 - Beans associated with those active profiles are instantiated
 - Also beans not associated with *any* profile
- Example: Two profiles activated – *jdbc* **and** *dev*

```
@SpringJUnitConfig(DevConfig.class)
@ActiveProfiles( { "jdbc", "dev" } )

public class TransferServiceTests { ... }
```

Profiles Activation with JavaConfig

- **@Profile** on *@Configuration* class or any of its *@Bean* methods

```
@SpringJUnitConfig(DevConfig.class)
@ActiveProfiles("jdbc")
public class TransferServiceTests
{...}
```

```
@Configuration
@Profile("jdbc")
public class DevConfig {

    @Bean
    public ... {...}

}
```

```
@Configuration
public class DevConfig {

    @Profile("jdbc")
    @Bean
    public ... {...}

}
```



Only beans matching an active profile or with *no* profile are loaded

Profiles Activation with Annotations

- **@Profile** on a *Component* class

```
@SpringJUnitConfig(DevConfig.class)
@ActiveProfiles("jdbc")
public class TransferServiceTests {
    ...
}
```

```
@Repository
@Profile("jdbc")
public class JdbcAccountRepository {
    ...
}
```



Only beans with current profile / no profile are component-scanned

Testing with Databases

- Integration testing against SQL database is common
- In-memory databases useful for this kind of testing
 - No prior install needed
- Common requirement: populate DB before test runs
 - Use the `@Sql` annotation:

```
@Test
@Sql ( "/testfiles/test-data.sql" )
public void successfulTransfer() {
    ...
}
```

Run this SQL script *before* this test method executes.

@Sql Examples

Run these scripts before *each* @Test method *unless* a method is annotated with its own @Sql

```
@SpringJUnitConfig(...)
@Sql( { "/testfiles/schema.sql", "/testfiles/load-data.sql" } )
public class MainTests {

    // schema.sql and load-data.sql only run before this test
    @Test
    public void success() { ... }

    @Test // Overrides to use own scripts
    @Sql ( scripts="/testfiles/setupBadTransfer.sql" )
    @Sql ( scripts="/testfiles/cleanup.sql",
            executionPhase=Sql.ExecutionPhase.AFTER_TEST_METHOD )
    public void transferError() { ... }
}
```

Run *before* @Test method

... run *after*
@Test method

@Sql Options

- When/how does the SQL run?
 - *executionPhase*: before (default) or after test method
 - *config*: Options to control SQL scripts
 - What to do if script fails? **FAIL_ON_ERROR**, **CONTINUE_ON_ERROR**, **IGNORE_FAILED_DROPS**, **DEFAULT***
 - SQL syntax control: comments, statement separator

```
@Sql( scripts = "/test-user-data.sql",  
      config = @SqlConfig(errorMode = ErrorMode.FAIL_ON_ERROR,  
                          commentPrefix = "//", separator = "@@") )
```

***DEFAULT** = whatever @Sql defines at class level, otherwise **FAIL_ON_ERROR**

Summary

- Testing is an *essential* part of any development
- Unit testing tests a class in isolation
 - External dependencies should be minimized
 - Consider creating stubs or mocks to unit test
 - *You don't need Spring to unit test*
- Integration testing tests the interaction of multiple units working together
 - Spring provides good integration testing support
 - Profiles for different test & deployment configurations
 - Built-in support for testing with Databases

Advanced Testing with Spring Boot and MockMVC Testing

Leveraging Spring Boot
enhancements for simplified
integration and unit testing

1.18.5

What is Spring Boot Testing Framework?

- Built on the top of Spring Testing Framework
- Provides a set of annotations and utilities for testing
 - `@SpringBootTest`
 - `@WebMvcTest`, `@WebFluxTest`
 - `@DataJpaTest`, `@DataJdbcTest`, `@JdbcTest`,
`@DataMongoTest`, `@DataRedisTest`
 - `@MockBean`

How to get Started? Add Spring Boot Test Starter

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

Testing Dependencies with spring-boot-starter-test

- **JUnit:** JUnit 5 is default version (from Spring Boot 2.2)
- **Spring Test & Spring Boot Test:** Testing annotations
- **AssertJ:** A fluent assertion library
- **Hamcrest:** A library of matcher
- **Mockito:** A Java mocking framework
- **JSONassert:** An assertion library for JSON
- **JsonPath:** XPath for JSON

Integration Testing with @SpringBootTest

- Automatically searches for `@SpringBootTestConfiguration`
 - An alternative to `@ContextConfiguration` for creating application context for testing
 - Use `@SpringBootTest` for integration testing and use `@ContextConfiguration` for slice testing
- Provides support for different `webEnvironment` modes
 - `RANDOM_PORT`, `DEFINED_PORT`, `MOCK`, `NONE`
- The embedded server gets started by the testing framework
 - When `RANDOM_PORT`, `DEFINED_PORT` are used
 - Integration testing can be done as part of CI/CD pipeline
- Auto-configures a `TestRestTemplate`

Code Example with *TestRestTemplate*

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class AccountClientBootTests {

    @Autowired
    private TestRestTemplate restTemplate;

    // Test code

}
```

Knows the “random”
port to talk to

Code Example with *TestRestTemplate* : Test code

@Test

public void addAndDeleteBeneficiary() {

String addUrl = **"/accounts/{accountId}/beneficiaries"**;

Relative path

URI newBeneficiaryLocation = restTemplate.postForLocation(**addUrl**, "David", 1);

Beneficiary newBeneficiary

= restTemplate.getForObject(newBeneficiaryLocation, Beneficiary.class);

assertThat(newBeneficiary.getName()).isEqualTo("David");

restTemplate.delete(newBeneficiaryLocation);

ResponseEntity<Beneficiary> response

= restTemplate.getEntity(newBeneficiaryLocation, Beneficiary.class);

assertThat(**response.getStatusCode()**).isEqualTo(HttpStatus.NOT_FOUND);

}

Response status check

Need for Spring MVC Testing

- Consider the controller below, how can you verify:
 - `@PutMapping` results in a correct URL mapping?
 - `@PathVariable` mapping is working?
 - Account is correctly mapped from incoming JSON / XML?
 - Returned status is HTTP 204?
 - Any exception is handled as expected?

```
@PutMapping("/account/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT) // 204
public void updateOrder(@RequestBody Account account, @PathVariable long id) {
    accountManager.update(id, account);
}
```

MVC Test Framework Overview



SPRING TEST
FRAMEWORK

- Part of Spring Framework
 - Found in `spring-test.jar`
- **Goal:** Provide first-class support for testing Spring MVC code
 - Process requests through DispatcherServlet
 - Does *not* require running Web container to test
 - No need to coordinate server URL / port with test code



See: [Spring Framework Reference, Spring MVC Test Framework](https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#spring-mvc-test-framework)

<https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html#spring-mvc-test-framework>

Example: MockMvc Test

```
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@SpringBootTest(webEnvironment = WebEnvironment.MOCK)
@AutoConfigureMockMvc
public final class AccountControllerTests {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void testBasicGet() {
        mockMvc.perform(get("/accounts"))
            .andExpect(status().isOk());
    }
}
```

Static imports make it easier to invoke Builder & Matcher static methods

Define a MockMVC environment (Usage of `@WebMvcTest` would be simpler, which will be shown later)

Perform tests on mockMvc instance

Setting Up Static Imports

- Static imports are key to fluid builders
 - `MockMvcRequestBuilders.*` and `MockMvcResultMatchers.*`
- You can add to Eclipse/STS 'favorite static members' in preferences
 - Java → Editor → Content Assist → Favorites
 - Add to favorite static members
 - `org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get`
 - `org.springframework.test.web.servlet.result.MockMvcResultMatchers.status`

Testing RESTful Controllers

- Argument to `perform()` dictates the action
 - `get()` (or `put()`, `post()`, etc.) from **MockMvcRequestBuilders**
 - Append with methods from **MockHttpServletRequestBuilders**

```
@Test
```

```
public void testRestfulGet() throws Exception {
```

```
    mockMvc.perform(
```

```
        get("/accounts/{acctId}", "123456001")
```

```
        .accept(MediaType.APPLICATION_JSON))
```

```
        ...
```

```
        // Continued ...
```

```
    }
```

MockMvcRequestBuilders
methods go inside `perform()`

MockMvcRequestBuilders - Static methods

- Standard HTTP get, put, post, delete operations
 - *fileUpload* also supported
 - Argument usually a URI template string
 - Returns a **MockHttpServletRequestBuilder** instance (for chaining other methods)

```
// Perform a get using URI template style
```

```
mockMvc.perform(get("/accounts/{acctId}", "123456001"))
```

```
// Perform a get using request parameter style
```

```
mockMvc.perform(get("/accounts?myParam={acctId}", "123456001"))
```

MockHttpServletRequestBuilder Examples

- Setting Accept Header

```
mockMvc.perform(get("/accounts/{acctId}", "123456001")
                .accept("application/json") // Request JSON Response
                ...
```

- PUTting JSON payload

```
mockMvc.perform(put("/accounts/{acctId}", "123456001")
                .content("{ ... }")
                .contentType("application/json")
                ...
```

MockHttpServletRequestBuilder - Static Methods

Method	Description
param	Add a request parameter – such as param(“myParam”, 123)
requestAttr	Add an object as a request attribute. Also, sessionAttr does the same for session scoped objects
header	Add a header variable to the request. Also see headers, which adds multiple headers
content	Request body
contentType	Set content type (Mime type) for body of the request
accept	Set the requested type (Mime type) for the expected response
locale	Set the local for making requests

Testing RESTful Controllers

- `perform()` returns `ResultActions` object
 - Can chain `expects` together – fluid syntax
 - `content()` and `jsonPath()` from `MockMvcResultMatchers`

```
@Test
public void testRestfulGet() throws Exception {
    mockMvc.perform(
        get("/accounts/{acctId}", "123456001")
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().contentType("application/json"));
}
```

`MockMvcResultMatchers`
methods go inside `andExpect()`

Printing Debug Information

- Sometimes you want to know what happened
 - `andDo()` performs action on `MvcResult`
 - `print()` sends the `MvcResult` to output stream
 - Or use `andReturn()` to get the `MvcResult` object

```
// Use this to access the print() method
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;

// Other static imports as well
// Use print() method in test to get debug information
mockMvc.perform(get("/accounts/{acctId}", "123456001"))
    .andDo(print())    // Add this line to print debug info to the console
    .andExpect(status().isOk())
    ...
```

Web Slice Testing with **@WebMvcTest**

- Disables full auto-configuration and instead apply only configuration relevant to MVC tests
- Auto-configure Mvc testing framework
 - **MockMvc** bean is auto configured
 - And optionally Spring Security
- Typically **@WebMvcTest** is used in combination with **@MockBean** for mocking its dependencies

@Mock vs. @MockBean for Dependency

- **@Mock**
 - From Mockito framework
 - Use it when Spring context is not needed
- **@MockBean**
 - From Spring Boot Framework
 - Use it when Spring context is needed
 - Creates a new mock bean when it is not present in the Spring context or replaces a bean with a mock bean when it is present