Spring Framework 5

Feature Highlights & Hidden Gems

Juergen Hoeller
Spring Framework Lead
Pivotal

Spring Framework 5.0

- **5.0 GA as of September 28th, 2017 – one week after JDK 9 GA!**
- **Embracing JDK 9 as well as Kotlin and Project Reactor**
- **Driven by functional API design and reactive architectures**

- **Major baseline upgrade: Java SE 8+, Java EE 7+**
  - JDK 8, Servlet 3.1, Bean Validation 1.1, JPA 2.1, JMS 2.0
  - support for JUnit 5 (next to JUnit 4.12)

- **Comprehensive integration with Java EE 8 API level**
  - Servlet 4.0, Bean Validation 2.0, JPA 2.2, JSON Binding API 1.0
  - e.g. Tomcat 9.0, Hibernate Validator 6.0, Apache Johnzon 1.1

spring by Pivotal

Component Style

Reactive Architectures

Hidden Gems

spring by Pivotal

The State of the Art: Component Classes

```
@Service
@Lazy
public class MyBookAdminService implements BookAdminService {

    // @Autowired
    public MyBookAdminService(AccountRepository repo) {
        ...
    }

    @Transactional
    public BookUpdate updateBook(Addendum addendum) {
        ...
    }
}
```

Composable Annotations

```
@Service
@Scope("session")
@Primary
@Transactional(rollbackFor=Exception.class)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyService {}


@MyService
public class MyBookAdminService {
    ...
}
```

The State of the Art: Configuration Classes

```
@Configuration
@Profile("standalone")
@EnableTransactionManagement
public class MyBookAdminConfig {

    @Bean
    public BookAdminService myBookAdminService() {
        MyBookAdminService service = new MyBookAdminService();
        service.setDataSource(bookAdminDataSource());
        return service;
    }

    @Bean
    public DataSource bookAdminDataSource() {
        ...
    }
}
```

Annotated MVC Controllers

```
@RestController
@CrossOrigin
public class MyRestController {

    @GetMapping("/books/{id}")
    public Book findBook(@PathVariable long id) {
        return this.bookAdminService.findBook(id);
    }

    @PostMapping("/books/new")
    public void newBook(@Valid Book book) {
        this.bookAdminService.storeBook(book);
    }
}
```

Functional Style vs Annotation Style

- **Spring 5 continues a well-established annotation story**
  - loosely coupled components, self-descriptive endpoints
- **Spring 5 provides functional-style APIs as an alternative**
  - programmatic bean registration and endpoint composition
  - no need for annotations or scanning, even avoiding reflection
  - non-null API design with explicit @Nullable declarations
- **First-class support for the Kotlin language out of the box**
  - foundation: Java 8 functional-style APIs + explicit nullability
  - Spring 5's Kotlin extensions make code even more concise

spring by Pivotal

Programmatic Bean Registration with Java 8

```
// Starting point may also be AnnotationConfigApplicationContext
GenericApplicationContext ctx = new GenericApplicationContext();
ctx.registerBean(Foo.class);
ctx.registerBean(Bar.class,
        () -> new Bar(ctx.getBean(Foo.class)));

// Or alternatively with some bean definition customizing
GenericApplicationContext ctx = new GenericApplicationContext();
ctx.registerBean(Foo.class, Foo::new);
ctx.registerBean(Bar.class,
        () -> new Bar(ctx.getBean(Foo.class)),
        bd -> bd.setLazyInit(true));
```

Programmatic Bean Registration with Kotlin

```
// Java-style usage of Spring's Kotlin extensions
val ctx = GenericApplicationContext()
ctx.registerBean(Foo::class)
ctx.registerBean { Bar(it.getBean(Foo::class)) }


// Gradle-style usage of Spring's Kotlin extensions
val ctx = GenericApplicationContext {
    registerBean<Foo>()
    registerBean { Bar(it.getBean<Foo>()) }
}
```

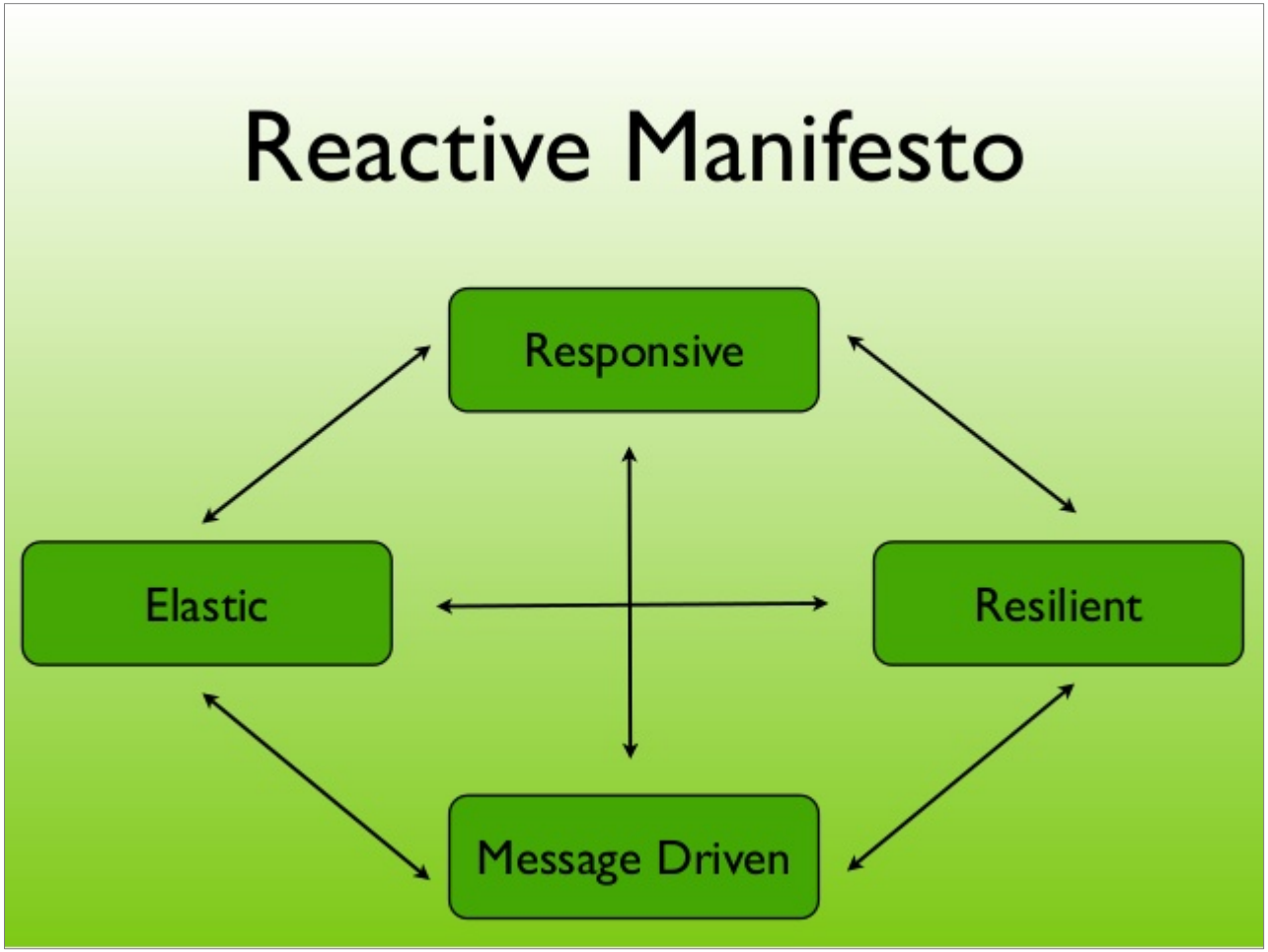
Component Style

Reactive Architectures

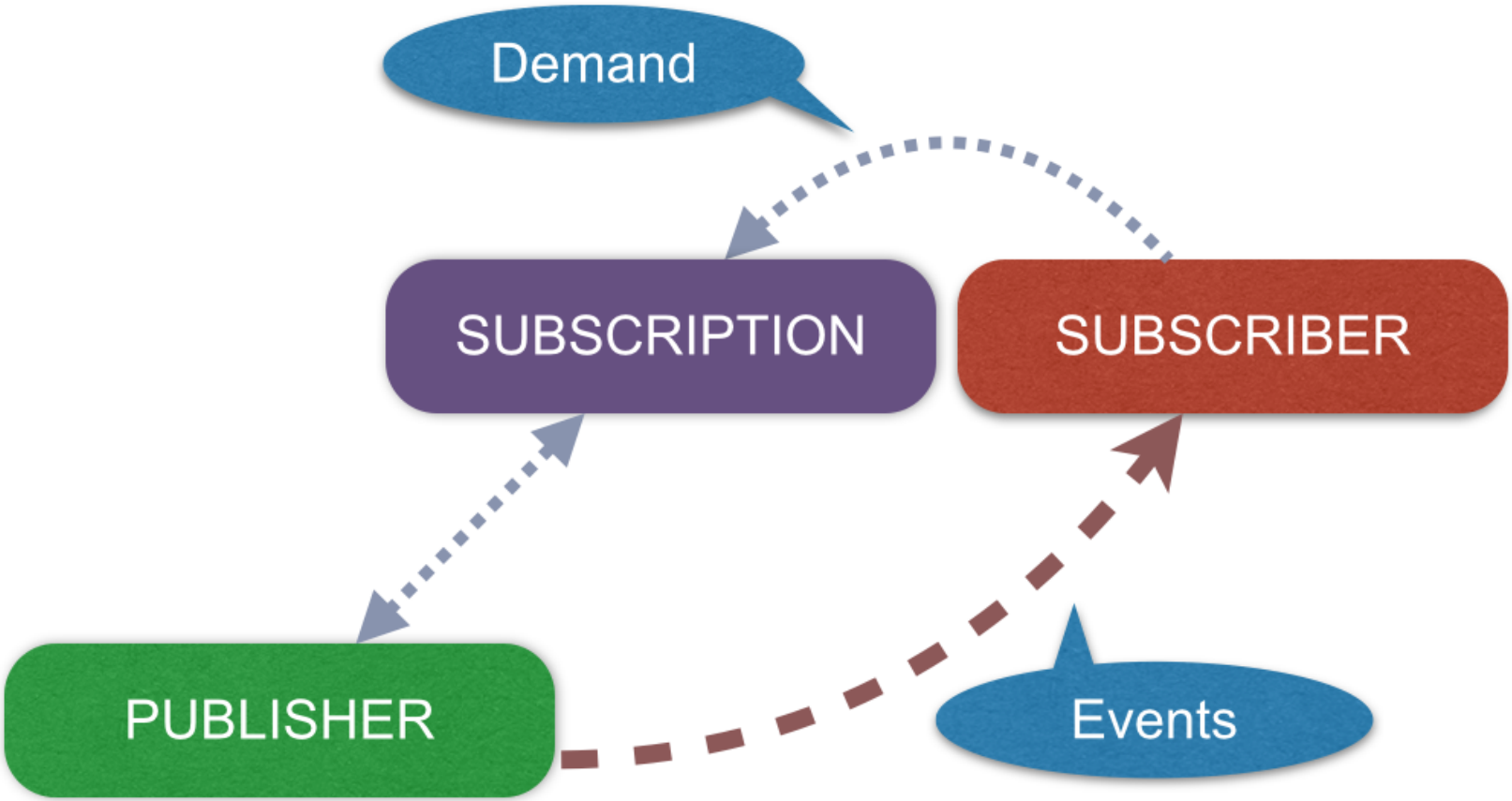
Hidden Gems

spring by Pivotal

The Importance of Reactive Architectures

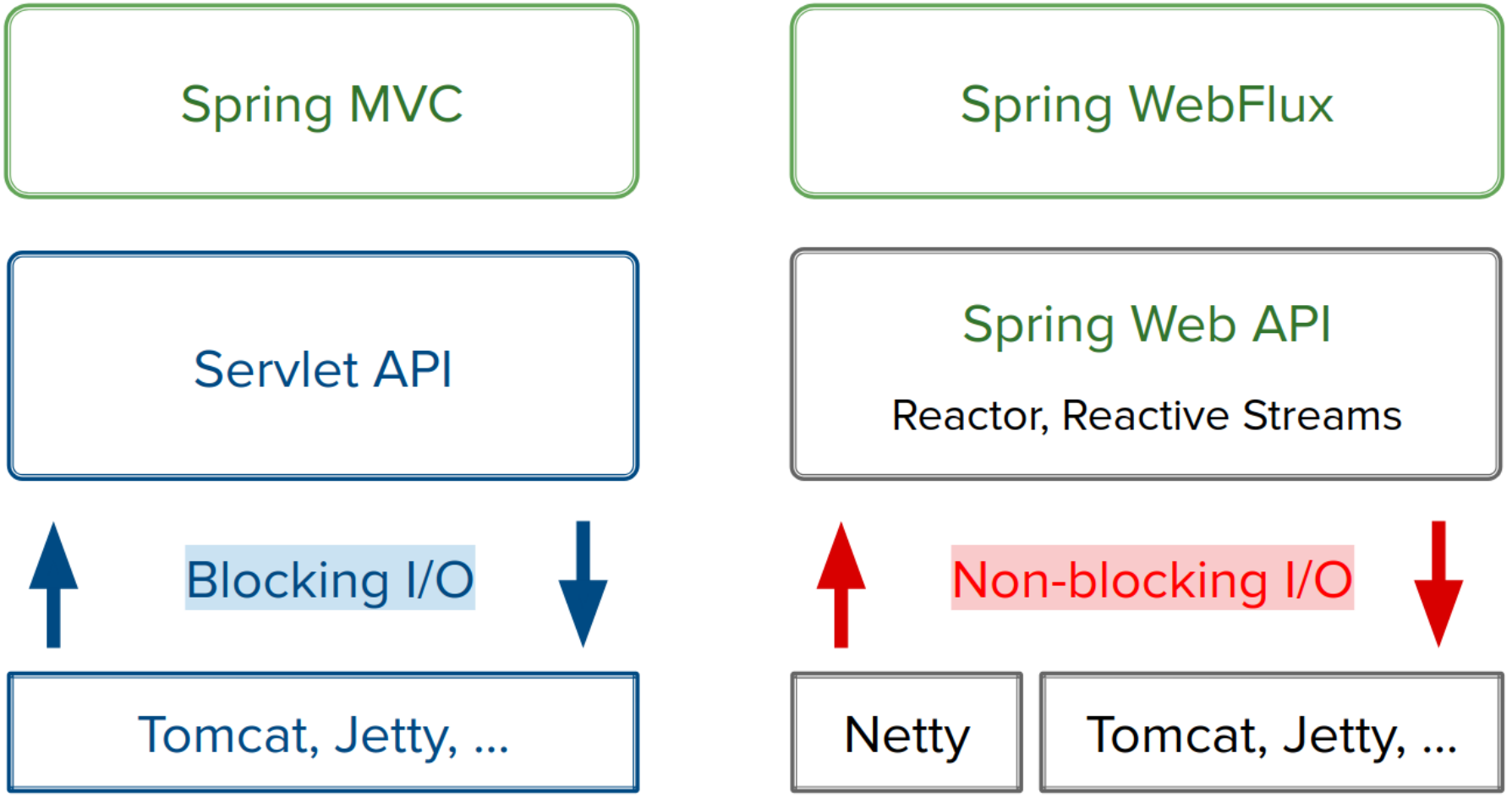


Reactor 3: Reactive Streams with Backpressure

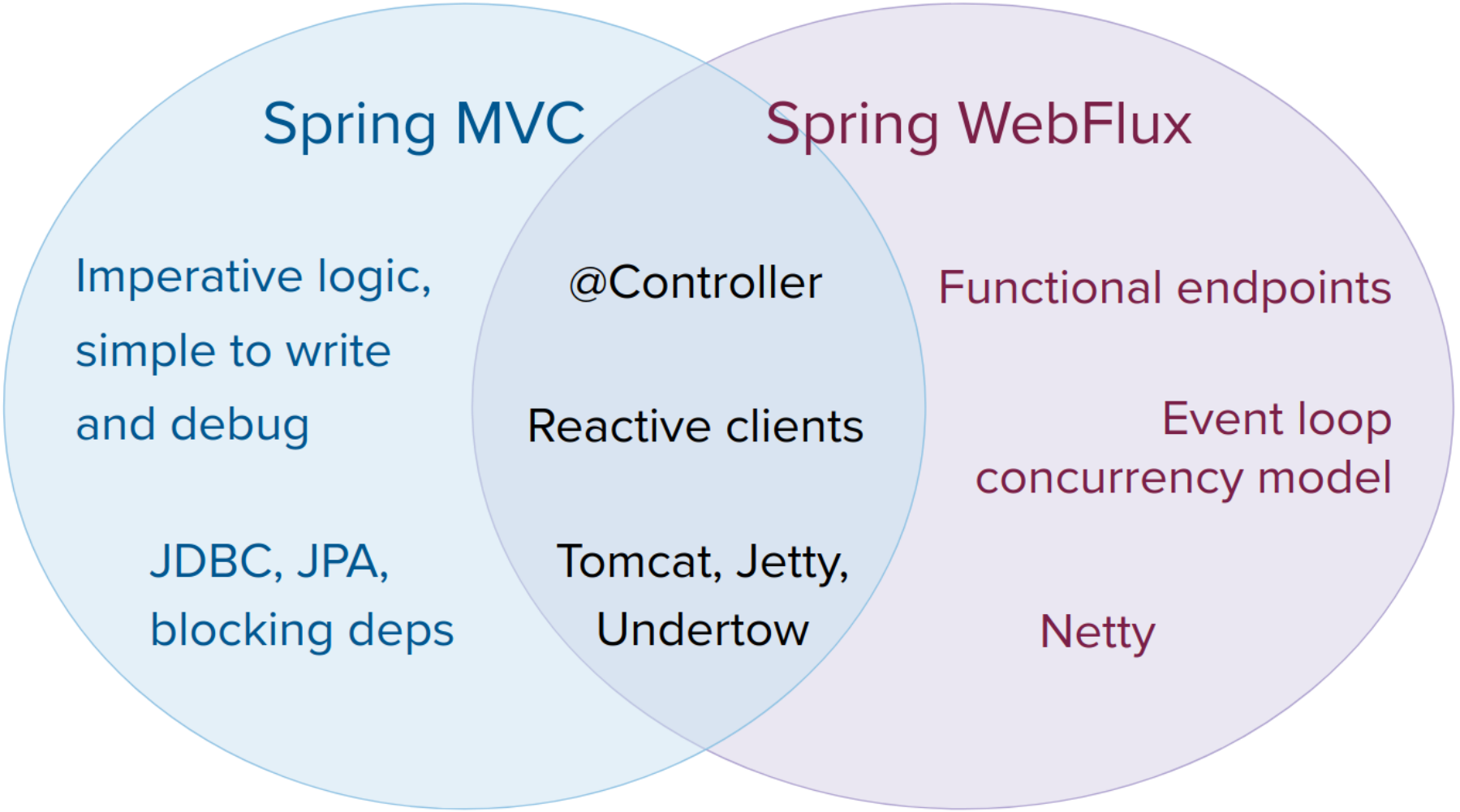


spring by Pivotal

Spring MVC on Servlets ↔ Spring WebFlux on Reactor



Spring MVC vs Spring WebFlux: Programming Model



Reactive Web Controller with Repository Interop

```
@Controller
public class MyReactiveWebController {

    private final UserRepository repository;

    public MyReactiveWebController(UserRepository repository) {
        this.repository = repository;
    }

    @GetMapping("/users/{id}")
    public Mono<User> getUser(@PathVariable Long id) {
        return this.repository.findById(id);
    }

    @GetMapping("/users")
    public Flux<User> getUsers() {
        return this.repository.findAll();
    }
}
```

Functional Web Endpoints with Method References

```
RouterFunction<?> router =
    route(GET("/users/{id}"), handlerDelegate::getUser)
    .andRoute(GET("/users"), handlerDelegate::getUsers);


public class MyReactiveHandlerDelegate {
    ...

    public Mono<ServerResponse> getUser(ServerRequest request) {
        Mono<User> user = Mono.justOrEmpty(request.pathVariable("id"))
            .map(Long::valueOf).then(this.repository::findById);
        return ServerResponse.ok().body(user, User.class);
    }

    public Mono<ServerResponse> getUsers(ServerRequest request) {
        Flux<User> users = this.repository.findAll();
        return ServerResponse.ok().body(users, User.class);
    }
}
```

Functional Web Endpoints in Lambda Style

```
UserRepository repository = ...;

RouterFunction<?> router =
    route(GET("/users/{id}"),
        request -> {
            Mono<User> user = Mono.justOrEmpty(request.pathVariable("id"))
                .map(Long::valueOf).then(repository::findById);
            return ServerResponse.ok().body(user, User.class);
        })
    .andRoute(GET("/users"),
        request -> {
            Flux<User> users = repository.findAll();
            return ServerResponse.ok().body(users, User.class);
        });
```

Reactive Web Client

```
// A functional HTTP client as an alternative to RestTemplate.
// Part of WebFlux but usable anywhere, even in MVC endpoints!

WebClient client = WebClient.create("http://example.org");

Mono<User> result = client.get()
        .uri("/users/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve().bodyToMono(User.class);

Flux<User> result = client.get()
        .uri("/users").accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve().bodyToFlux(User.class);
```

Component Style

Reactive Architectures

Hidden Gems

Nullability

- **Comprehensive nullability declarations across the codebase**
  - non-null method parameters, method return values, fields by default
  - individual `@Nullable` declarations for actually nullable return values etc
- **The Java effect: nullability validation in IntelliJ IDEA and Eclipse**
  - e.g. for “Constant conditions & exceptions” inspection in Intellij IDEA
  - allowing us to catch subtle bugs/gaps in the framework's own codebase
  - allowing applications to validate their own interaction with Spring APIs
- **The Kotlin effect: straightforward assignments to non-null variables**
  - Kotlin compiler only allows such assignments for APIs with clear nullability

Programmatic Lookup via ObjectProvider

- `@Autowired ObjectProvider<MyBean> myBeanProvider`

- **ObjectProvider methods with nullability declarations**
  - `@Nullable T getIfAvailable()`
  - `@Nullable T getIfUnique()`

- **Overloaded variants with java.util.function callbacks**
  - `T getIfAvailable(Supplier<T> defaultSupplier)`
  - `void ifAvailable(Consumer<T> dependencyConsumer)`
  - `T getIfUnique(Supplier<T> defaultSupplier)`
  - `void ifUnique(Consumer<T> dependencyConsumer)`

Data Class Binding

- **Spring data binding can work with immutable classes now**
  - traditional setter methods versus named constructor arguments
- **Property names matched against constructor parameter names**
  - explicit property names via `@ConstructorProperties`
  - or simply inferred from class bytecode (`-parameters` or `-debug`)
- **Perfect match: Kotlin data classes and Lombok data classes**
  - `data class Person(val name: String, val age: Int)`
  - ```
    @Data public class Person {
      private final String name;
      private final int age;
    }
    ```


spring by Pivotal
    ```

Asynchronous Execution

- **ListenableFuture interface with direct CompletableFuture support**
  - `ListenableFuture.completable()` returns `CompletableFuture`
  - e.g. for futures returned from `AsyncListenableTaskExecutor`
- **TaskScheduler interface with alternative to Date and long arguments**
  - `scheduleAtFixedRate(Runnable, Instant, Duration)`
  - `scheduleWithFixedDelay(Runnable, Instant, Duration)`
- **ScheduledTaskHolder interface for monitoring the current tasks**
  - `ScheduledAnnotationBeanPostProcessor.getScheduledTasks()`
  - `ScheduledTaskRegistrar.getScheduledTasks()`

spring by Pivotal

Refined Resource Interaction

- **Overhaul of Spring's Resource abstraction in core.io**
  - `Resource.isFile()`
  - `DefaultResourceLoader` returns writable `FileUrlResource` for “file:”
- **Exposing NIO.2 API at the application level**
  - `Resource.readableChannel()`
  - `WritableResource.writableChannel()`
- **Internally using NIO.2 file access wherever possible**
  - `FileSystemResource.getInput/OutputStream()`
  - `FileCopyUtils.copy(File, File)`

spring by Pivotal

Build-Time Components Indexer

- **Classpath scanning on startup may be slow**
  - `<context:component-scan>` or `@ComponentScan`
  - file system traversal of all packages within the specified base packages
- **The common solution: narrow your base packages**
  - Spring only searches within the specified roots in the classpath
  - alternatively: fully enumerate your component classes (no scanning at all)
- **New variant in 5.0: a build-time annotation processor**
  - spring-context-indexer generates `META-INF/spring.components` per jar
  - automatically used at runtime for compatible component-scan declarations

spring by Pivotal

Commons Logging Bridge

- **“spring-jcl” as a custom Commons Logging bridge**
  - separate module within the Spring Framework umbrella
  - the only required dependency of the spring-core module

- **First-class support for Log4j 2, SLF4J, and JUL (java.util.logging)**
  - Spring-style runtime detection of Log4J 2 and SLF4J providers
  - strong integration with java.util.logging (location-aware LogRecord etc)

- **No custom bridges and no custom excludes in application POMs!**
  - streamlined setup of Log4j or Logback without extra bridge artifacts
  - no exclude of standard “commons-logging” for “jcl-over-slf4j” anymore

Spring Framework 5.0

Q3 2017

annotation-based components
functional style with Java & Kotlin
reactive web stack on Reactor

Spring Framework 5.1

Q3 2018

JDK 11, Reactor 3.2, etc