Spring Framework & Beyond

Part I: Core

Spring Core IoC

- ApplicationContext and BeanFactory
- Bean lifecycle and scopes
- Component scanning and stereotypes
- Java-based and annotation configuration
- Profiles and conditional beans
- Property sources and Environment

Dependency Injection

- Constructor, setter, and field injection
- Qualifier and Primary
- Circular dependency resolution
- Provider and ObjectFactory
- Custom scopes
- DI without Spring (manual wiring)

Spring AOP

- Aspect, advice, and pointcut model
- Before, after, and around advice
- Annotation-driven aspects
- Proxy mechanics (JDK vs. CGLIB)
- Custom annotations with AOP

- Performance and transaction aspects

Part II: Web & REST

Spring Boot REST

- @RestController and request mapping
- Request and response body handling
- Validation with Bean Validation
- Exception handling (@ExceptionHandler, @ControllerAdvice)
- Content negotiation (JSON, XML)
- HATEOAS and hypermedia

Web Applications

- Spring MVC architecture
- Servlet filters and interceptors
- Session and cookie management
- WebSocket and SSE
- Security integration
- Error pages and global handlers

Serialization Patterns

- Jackson configuration and customization
- Custom serializers and deserializers
- JSON views and mixins
- Date/time serialization
- Polymorphic type handling
- Performance tuning

Part III: Data & Persistence

Spring Data Hibernate

- Repository pattern and query methods
- JPA entity mapping and relationships
- Fetch strategies (LAZY vs. EAGER)
- N+1 problem and solutions
- Specifications and Criteria API
- Auditing and soft deletes

Data JDBC

- JdbcTemplate and NamedParameterJdbcTemplate
- RowMapper and ResultSetExtractor
- Batch operations
- Connection pooling (HikariCP)
- Spring Data JDBC vs. JPA
- Stored procedures and functions

Transaction Management

- @Transactional and propagation levels
- Isolation levels and read phenomena
- Programmatic vs. declarative transactions
- Rollback rules and exceptions
- Distributed transactions and Saga pattern
- Testing transactional code

Part IV: Messaging & Events

Messaging RabbitMQ Kafka

- RabbitMQ exchanges, queues, and bindings
- Spring AMQP listener containers
- Kafka producers and consumers
- Kafka Streams and stream processing
- Dead letter queues and retry
- Idempotent consumers

Messaging Events

- ApplicationEvent and EventListener
- Asynchronous event handling
- Domain events and event publishing
- Event sourcing fundamentals
- CQRS pattern
- Spring Integration channels

Part V: Batch & Workflow

Spring Batch

- Job, step, and execution model
- Chunk-oriented processing (reader, processor, writer)
- Tasklet steps
- Parallel and partitioned steps
- Retry, skip, and restart policies
- Job scheduling and monitoring

State Machines

- Enum-based state machines
- Record-based state transitions
- Pattern matching for state logic
- Spring State Machine framework
- Guards, actions, and transitions
- Persistent and distributed state

Part VI: Cloud & Deployment

Docker Kubernetes

- Dockerfile and multi-stage builds
- Spring Boot Docker layers
- Kubernetes Deployments and Services
- ConfigMaps, Secrets, and volumes
- Health checks (liveness, readiness)
- Helm charts and operators

Serverless Spring Cloud Function

- Function, Supplier, and Consumer beans
- AWS Lambda with Spring Cloud Function
- Azure Functions and Google Cloud
- Cold start optimization
- GraalVM native images for serverless
- Testing serverless functions

Part VII: Alternative Frameworks

Quarkus GraalVM

- Quarkus dev mode and live reload
- CDI and ArC dependency injection
- GraalVM native image compilation
- Reflection configuration and substitutions
- Reactive with Mutiny
- Startup time and memory comparison

Jakarta EE

- CDI, JAX-RS, and JPA
- Bean Validation and interceptors
- Jakarta REST endpoints
- Servlet and filter patterns
- Application servers vs. embedded
- Migration from Java EE

Plain Java REST

- HTTP server without frameworks
- Routing and handler patterns
- JSON processing with Jackson
- Manual dependency wiring
- Embedded Jetty and Undertow
- When frameworks are overhead

Chapter 1: Spring Core Fundamentals

Chapter 2: Dependency Injection & IoC

Dependency Injection turns object creation and wiring inside-out. Instead of objects creating their dependencies, a container creates and injects them. This inverts control—hence Inversion of Control (IoC)—making code more modular, testable, and maintainable. Spring dominates the Java DI landscape, but understanding the principles helps you recognize patterns in CDI, Guice, and Dagger. This chapter explores how DI containers work, when to use them, and how to leverage Spring effectively.

15.1 Inversion of Control Principles

Hard-wired dependencies make components brittle, untestable, and difficult to evolve. By inverting control so that an external entity (a container or assembler) creates and injects dependencies rather than letting objects construct their own, you achieve loose coupling that enables modular architectures, easier testing, and clearer boundaries. This approach adheres to SOLID principles, particularly the Dependency Inversion Principle, leading to more robust and flexible software architectures.

Without Dependency Injection (Tight Coupling)

When services instantiate their own dependencies using `new`, they become tightly coupled to concrete implementations. This makes unit testing difficult since you cannot substitute mocks, and changing implementations requires modifying consumer code throughout the application.

Dependency Injection Example

This tightly coupled `OrderService` creates its own `MySQLOrderRepository` and `EmailNotificationService` instances. Testing requires a real database and

email server since you cannot inject mocks. Swapping implementations means changing this class directly.

```
public class OrderServiceTightlyCoupled {
  private final MySQLOrderRepository repository; // Direct dep
  private final EmailNotificationService notificationService;

  public OrderServiceTightlyCoupled() {
    this.repository = new MySQLOrderRepository();
    this.notificationService = new EmailNotificationService();
  }
  // ...
}
```

With Dependency Injection (Loose Coupling)

Constructor injection accepts dependencies as parameters rather than creating them internally. The service depends on interfaces like `OrderRepository` rather than `MySQLOrderRepository`, enabling easy substitution of implementations for testing, configuration changes, or different deployment environments.

Dependency Injection Example

This `OrderServiceWithDI` receives its `OrderRepository` and `NotificationService` through constructor parameters. The service depends only on abstractions, allowing callers to inject any implementation—production databases, in-memory fakes, or mock objects for testing.

```
public class OrderServiceWithDI {
  private final OrderRepository repository; // Abstraction
  private final NotificationService notificationService;

  // Dependencies injected via constructor.
  public OrderServiceWithDI(
      OrderRepository repository,
      NotificationService notificationService) {
    this.repository = repository;
    this.notificationService = notificationService;
  }
```

```
  // ...
}
```

Types of Dependency Injection

Constructor injection is preferred for required dependencies—fields are final and objects are immutable. Setter injection suits optional dependencies with defaults. Field injection using reflection is discouraged because it hides dependencies and complicates testing without a container.

Dependency Injection Example

Three injection styles compared: constructor injection passes dependencies as parameters ensuring immutability, setter injection allows optional dependencies to be configured after construction, and field injection uses `@Autowired` on private fields requiring reflection to populate.

```
// Constructor injection: Most robust approach.
public class ConstructorInjectionExample {
  public ConstructorInjectionExample(OrderRepository repo) {}
}

// Setter injection: For optional dependencies.
public class SetterInjectionExample {
  public void setRepository(OrderRepository repo) {}
}

// Field injection: Requires reflection. Discouraged.
public class FieldInjectionExample {
  // @Autowired private OrderRepository repository;
}
```

Manual Dependency Injection

You can perform dependency injection without any framework by manually instantiating objects and passing them to constructors in your bootstrap code. This works well for small applications but becomes tedious as dependency graphs grow complex.

Dependency Injection Example

Manual wiring creates each dependency explicitly and passes it to consumers. Here `PostgreSQLOrderRepository` and `EmailNotificationService` are instantiated first, then injected into `OrderService`. No framework magic—just straightforward object construction and composition.

```
// Manually creating and wiring dependencies.
OrderRepository repository = new PostgreSQLOrderRepository();
var notifService = new EmailNotificationService();
var orderService = new OrderService(repository, notifService);
```

Testing with Dependency Injection

Dependency injection's primary benefit is testability. Unit tests can inject mock or fake implementations that return predictable data, avoid network calls, and verify interactions. No database, email server, or external service needed—tests run fast and deterministically.

Dependency Injection Example

Anonymous inner classes implement the repository and notification interfaces as mocks. The mock repository returns a test order without database access. These mocks inject into `OrderServiceWithDI`, enabling isolated unit tests that verify business logic independently.

```
// Create mock implementations for testing.
OrderRepository mockRepo = new OrderRepository() {
  @Override public void save(Order order) { /* ... */ }
  @Override public Order findById(String id) {
    return new Order("test-id", "Test Order");
  }
};
NotificationService mockNotif = new NotificationService() {
  @Override public void sendOrderConfirmation(Order o) {}
};

// Inject mocks into the service under test.
```

```
var service = new OrderServiceWithDI(mockRepo, mockNotif);
service.placeOrder(new Order("id", "Test"));
```

15.2 Spring Framework

Manually wiring beans becomes tedious and error-prone as applications scale to hundreds of components. Spring's IoC container auto-detects, configures, and proxies beans using declarative annotations or XML, automatically managing object creation, dependencies, and lifecycle. As the de facto standard for enterprise Java development, understanding Spring's patterns is essential for building modular, testable, and scalable services.

Core Container and Beans

Spring's `ApplicationContext` is the IoC container that creates, configures, and manages beans. It reads configuration from `@Configuration` classes, XML, or component scanning, then builds a dependency graph and instantiates beans in the correct order.

Dependency Injection Example

Java-based configuration uses `@Configuration` classes containing `@Bean` methods. Each method returns an object that Spring manages. Method calls within the configuration class are intercepted by Spring proxies, ensuring singleton beans return the same instance.

```
// Java-based config: Modern way to configure Spring beans.
@Configuration
public static class AppConfig {
  @Bean // Produces a bean managed by Spring.
  public OrderRepository orderRepository() {
    return new PostgreSQLOrderRepository();
  }

  @Bean
  public OrderService orderService() {
    // Spring auto-injects dependencies.
```

```
    return new OrderService(
        orderRepository(), notificationService());
  }
}
```

Bean Scopes

Spring beans have scopes controlling their lifecycle. Singleton scope (default) creates one instance per container. Prototype scope creates a new instance each time the bean is requested. Web scopes include request, session, and application for HTTP-aware contexts.

Dependency Injection Example

The `@Scope` annotation controls bean lifecycle. `OrderRepository` with singleton scope shares one instance across the application. `ShoppingCart` with prototype scope creates a fresh instance for each injection point, appropriate for stateful per-user objects.

```
@Configuration
public static class BeanScopeConfig {
  @Bean
  @Scope("singleton") // Default: one instance per container.
  public OrderRepository orderRepository() {
    return new PostgreSQLOrderRepository();
  }

  @Bean
  @Scope("prototype") // New instance per request.
  public ShoppingCart shoppingCart() {
    return new ShoppingCart();
  }
}
```

Bean Lifecycle

Spring provides hooks into bean lifecycle events. `@PostConstruct` methods run after dependency injection completes—useful for initialization logic.

`@PreDestroy` methods run before the container shuts down—useful for cleanup like closing connections or releasing resources.

Dependency Injection Example

`LifecycleBean` uses JSR-250 annotations for lifecycle callbacks. The `@PostConstruct` method executes after Spring injects all dependencies. The `@PreDestroy` method executes during graceful shutdown, allowing cleanup before the bean is destroyed.

```
@Component
public static class LifecycleBean {
  @PostConstruct
  public void init() {
    System.out.println("Bean initialized after DI.");
  }

  @PreDestroy
  public void cleanup() {
    System.out.println("Bean destroyed before container shutdown.");
  }
}
```

Dependency Injection Types in Spring

Spring supports all injection styles. Constructor injection is preferred—`@Autowired` is optional when there's a single constructor. Setter injection suits optional dependencies. Field injection works but hides dependencies and complicates testing without Spring's test support.

Dependency Injection Example

Constructor injection with `@Autowired` makes the repository a required, immutable dependency. Setter injection allows optional configuration after construction. Field injection (commented out) directly populates private fields but is discouraged in modern Spring applications.

```java
@Component
public static class InjectionExamples {
  private final OrderRepository repository; // Preferred

  @Autowired // Optional for single constructor.
  public InjectionExamples(OrderRepository repository) {
    this.repository = repository;
  }

  @Autowired // Setter injection.
  public void setNotificationService(NotificationService svc) {
    this.notificationService = notificationService;
  }

  // @Autowired private PaymentProcessor processor; // Avoid
}
```

Qualifiers and Primary

When multiple beans implement the same interface, Spring cannot autowire without disambiguation. `@Qualifier` specifies which implementation to inject by name. `@Primary` marks one bean as the default choice when no qualifier is specified.

Dependency Injection Example

Two `OrderRepository` beans are defined with different qualifiers. `ServiceUsingQualifier` explicitly requests the postgres-qualified bean via `@Qualifier("postgres")`. Without the qualifier, Spring would fail with an ambiguous bean exception during startup.

```java
@Configuration
public static class QualifierConfig {
  @Bean @Qualifier("mysql")
  public OrderRepository mysqlRepository() {
    return new MySQLOrderRepository();
  }
  @Bean @Qualifier("postgres")
  public OrderRepository postgresRepository() {
    return new PostgreSQLOrderRepository();
  }
```

```
}

@Component
public static class ServiceUsingQualifier {
  @Autowired
  public ServiceUsingQualifier(
      @Qualifier("postgres") OrderRepository repo) {}
}
```

15.3 Spring Boot

Traditional Spring configuration demands verbose XML or Java config just to start a service, and integrating various libraries adds further boilerplate. Spring Boot simplifies this with auto-configuration, starter dependencies, and an opinionated setup that gets you running with minimal configuration. As the standard for building new Spring applications, Boot dramatically increases developer productivity and provides a robust foundation for microservices and cloud-native applications.

Auto-configuration

Spring Boot examines your classpath and automatically configures beans based on what libraries are present. If you include a database driver, Boot configures a `DataSource`. If you include Spring MVC, Boot configures an embedded web server and dispatcher servlet.

Dependency Injection Example

`@SpringBootApplication` combines three annotations: `@Configuration` for Java config, `@EnableAutoConfiguration` for classpath scanning, and `@ComponentScan` for finding beans. This single annotation bootstraps a fully configured application with sensible defaults.

```
// @SpringBootApplication combines @Configuration,
// @EnableAutoConfiguration, and @ComponentScan.
@SpringBootApplication
```

```
public static class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
    // Spring Boot auto-configures based on classpath.
  }
}
```

Starters

Spring Boot starters are curated dependency sets that pull in related libraries together. Adding `spring-boot-starter-web` brings Spring MVC, embedded Tomcat, and Jackson JSON processing. Starters ensure compatible versions and reduce dependency management complexity.

Dependency Injection Example

With `spring-boot-starter-web` on the classpath, this minimal application serves HTTP requests. The `@RestController` handles web requests while Boot auto-configures the embedded server. No XML, no explicit server setup—just annotate and run.

```
// Adding `spring-boot-starter-web` brings in:
// - Spring MVC
// - Embedded Tomcat
// - Jackson for JSON processing
// - And auto-configures them all.
//
// Example: A simple REST controller works out of the box.
@SpringBootApplication
public static class WebApplication {
  @RestController
  public static class HelloController {
    @GetMapping("/hello")
    public String hello() {
      return "Hello, World!";
    }
  }
}
```

Application Properties

Spring Boot externalizes configuration through `application.properties` or `application.yml` files. These configure server ports, database URLs, logging levels, and custom application settings. Properties can be overridden via environment variables or command-line arguments.

Application Properties Example

Properties files configure infrastructure and custom settings. `@ConfigurationProperties` binds prefixed properties to a type-safe POJO. Boot validates the configuration at startup, catching typos and type mismatches before the application runs.

```
// Example `application.properties` content:
// server.port=8080
// spring.datasource.url=jdbc:postgresql://localhost/mydb
// app.name=My Application

// Custom properties bind to POJOs via @ConfigurationProperties.
@ConfigurationProperties(prefix = "app")
public static class AppProperties {
  private String name;
  private String version;
  // ... getters and setters ...
}
```

Profiles

Spring profiles activate different configurations for different environments. Beans annotated with `@Profile("dev")` only load when the dev profile is active. This enables environment-specific datasources, feature flags, and service endpoints without code changes.

Profiles Example

Development configuration uses an in-memory H2 database for fast local iteration. Production configuration uses HikariCP connection pooling with a real

database. Activate profiles via `spring.profiles.active` property or `SPRING_PROFILES_ACTIVE` environment variable.

```
@Configuration
@Profile("dev") // Active only when "dev" profile is active.
public static class DevelopmentConfig {
  @Bean
  public DataSource dataSource() {
    return new H2DataSource(); // In-memory DB for dev.
  }
}

@Configuration
@Profile("prod") // Active only when "prod" profile is active.
public static class ProductionConfig {
  @Bean
  public DataSource dataSource() {
    return new HikariDataSource(); // Real DB for prod.
  }
}
```

15.4 CDI (Contexts and Dependency Injection)

Java EE environments needed a standardized, portable DI model beyond proprietary frameworks. CDI (Contexts and Dependency Injection, JSR 365) defines injection scopes, qualifiers, and contextual lifecycles for portable components, providing a robust, type-safe, and extensible model for managing stateful objects. Even outside EE, CDI concepts influence Jakarta and MicroProfile runtimes, making familiarity with its patterns valuable for enterprise application development.

CDI Beans and Scopes

CDI uses scope annotations to control bean lifecycle within Jakarta EE containers. `@ApplicationScoped` creates a singleton shared across the application. `@RequestScoped` creates a new instance per HTTP request. `@SessionScoped` ties instances to user sessions.

CDI Beans Example

`@Inject` performs dependency injection in CDI, similar to Spring's `@Autowired`. The `OrderService` is application-scoped, so one instance handles all requests. `ShoppingCart` is request-scoped, creating a fresh cart for each HTTP request.

```
// @ApplicationScoped: Singleton within the application.
@ApplicationScoped
public static class OrderService {
  @Inject // Injects the OrderRepository dependency.
  private OrderRepository repository;

  public void placeOrder(Order order) {
    repository.save(order);
  }
}

// @RequestScoped: A new instance is created for each HTTP request.
@RequestScoped
public static class ShoppingCart {}
```

Qualifiers

CDI qualifiers are custom annotations that distinguish between multiple implementations of the same interface. Unlike Spring's string-based qualifiers, CDI qualifiers are type-safe annotations. The container uses qualifier annotations to select the correct implementation at injection points.

CDI Qualifiers Example

Custom `@MySQL` and `@PostgreSQL` annotations are defined as qualifiers. These annotations mark repository implementations and injection points. When injecting `OrderRepository`, adding `@PostgreSQL` tells CDI to select the PostgreSQL implementation specifically.

```
// Custom qualifier annotations.
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
public @interface MySQL {}
```

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD})
public @interface PostgreSQL {}

// Injecting a specific implementation using a qualifier.
@ApplicationScoped
public static class OrderServiceWithQualifier {
  @Inject
  @PostgreSQL // Injects the PostgreSQLOrderRepository.
  private OrderRepository repository;
}
```

Producer Methods

CDI producer methods create beans that cannot be instantiated by the container directly—objects requiring complex initialization, third-party classes, or runtime configuration. Methods annotated with `@Produces` return objects that CDI manages and injects into dependent beans.

Producer Methods Example

The `@Produces` annotation marks `createDataSource()` as a bean producer. CDI calls this method to create the `DataSource` bean, which can then be injected anywhere. The `@ApplicationScoped` annotation ensures only one `DataSource` instance exists.

```
@ApplicationScoped
public static class DataSourceProducer {
  @Produces // Marks this method as a producer of a DataSource bean.
  @ApplicationScoped
  public DataSource createDataSource() {
    HikariDataSource ds = new HikariDataSource();
    ds.setJdbcUrl("jdbc:postgresql://localhost/mydb");
    return ds;
  }
}
```

15.5 Guice Patterns

Large Java SE services often need powerful DI without the heft of a full Spring context. Google Guice uses Java code modules to configure type-safe bindings, scopes, and providers, emphasizing explicit configuration and avoiding XML. With its smaller footprint, faster startup, and strong compile-time guarantees, Guice offers a compelling alternative that excels in cloud-native or command-line applications.

Guice Module

Guice modules configure bindings in plain Java code. The `configure()` method declares which implementation satisfies each interface. `@Provides` methods work like Spring's `@Bean`, creating objects with custom initialization logic. Scopes like `Singleton` control instance lifecycle.

Guice Module Example

`OrderModule` binds `OrderRepository` to its PostgreSQL implementation. `NotificationService` binding includes `.in(Singleton.class)` for singleton scope. The `@Provides` method creates a configured `DataSource` with custom JDBC URL settings.

```
public static class OrderModule extends AbstractModule {
  @Override
  protected void configure() {
    // Binds OrderRepository to its PostgreSQL impl.
    bind(OrderRepository.class).to(PostgreSQLOrderRepository.class);
    // Binds NotificationService as singleton.
    bind(NotificationService.class)
        .to(EmailNotificationService.class).in(Singleton.class);
  }

  @Provides // Provider method, similar to Spring's @Bean.
  @Singleton
  public DataSource provideDataSource() {
    HikariDataSource ds = new HikariDataSource();
    ds.setJdbcUrl("jdbc:postgresql://localhost/mydb");
    return ds;
```

```
    }
}
```

Using Guice

Application bootstrap creates an `Injector` from one or more modules. The injector builds the complete dependency graph and provides fully-wired instances on demand. Unlike Spring, Guice has no component scanning—all bindings must be explicitly declared in modules.

Using Guice Example

`Guice.createInjector()` builds the dependency graph from the module configuration. Calling `getInstance()` returns a fully-constructed `OrderService` with all dependencies injected. Guice validates the graph at injector creation, failing fast on missing bindings.

```
// Create the Guice injector from the module.
Injector injector = Guice.createInjector(new OrderModule());

// Get instance; Guice auto-injects dependencies.
var orderService = injector.getInstance(OrderService.class);
orderService.placeOrder(new Order("123", "Widget"));
```

15.6 Dagger for Compile-time DI

Reflection-based DI containers add startup cost and may not catch configuration errors until runtime. Dagger addresses this by generating plain Java code for dependency graphs at compile time, eliminating reflection overhead and validating the entire wiring before the application runs. This makes Dagger ideal for performance-critical environments like Android and serverless workloads where cold-start latency and runtime efficiency are paramount.

Dagger Component

Dagger components are interfaces that define the dependency graph's entry points. At compile time, Dagger generates an implementation that constructs all dependencies. The `@Component` annotation lists which modules provide bindings for this graph.

Dagger Component Example

`ApplicationComponent` declares that `OrderService` can be obtained from this graph. Dagger's annotation processor generates `DaggerApplicationComponent` with all wiring code. Compile-time generation means no reflection overhead and immediate detection of missing dependencies.

```
// ApplicationComponent defines what Dagger can provide.
@Component(modules = OrderModule.class)
@Singleton
public interface ApplicationComponent {
  OrderService orderService(); // Dagger provides this.
}
```

Dagger Module

Dagger modules contain `@Provides` methods that create dependencies. Unlike Guice's imperative binding DSL, Dagger modules are declarative—each method produces one type. The annotation processor analyzes these methods to generate efficient factory code at compile time.

Dagger Module Example

Each `@Provides` method returns a specific dependency. `@Singleton` scope ensures one instance per component. Dagger generates code that calls these methods in the correct order, passing results to dependent providers automatically.

```
@Module
public static class OrderModule {
```

```
  @Provides
  @Singleton
  public OrderRepository provideOrderRepository() {
    return new PostgreSQLOrderRepository();
  }

  @Provides
  @Singleton
  public NotificationService provideNotificationService() {
    return new EmailNotificationService();
  }
}
```

Using Dagger

Application code obtains dependencies from the generated component implementation. `DaggerApplicationComponent.create()` instantiates the component with all wiring pre-computed. Method calls on the component return fully-constructed objects with zero runtime reflection overhead.

Using Dagger Example

The generated `DaggerApplicationComponent` class contains all factory logic. Calling `orderService()` returns an `OrderService` with its repository and notification service already injected. This generated code runs as fast as hand-written factory methods.

```
// Dagger generates `DaggerApplicationComponent` at compile time.
var component = DaggerApplicationComponent.create();

// Get the OrderService instance from the component.
OrderService orderService = component.orderService();
orderService.placeOrder(new Order("123", "Widget"));
```