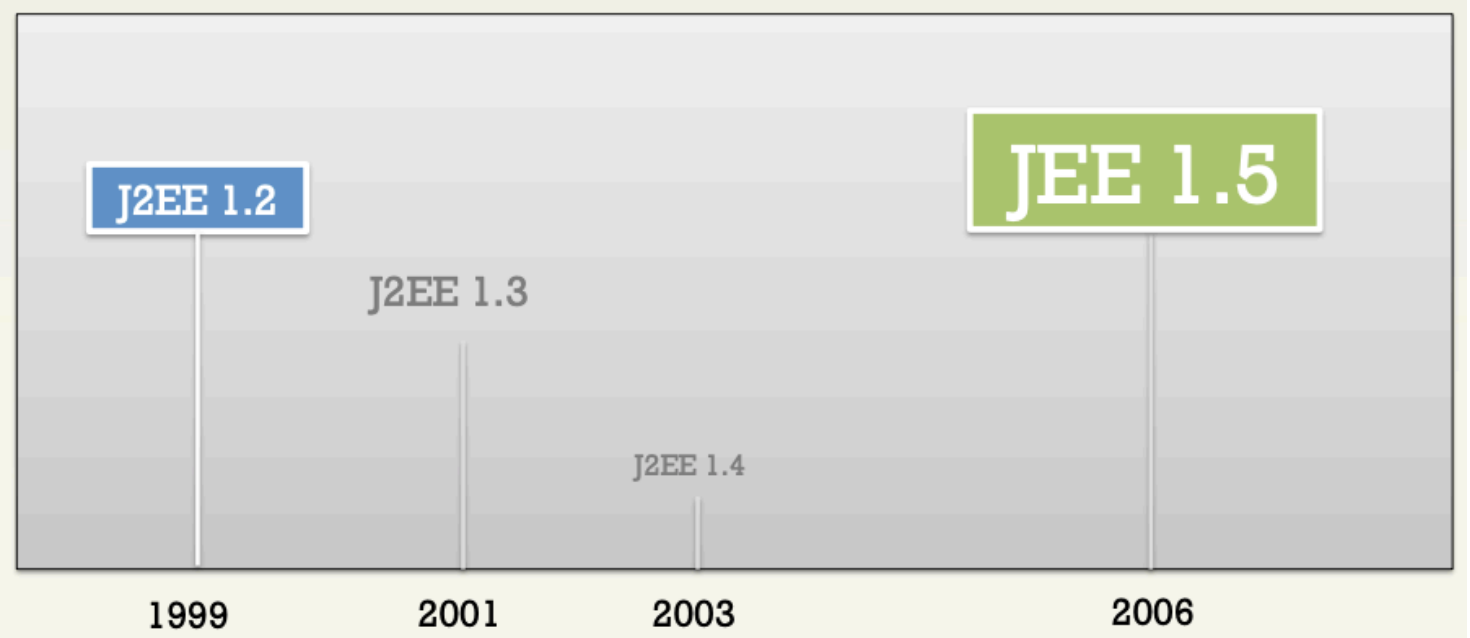
Spring Framework

by Andrés Arcia

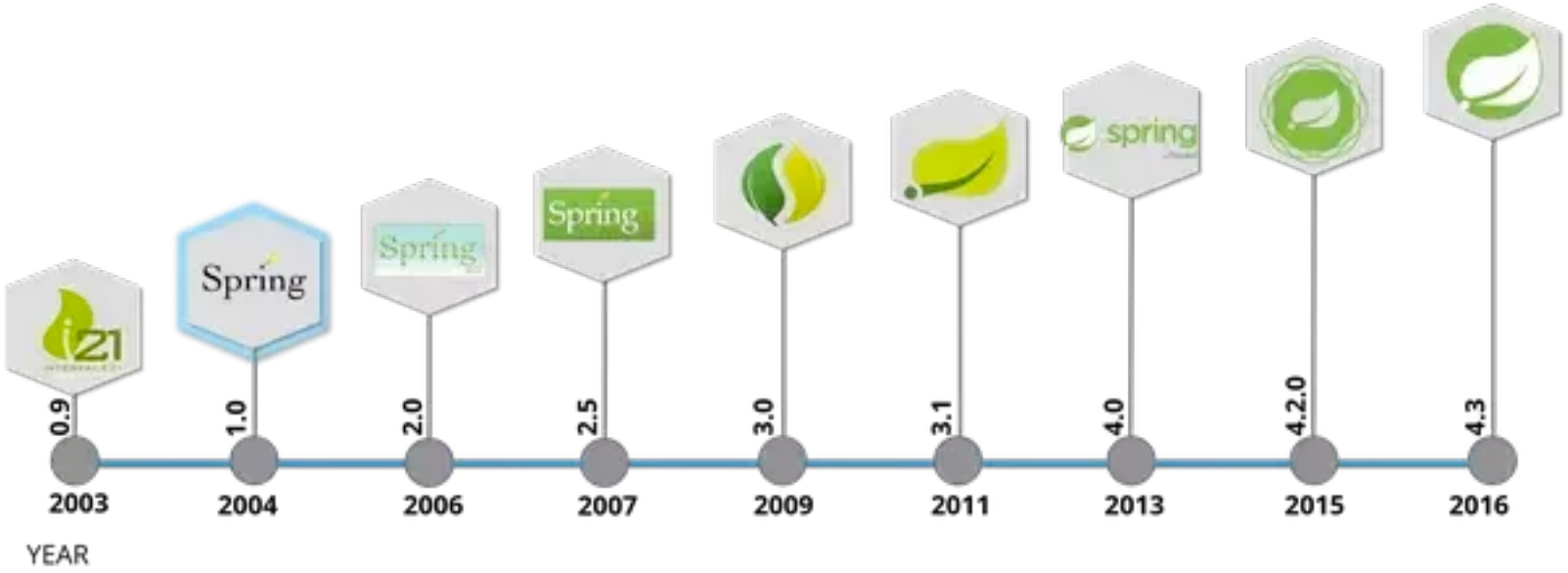
Our history!

- Time consuming
- High complexity



Our history!

- 2002 new book is published with a new approach
- Book comes with a proposal framework
- 2003 framework is born under Apache license



What is the problem?

- Tight coupling (rigid)
- Hard to refactor (fragility)
- Expensive


```
4
5   public class HumanBot {
6
7       private final HelloWorld helloWorld = new HelloWorld();     // Tight coupling
8
9       public String greeting() { return "Hey body, " + helloWorld.sayHello(); }
12
13  }
14
```

An alternative came up

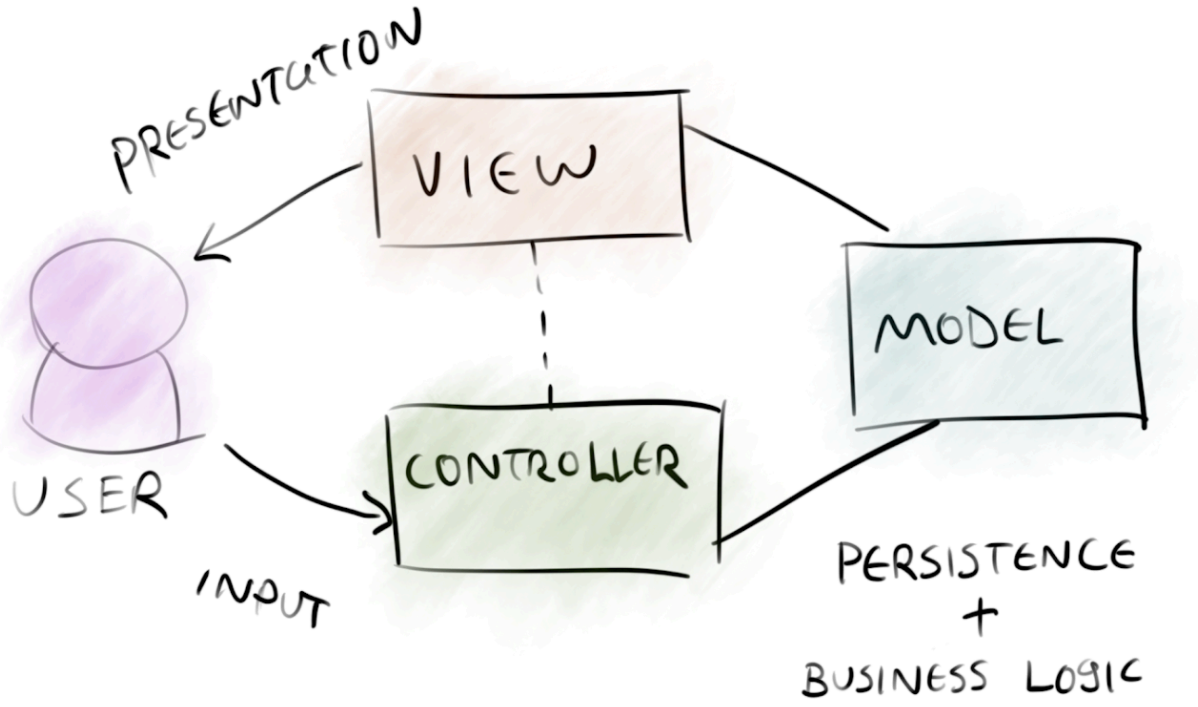
- More flexible
- Easier and less expensive to refactor
- Give the responsibility to other component

```java
public class ReusableBot {

    private final HelloMoon helloMoon;


    public ReusableBot(HelloMoon helloMoon) {
        this.helloMoon = helloMoon;
    }

    public String greeting() { return "Hey, " + helloMoon.sayHello(); }

}
```

```java
        HelloMoon helloMoon = new HelloMoon();

        ReusableBot reusableBot = new ReusableBot(helloMoon);
        System.out.println(reusableBot.greeting());
```

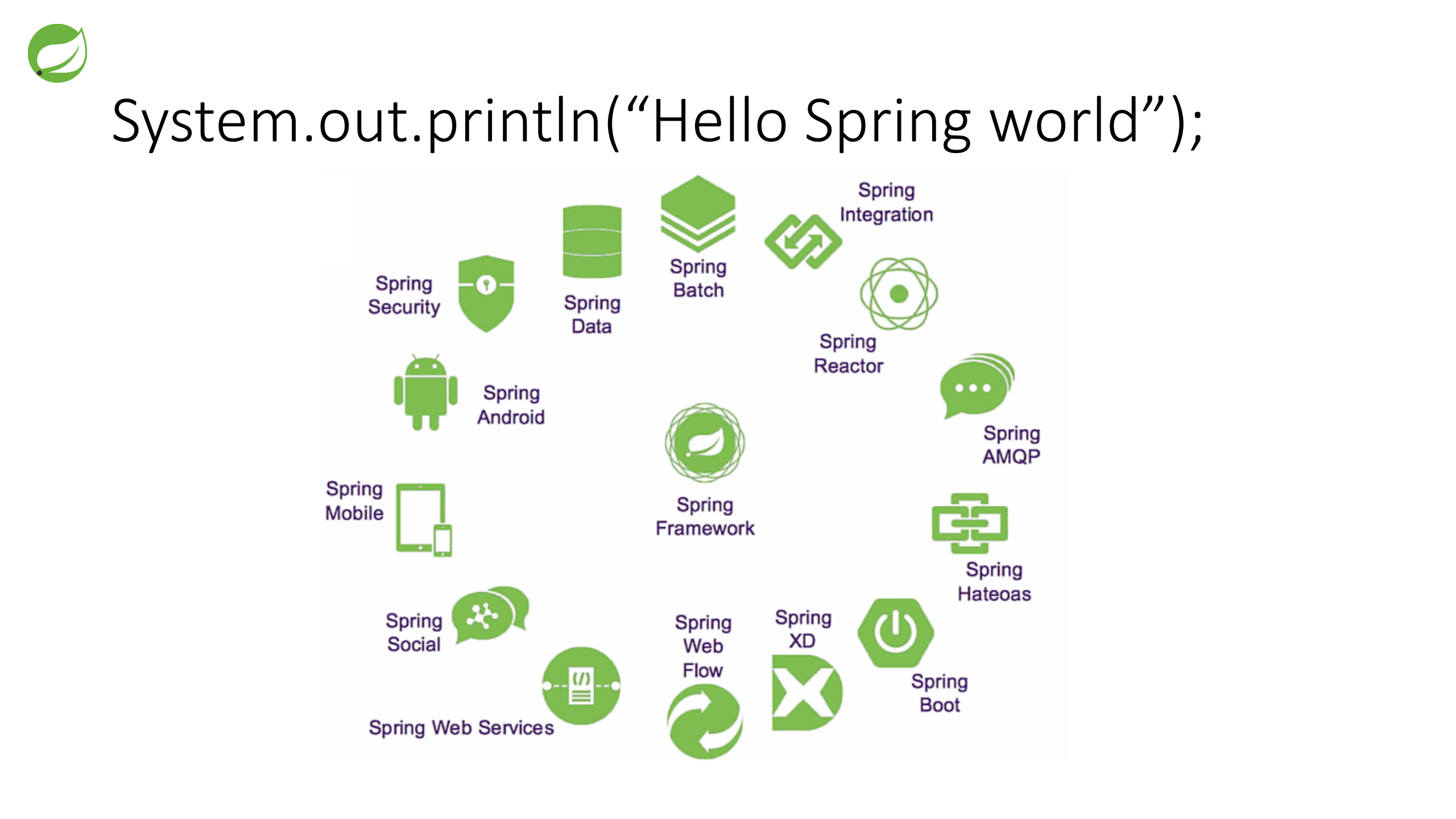
What is Spring Framework?

- Is a Dependency Injection (DI) Framework.
- See some examples (ex. 1-2)



Spring Framework

- Is divided into modules. Apps can choose which modules they need.
- The heart is the core container, configuration model and DI mechanism.
- 2003 framework is born under Apache license



System.out.println("Hello Spring world");

Spring Framework Philosophy

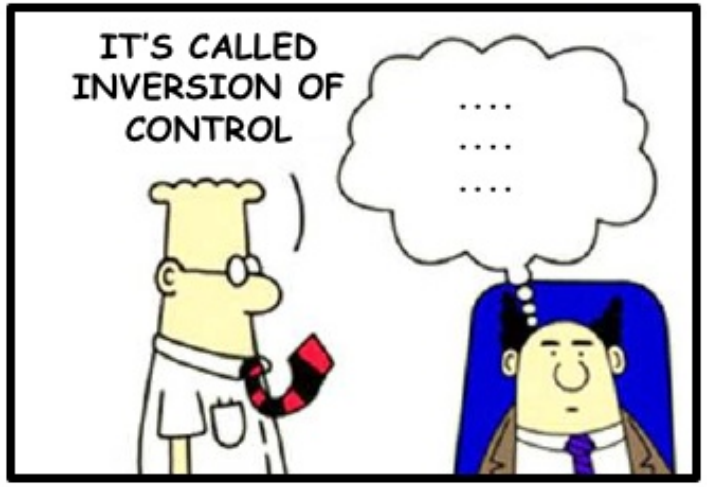
- Provide choice at every level
- Accommodate diverse perspectives
- Backward compatibility
- Care about API design
- Set high standards for code quality

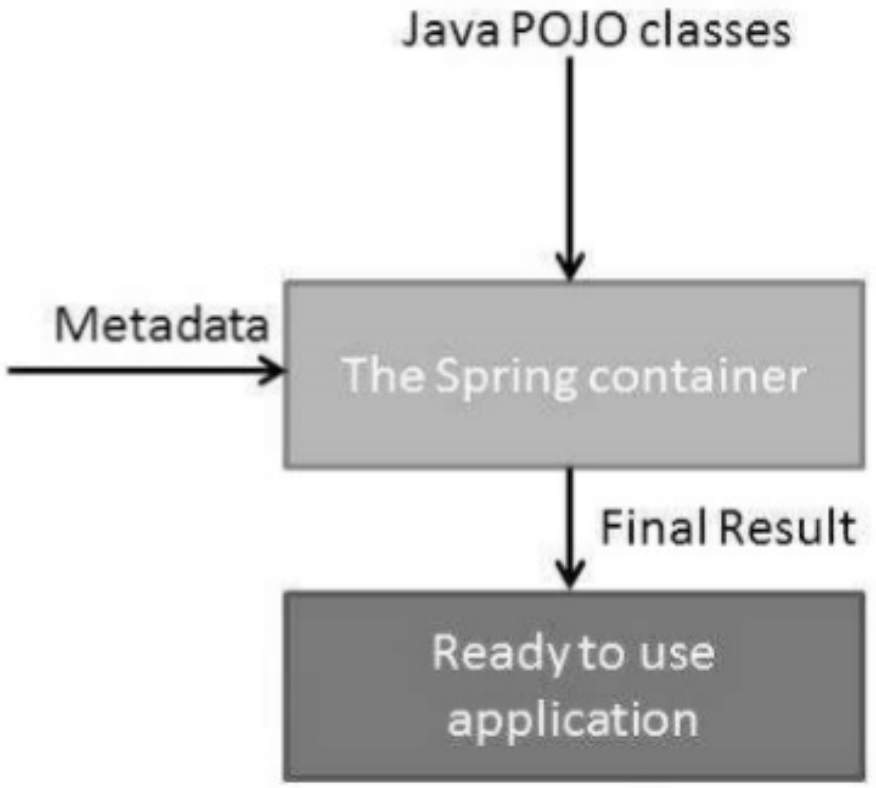
Spring advantages

- Build simple
- Portable
- Less coupling
- Cohesive
- Fast development and testing
- Non invasive (external) code or implementations, only POJO’s
- Any part can easily be “swapped out”
- Works from EE to a Standalone

What is IoC principle?

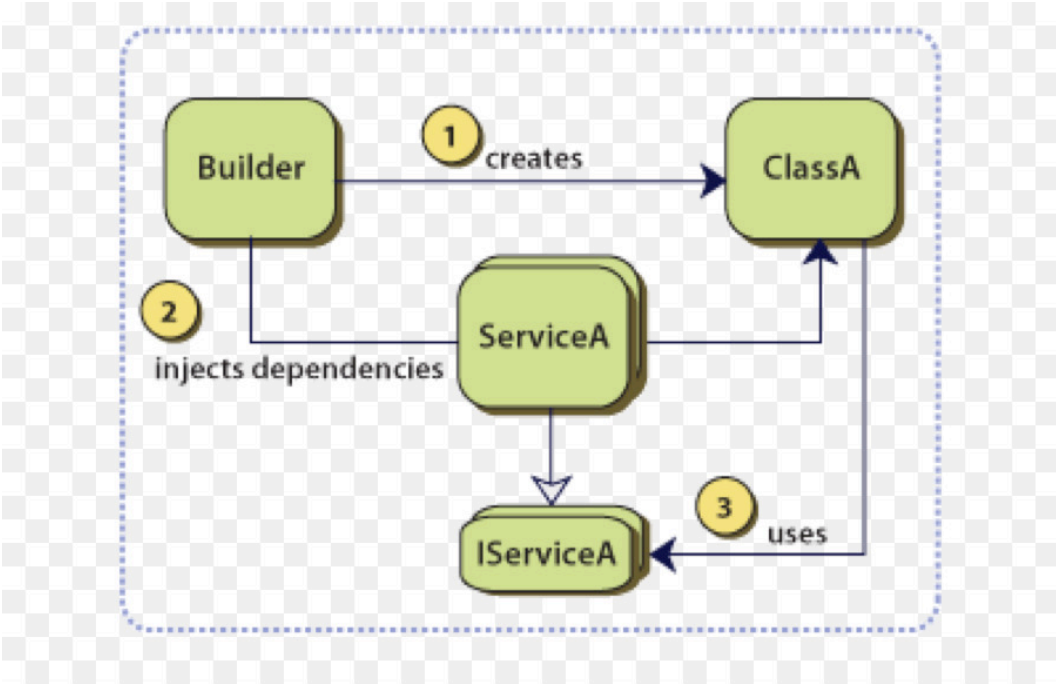
- IoC defines that all dependencies, life cycle events and configuration will reside out of the components.
- Based on “Hollywood” principle “Don’t call us, we’ll call you”.





What is DI?

- Is a characteristic of an IoC, it creates and associate dependencies between components.
- See some examples (ex. 3)



Why DI?

- Simplifies your code.
- Frees it from burden of resolving its dependencies.
- Promotes programming to interfaces.
- Improves testability
- Allows for centralized control over object lifecycle

What does Spring do?

- Spring manages the lifecycle of the application; all resources are fully initialized before use.
- Spring always creates the resources in the right order based on their dependencies.
- Spring resolves all dependencies by its own.

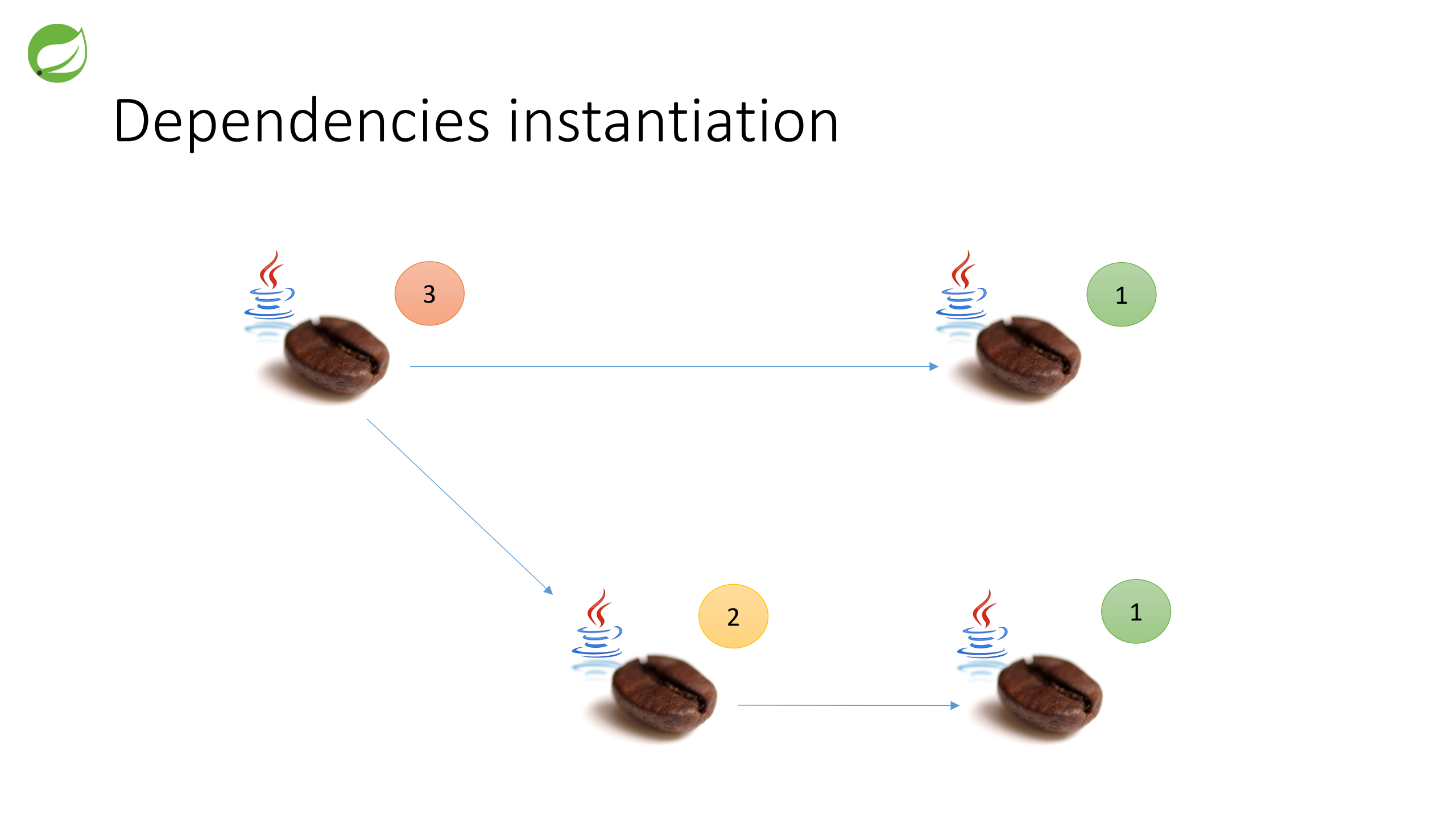
How Spring Works?

Configuration Instructions

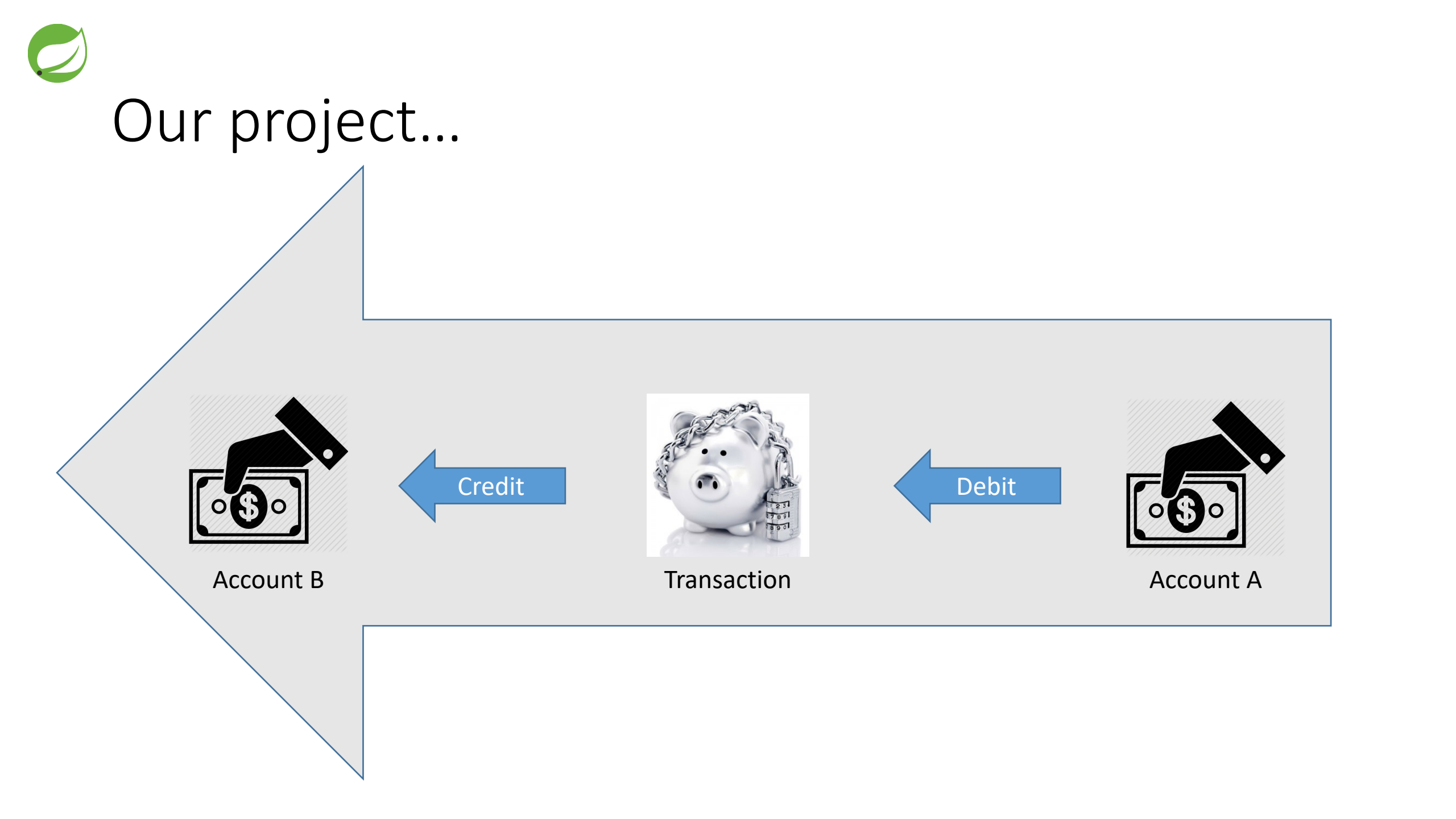
Spring Application Context

Fully Configured Application System (Ready for use)

POJO's



Dependencies instantiation


Our project…
Account B
Credit
Transaction
Debit
Account A

Our project...

Account

+id: String
+userId: String
+balance: Double

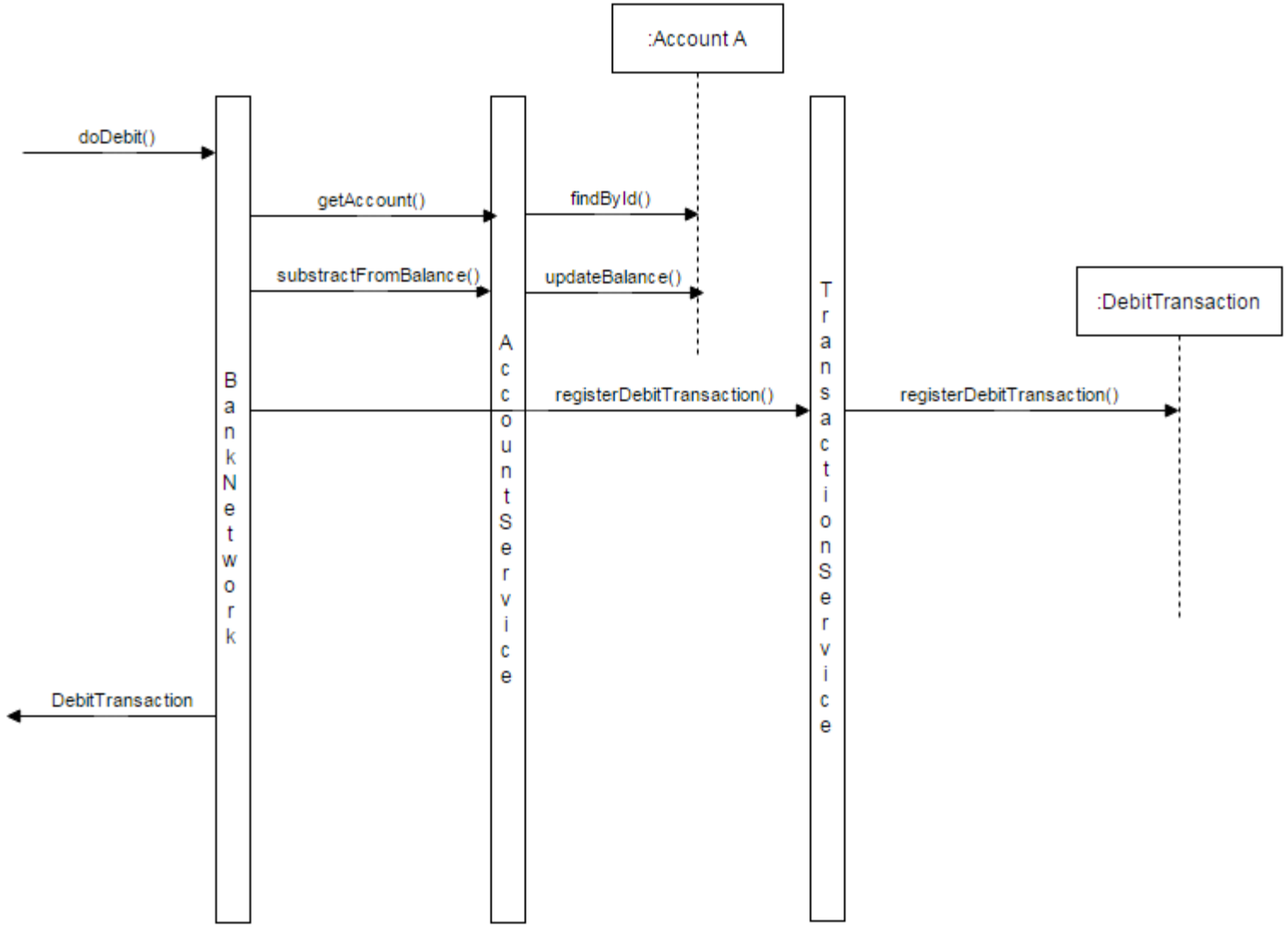
Transaction

+id: String
+originAccountId: String
+destinyAccountId: String
+amount: Double
+type: Transaction.Type
+confirmation: Transaction.Confirmation
+timespan: Date

Our project…

Debit

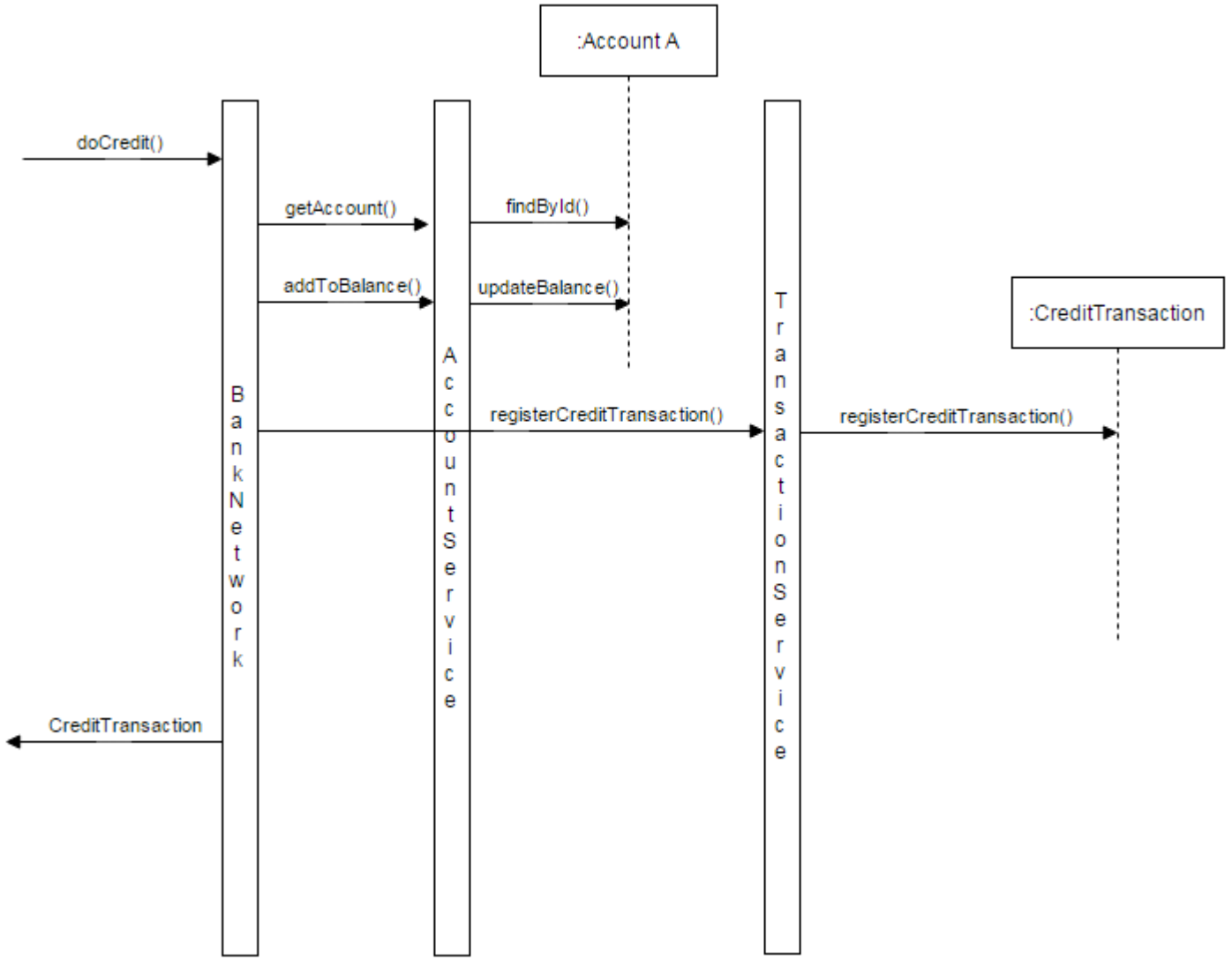
- The “Debit”



Our project…

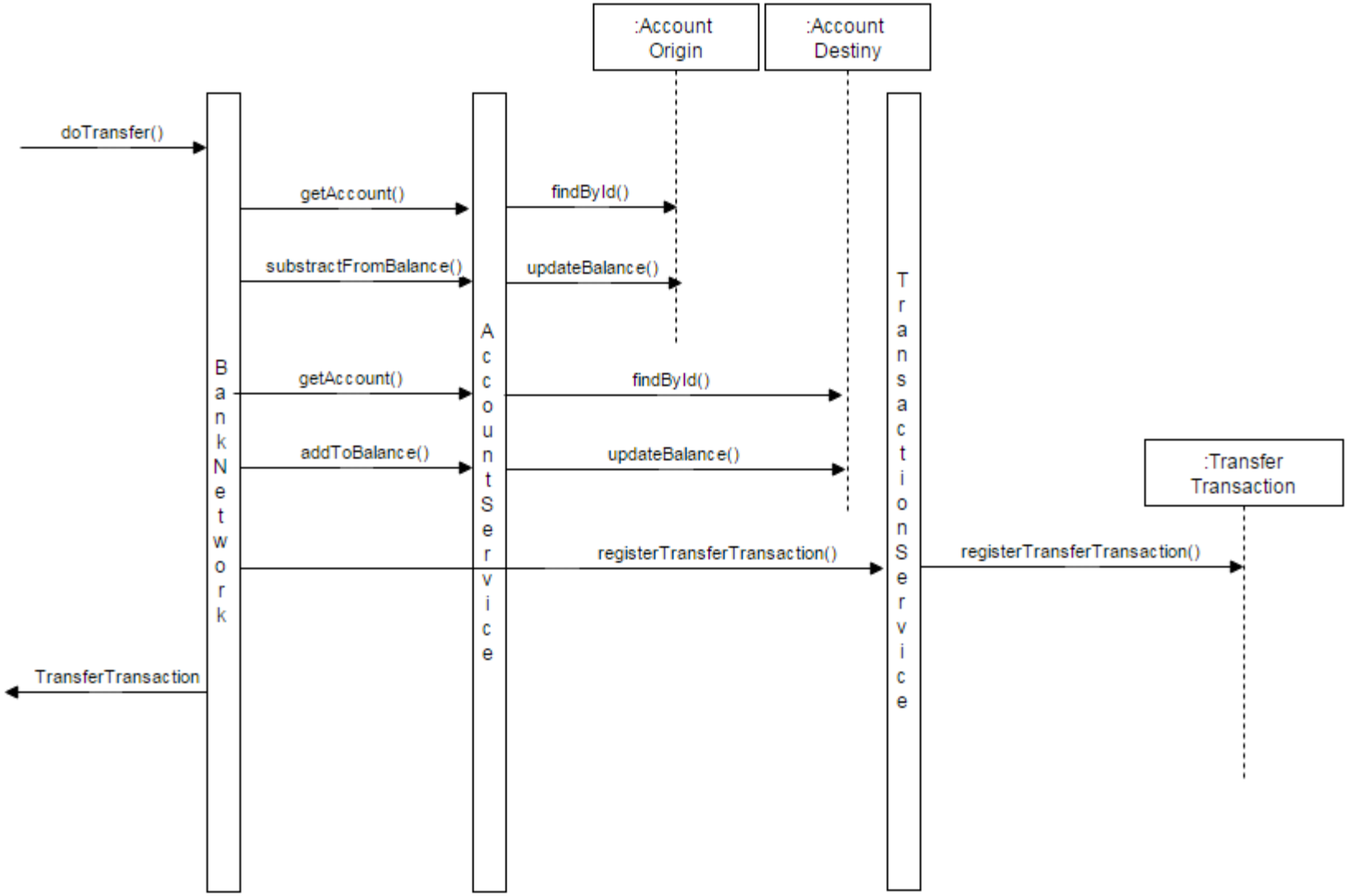
Credit

- The “Credit”



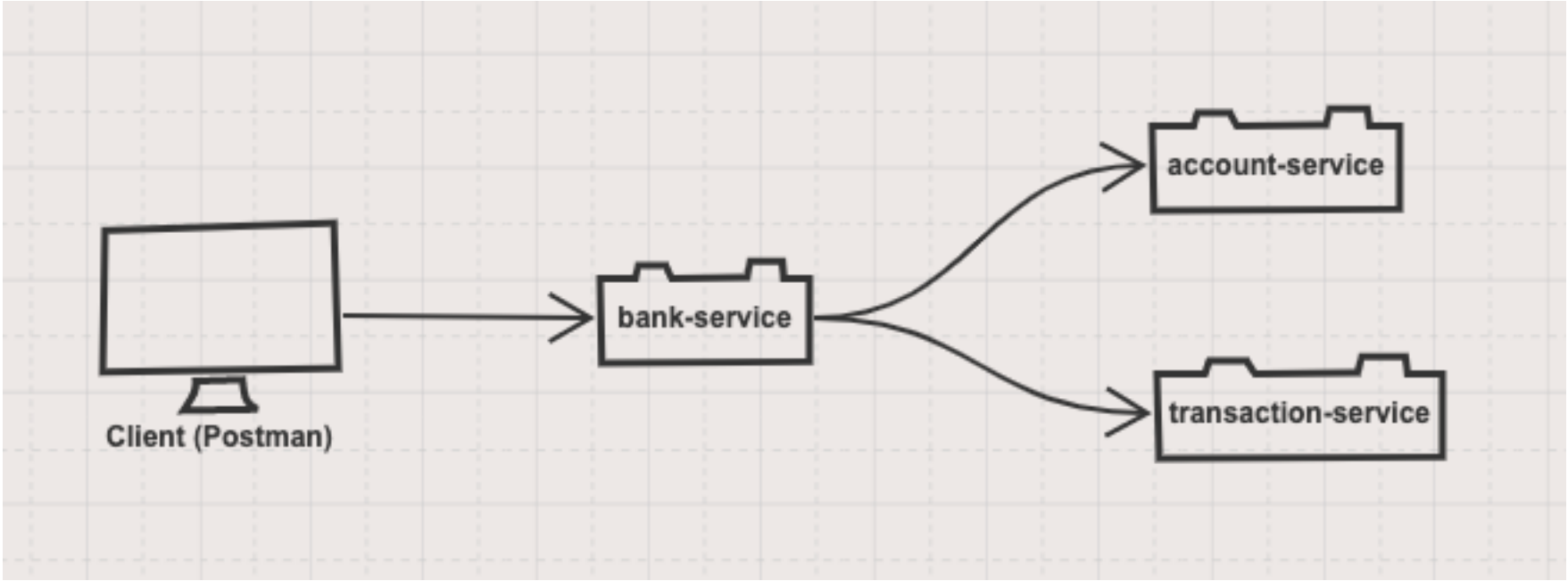
Our project...

- The “Transfer”



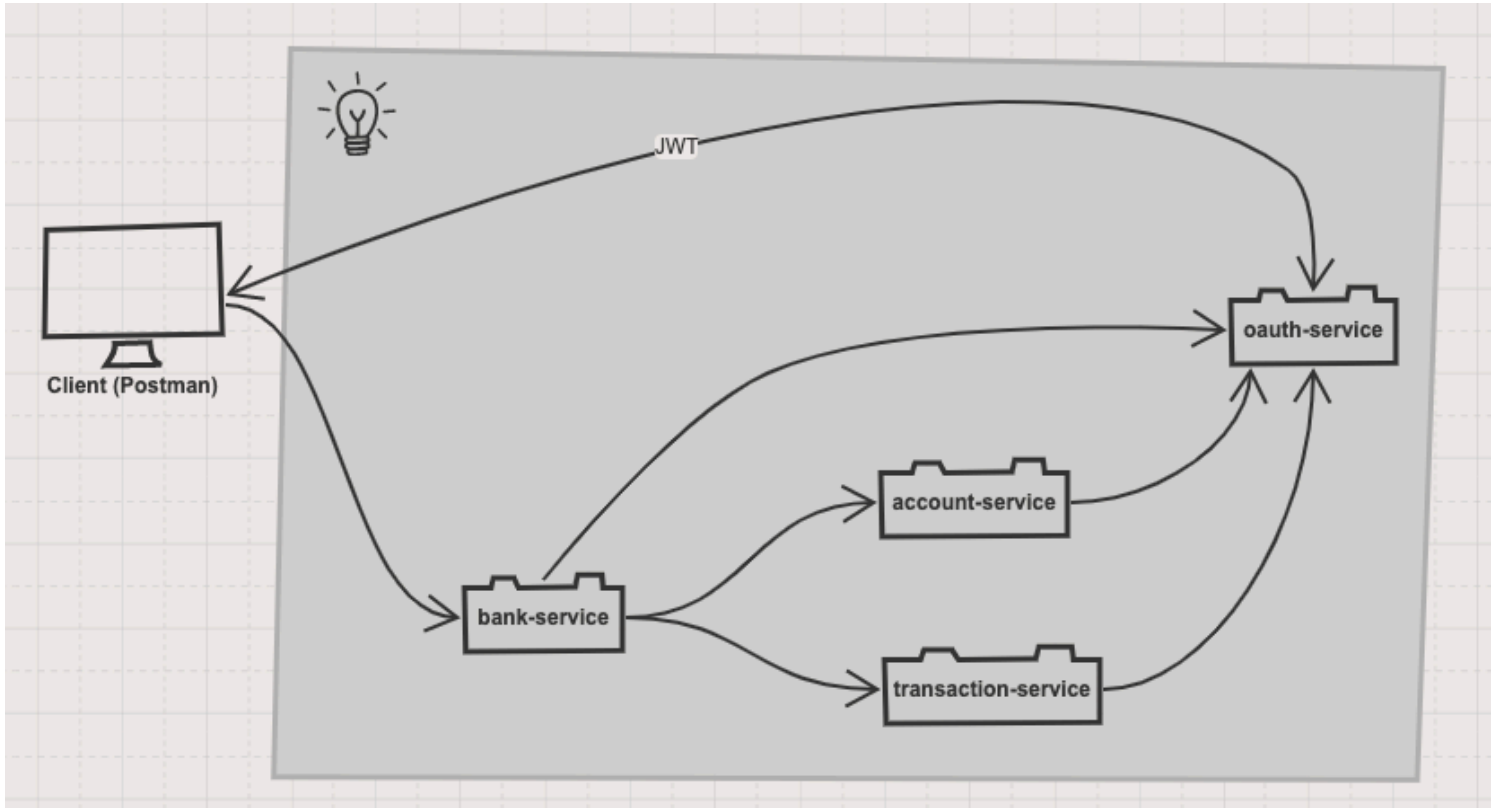
Our project...

- Milestone-1.0



Our project...

- Milestone-2.0



Recap & QA

Recap & QA

- Inversion of Control (IoC)… based on “Hollywood” principle, “Don’t call us, we’ll call you”
- Dependency Injection (DI)... characteristic of IoC.

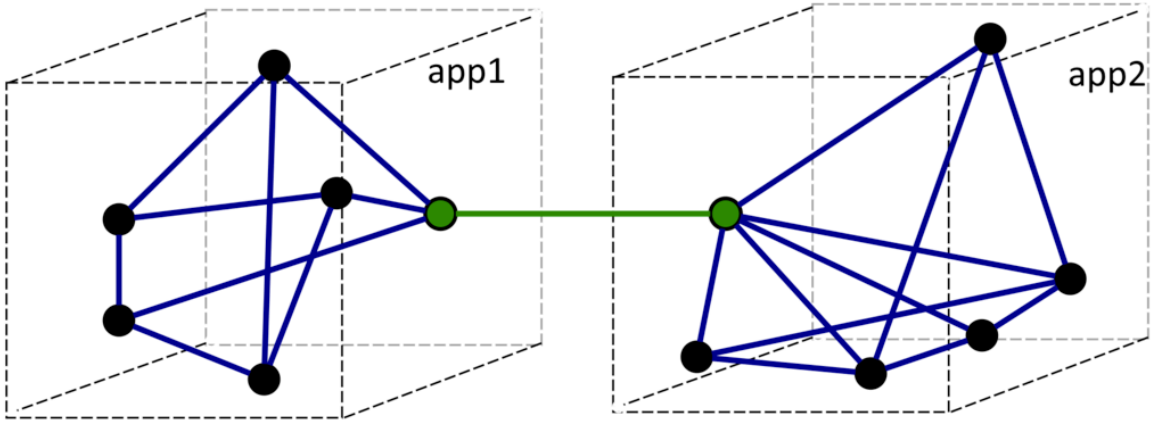
```
16          // Let's program against an interface
17      //    Hello hello = new HelloWorld();
18      //    Hello hello = new HelloMoon();
19          Hello hello = new HelloTest();
20
21          ReusableBot reusableBot = new ReusableBot(hello);
22          System.out.println(reusableBot.greeting());
23
```

Recap & QA

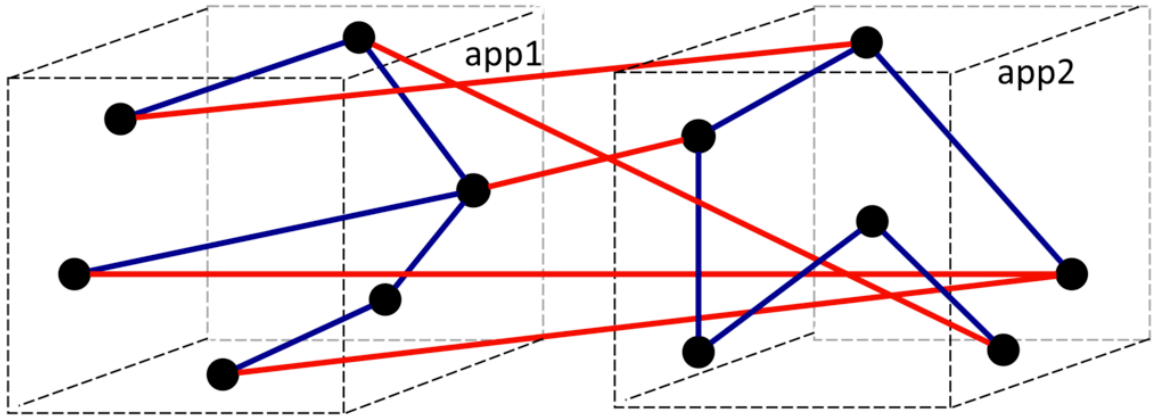
- Spring Framework is a DI framework.
- Spring has many modules and give us the option to include what we need (not opinionated).

Recap & QA

- Cohesion?



a) Good

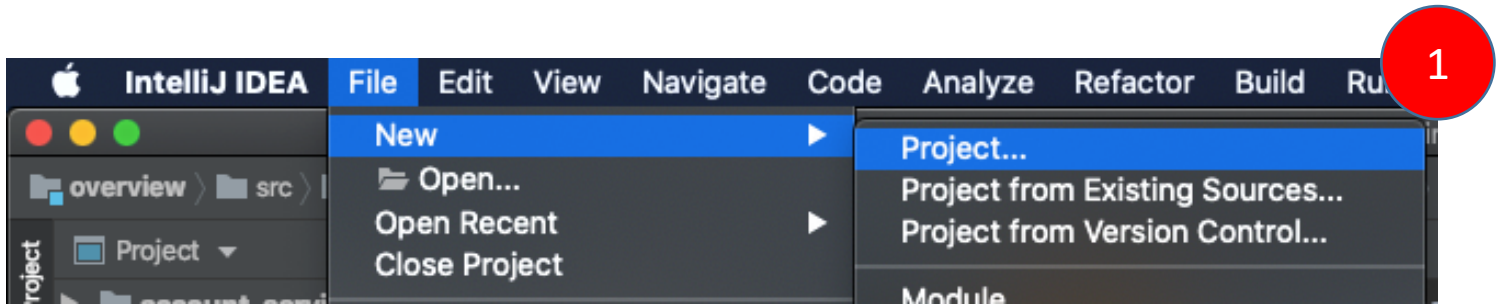


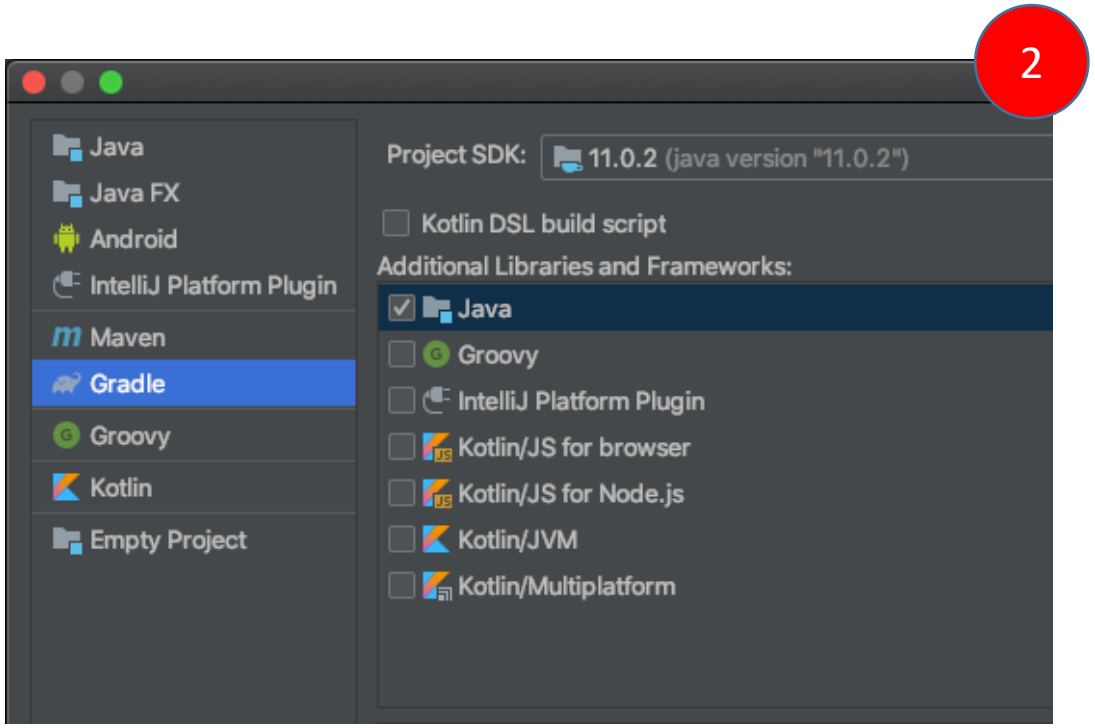
b) Bad

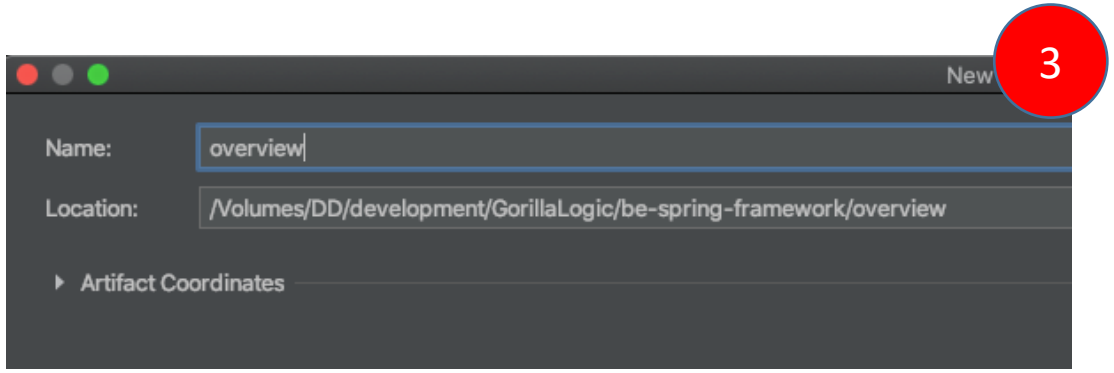
Spring Core – @nnotations

by Andrés Arcia

Let’s create our project







Let’s create our project

- compile group: 'org.springframework', name: 'spring-core', version: '5.2.5.RELEASE' //Spring Core
- compile group: 'org.springframework', name: 'spring-context', version: '5.2.5.RELEASE' //Spring Context
- testCompile group: 'org.springframework', name: 'spring-test', version: '5.2.5.RELEASE' //Spring Test

```
14  dependencies {
15      compile group: 'org.springframework', name: 'spring-core', version: '5.2.5.RELEASE'     //Spring Core
16      compile group: 'org.springframework', name: 'spring-context', version: '5.2.5.RELEASE'  //Spring Context
17
18      testCompile group: 'org.springframework', name: 'spring-test', version: '5.2.5.RELEASE' //Spring Test
19      testCompile group: 'junit', name: 'junit', version: '4.12'
20  }
21
```

Repo: https://github.com/spring-projects/spring-framework

Spring - @nnotations – Part 1

- @Configuration
- @Bean
- @Bean(“…”)
- @Import(“…”)
- @ComponentScan(“…”)
- @Scope(“…”)
- @Lazy(false/true)
- @Autowired
- @Autowired(required=false) **Deprecated*
- @Qualifier(“…”)
- @PostConstruct and @PreDestroy

Pure Java (the old way)

```java
package com.sp.repository.user;

import java.sql.Connection;

public class OldJDBCUserRepository implements UserRepository{

    private DataSource dataSource;

    public OldJDBCUserRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }


    @Override
    public User registerUser(String name) {
        String sql = "insert into T_USER (USER_ID, USER_NAME) values (?, ?)";
        Connection conn = null;
        PreparedStatement ps = null;
        try {
            conn = dataSource.getConnection();
            ps = conn.prepareStatement(sql);
            String id = nextIdNumber();
            ps.setString(1, id);
            ps.setString(2, name);
            ps.execute();
            return new User(id, name);
        } catch (SQLException e) {
            throw new RuntimeException("SQL exception occured inserting user record", e);
        } finally {
            if (ps != null) {
                try {
                    // Close to prevent database cursor exhaustion
                    ps.close();
                } catch (SQLException ex) {
                }
            }
            if (conn != null) {
                try {
                    // Close to prevent database connection exhaustion
                    conn.close();
                } catch (SQLException ex) {
                }
            }
        }
    }
```

```java
package com.sp.repository.user;

public class UserServiceImpl implements UserService {

    private UserRepository userRepository;

    public UserServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
```

```java
package com.sp.model;

import java.util.List;

public class BankNetworkImpl implements BankNetwork {


    private UserService userService;

    public BankNetworkImpl() {
        UserRepository userRepository = new OldJDBCUserRepository(getDataSource());
        userService = new UserServiceImpl(userRepository);
    }

    public DataSource getDataSource(){
        return null; //A DataSource returned
    }
```

@Configuration

```java
package com.sp.config;

import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfigOption1 {

}
```

- **@Configuration** annotation lets Spring know where to start looking for configuration details
- “AppConfigOption1” is a normal class, it does not extend any other class (but it may do) and it might implement an interface (non commonly seen)

@Bean

```java
@Bean(name="MyUserService")
@Scope("singleton")
@Lazy(false)
public UserService userService(){
    return new UserServiceImpl(userRepository());
}
```

- **@Bean** annotation allows Spring to know all resources that will compose the Spring Application Context.

- Parameter "**name**" will determine how to retrieved the bean from Spring context. This is just required when you want to define specific functionality.

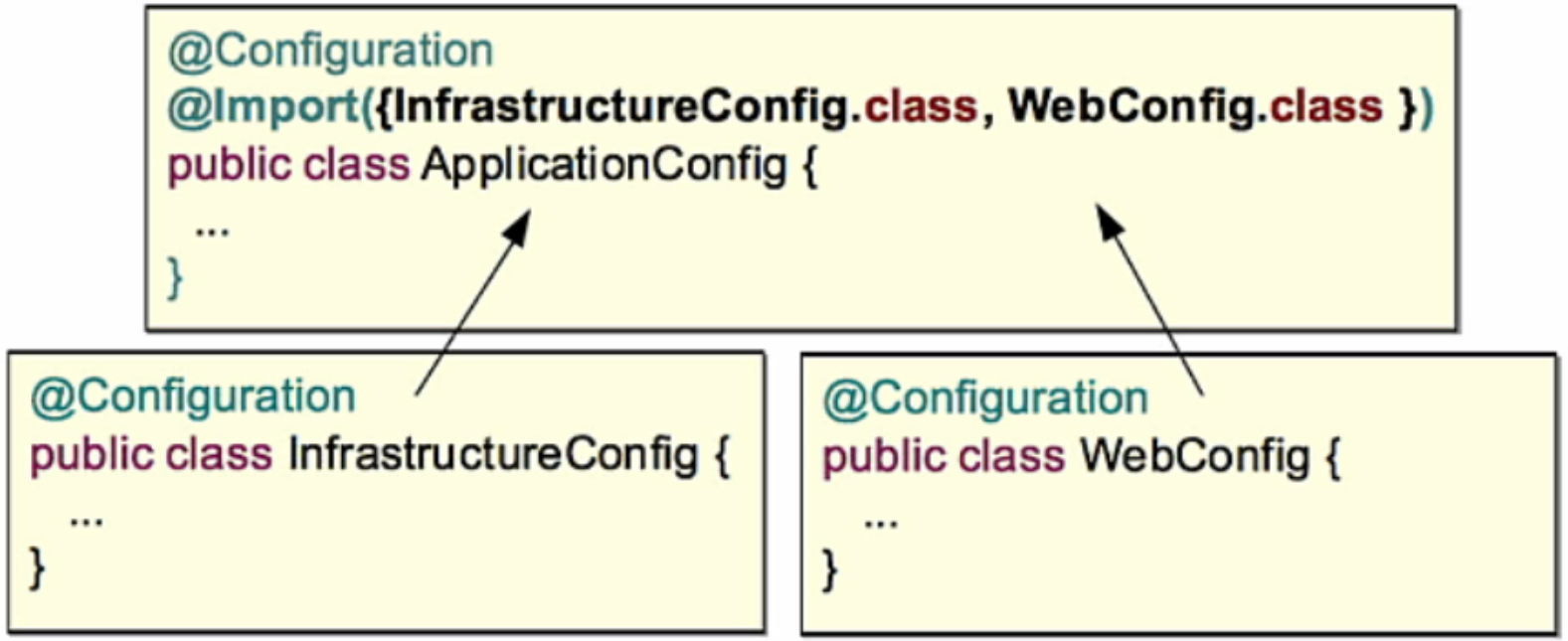
- **@Scope** annotations is used to determined what kind of scope the resource might have (**default singleton**)
  - singleton: a single instance per IoC container.
  - prototype: a new instance is created each time the bean is requested.
  - request: a new instance is created per HTTP request call.
  - session: a new instance is created per HTTP session.
  - global session: a new instance is created to the lifecycle of a global HTTP session.

- **@Lazy** will let Spring know, how to initialize the bean (**default false**)

@Import

```java
@Configuration
@Import({InfrastructureConfig.class})
public class AppConfigOption2 {

}
```

- **@Import** annotation allows Spring to group multiple @Configuration files.
- Easier ton maintain multiple smaller files



@ComponentScan(“…”)

```
@Configuration
@ComponentScan("com.sp")
@Import({InfrastructureConfig.class})
public class AppConfig {


}
```

- **@ComponentScan(“…”)** used to let Spring know what needs to be scanned looking for @Component annotations and it’s childs (@Service, @Repository, @Controller, @Configuration).

- **@ComponentScan(“…”)** annotation receives a String parameter with the path that is going to be scanned.

- Jar dependencies are also scanned.

@ComponentScan("...")

- Really bad:

```
@ComponentScan ( "org,com" )
```

All "org" and "com" packages in the classpath will be scanned!!

- Still bad:

```
@ComponentScan ( "com" )
```

- OK:

```
@ComponentScan ( "com.acme.app" )
```

- Optimized:

```
@ComponentScan ( "com.acme.app.repository,
    com.acme.app.service, com.acme.app.controller" )
```

@Autowired

```java
@Configuration
@Import({InfrastructureConfig.class})
public class AppConfigOption2 {

    //Using the @Autowire Spring will find an object that can resolve
    //the dependency correctly, variable naming does not impact this
    @Autowired
    private DataSource dataSource;
```

- **@Autowired** is used to reference bean defined in a separate configuration files.

```java
@Configuration
@Import( InfrastructureConfig.class )
public class ApplicationConfig {

  @Autowired
  public DataSource dataSource;


  @Bean
  public AccountRepository accountRepository() {
    return new JdbcAccountRepository( dataSource );
  }
}
```

```java
@Configuration
public class InfrastructureConfig {
  @Bean
  public DataSource dataSource() {
    DataSource ds = new BasicDataSource();
    ...
    return ds;
  }
}
```

@Autowired

- **@Autowired** can be used in different places:

Construct-injection
(Use it for mandatory fields)

```
@Autowired
public TransferServiceImpl(AccountRepository a) {
    this.accountRepository = a;
}
```

Method-injection
(Use it for optional fields)

```
@Autowired
public void setAccountRepository(AccountRepository a) {
  this.accountRepository = a;
}
```

Field-injection
(Should avoid using it bc is hard to be used out of Spring container)

```
@Autowired
private AccountRepository accountRepository;
```

@Autowired(required=**false**) **Deprecated*

- **@Autowired(required=false)** default behavior even if it is not explicitly placed.

- **@Autowired(required=true)** is used if the dependency is not required, in other words if the dependency exists.

```
@Autowired(required=false)
public void setAccountRepository(AccountRepository a) {
   this.accountRepository = a;
}
```

Only inject *if* dependency exists

@Qualifier("...")

- **@Qualifier("...")** is used to determined which bean is going to be injected.

```
@Component
public class TransferServiceImpl implements TransferService {
  @Autowired
  public TransferServiceImpl(AccountRepository accountRepository) { ... }
}
```

```
@Component
public class HibernateAccountRepository implements AccountRepository {..}
```

Which one should get injected?

```
@Component
public class JdbcAccountRepository implements AccountRepository {..}
```

Exception: NoSuchBeanDefinitionException...

@Qualifier("…")

- **@Qualifier("…")** is also available for method and field injection.
- If the ID is not defined Spring will base the search in class name of the @Component annotation.

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
  @Autowired
  public TransferServiceImpl( @Qualifier("jdbcAccountRepository")
      AccountRepository accountRepository) { … }
```

qualifier

```
@Component("jdbcAccountRepository")
public class JdbcAccountRepository implements AccountRepository {..}
```

bean ID

```
@Component("hibernateAccountRepository")
public class HibernateAccountRepository implements AccountRepository {..}
```

@PostConstruct and @PreDestroy

- **@PostConstruct** is always called after the bean is instantiated and all it's dependencies are created.
- **@PreDestroy** is always called just before the bean is destroyed.

- Methods used for **@PostConstruct** and **@PreDestroyed** have to be void and without parameters.

```
public class JdbcAccountRepository {
    @PostConstruct
    void populateCache() { }

    @PreDestroy
    void clearCache() { }
}
```

Method called at startup *after* dependency injection

Method called at shutdown prior to destroying the bean instance

@Configuration – Option 1 (Mixed Configuration)

- **@Configuration** class contains infrastructure and business configurations. This is allowed by not the best practice.
- Injection is done via method local calls.
- The order of object creation matters, it goes from the ones that are not dependent to the ones that have dependencies.
- Spring takes control and initializes our dependencies.

```java
@Configuration
public class AppConfigOption1 {

    //Service beans
    @Bean
    public BankNetwork bankNetwork(){
        return new BankNetworkImpl(accountService(), transactionService());
    }

    @Bean(name="MyUserService")
    @Scope("singleton")
    @Lazy(false)
    public UserService userService(){
        return new UserServiceImpl(userRepository());
    }

    @Bean(name="MyAccountService")
    public AccountService accountService(){
        return new AccountServiceImpl(accountRepository());
    }

    @Bean
    public TransactionService transactionService(){
        return new TransactionServiceImpl(transactionRepository());
    }

    //Respository beans
    @Bean
    public UserRepository userRepository(){
        return new JdbcUserRepository(dataSource());
    }

    @Bean
    public AccountRepository accountRepository(){
        return new JdbcAccountRepository(dataSource());
    }

    @Bean
    public TransactionRepository transactionRepository(){
        return new JdbcTransactionRepository(dataSource());
    }


    //Data source bean
    @Bean
    public DataSource dataSource(){
        return
            (new EmbeddedDatabaseBuilder())
            .addScript("classpath:com/sp/db/schema.sql")
            .addScript("classpath:com/sp/db/test-data.sql")
            .build();
    }

}
```

@Configuration – Option 1 (Mixed Configuration)

```java
@Configuration
public class AppConfigOption1 {

    //Service beans
    @Bean
    public BankNetwork bankNetwork(){
        return new BankNetworkImpl(accountService(), transactionService());
    }

    @Bean(name="MyUserService")
    @Scope("singleton")
    @Lazy(false)
    public UserService userService(){
        return new UserServiceImpl(userRepository());
    }

    @Bean(name="MyAccountService")
    public AccountService accountService(){
        return new AccountServiceImpl(accountRepository());
    }

    @Bean
    public TransactionService transactionService(){
        return new TransactionServiceImpl(transactionRepository());
    }
```

```java
public class BankNetworkImplTest {

    private BankNetwork bankNetwork;
    private UserService userService;
    private AccountService accountService;
    private TransactionService transactionService;

    private ApplicationContext appContext;

    @Before
    public void setUp() throws Exception {

        //AppContext option 1: All configurations are all together in one file
        appContext = new AnnotationConfigApplicationContext(AppConfigOption1.class);

        //String declaration is case sensitive against @Bean definition in the @Configuration class
        userService = (UserService) appContext.getBean("MyUserService");
        //This method is used when there are 2 definitions in the @Configuration of the same type but with different functionality
        accountService = appContext.getBean("MyAccountService", AccountService.class);
        //Spring automatically resolves against class type in @Configuration class
        transactionService = appContext.getBean(TransactionService.class);

        bankNetwork = appContext.getBean(BankNetwork.class);
    }
```

@Configuration – Option 2 (Partitioning Configuration)

- **@Configuration** class contains only business configurations. Infrastructure configuration is separated in “InfraestructureConfig.class”, which is imported in the beginning of the class by **@Import** annotation.

- Injection is done via method local calls. Only “dataSource” dependency is resolved via **@Autowired**.

```java
@Configuration
@Import({InfrastructureConfig.class})
public class AppConfigOption2 {

    //Using the @Autowire Spring will find an object that can resolve
    //the dependency correctly, variable naming does not impact this
    @Autowired
    private DataSource dataSource;

    //Service beans
    @Bean
    public BankNetwork bankNetwork(){
        return new BankNetworkImpl(accountService(), transactionService());
    }

    @Bean(name="MyUserService")
    @Scope("singleton")
    @Lazy(false)
    public UserService userService(){
        return new UserServiceImpl(userRepository());
    }

    @Bean(name="MyAccountService")
    public AccountService accountService(){
        return new AccountServiceImpl(accountRepository());
    }

    @Bean
    public TransactionService transactionService(){
        return new TransactionServiceImpl(transactionRepository());
    }

    //Respository beans
    @Bean
    public UserRepository userRepository(){
        return new JdbcUserRepository(dataSource);
    }

    @Bean
    public AccountRepository accountRepository(){
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransactionRepository transactionRepository(){
        return new JdbcTransactionRepository(dataSource);
    }
}
```

@Configuration – Option 2 (Partitioning Configuration)

```java
@Configuration
@Import({InfrastructureConfig.class})
public class AppConfigOption2 {

    //Using the @Autowire Spring will find an object that can resolve
    //the dependency correctly, variable naming does not impact this
    @Autowired
    private DataSource dataSource;

    //Service beans
    @Bean
    public BankNetwork bankNetwork(){
        return new BankNetworkImpl(accountService(), transactionService());
    }

    @Bean(name="MyUserService")
    @Scope("singleton")
    @Lazy(false)
    public UserService userService(){
        return new UserServiceImpl(userRepository());
    }

    @Bean(name="MyAccountService")
    public AccountService accountService(){
        return new AccountServiceImpl(accountRepository());
    }

    @Bean
    public TransactionService transactionService(){
```

```java
public class BankNetworkImplTest {

    private BankNetwork bankNetwork;
    private UserService userService;
    private AccountService accountService;
    private TransactionService transactionService;

    private ApplicationContext appContext;

    @Before
    public void setUp() throws Exception {

        //AppContext option 2: Use the annotation @Import to group @Configuration definition classes
        //(Annotation @Configuration is not mandatory in this kind of declaration, with only
        //one @Bean annotation found, Spring knows that is works as @Configuration)
        appContext = new AnnotationConfigApplicationContext(AppConfigOption2.class);


        //String declaration is case sensitive against @Bean definition in the @Configuration class
        userService = (UserService) appContext.getBean("MyUserService");
        //This method is used when there are 2 definitions in the @Configuration of the same type but with different functionality
        accountService = appContext.getBean("MyAccountService", AccountService.class);
        //Spring automatically resolves against class type in @Configuration class
        transactionService = appContext.getBean(TransactionService.class);

        bankNetwork = appContext.getBean(BankNetwork.class);
    }
```

@Configuration – Option 3 (Partitioning Configuration)

- **@Import** annotation is not used.

- Injection is done via method local calls. Only “dataSource” dependency is resolved via **@Autowired**.

```java
@Configuration
public class AppConfigOption3 {

    @Autowired
    private DataSource dataSource;

    //Service beans
    @Bean
    public BankNetwork bankNetwork(){
        return new BankNetworkImpl(accountService(), transactionService());
    }

    //The DI is done as a method variable and not via @Autowired, Spring is
    //able to resolve this dependency without issues
    @Bean(name="MyUserService")
    @Scope("singleton")
    @Lazy(false)
    public UserService userService(DataSource localDataSource){
        return new UserServiceImpl(userRepository(localDataSource));
    }

    @Bean(name="MyAccountService")
    public AccountService accountService(){
        return new AccountServiceImpl(accountRepository());
    }

    @Bean
    public TransactionService transactionService(){
        return new TransactionServiceImpl(transactionRepository());
    }

    //Respository beans
    @Bean
    public UserRepository userRepository(DataSource localDataSource){
        return new JdbcUserRepository(localDataSource);
    }

    @Bean
    public AccountRepository accountRepository(){
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransactionRepository transactionRepository(){
        return new JdbcTransactionRepository(dataSource);
    }
}
```

@Configuration – Option 3 (Partitioning Configuration)

```java
@Configuration
public class AppConfigOption3 {

    @Autowired
    private DataSource dataSource;

    //Service beans
    @Bean
    public BankNetwork bankNetwork(){
        return new BankNetworkImpl(accountService(), transactionService());
    }

    //The DI is done as a method variable and not via @Autowired, Spring is
    //able to resolve this dependency without issues
    @Bean(name="MyUserService")
    @Scope("singleton")
    @Lazy(false)
    public UserService userService(DataSource localDataSource){
        return new UserServiceImpl(userRepository(localDataSource));
    }

    @Bean(name="MyAccountService")
    public AccountService accountService(){
        return new AccountServiceImpl(accountRepository());
    }

    @Bean
    public TransactionService transactionService(){
```

```java
public class BankNetworkImplTest {

    private BankNetwork bankNetwork;
    private UserService userService;
    private AccountService accountService;
    private TransactionService transactionService;

    private ApplicationContext appContext;

    @Before
    public void setUp() throws Exception {

        //AppContext option 3: Explicitly declare all @Configuration classes to use in the ApplicationContext
        //definition (Annotation @Configuration is not mandatory in this kind of declaration, with only
        //one @Bean annotation found, Spring knows that is works as @Configuration)
        appContext = new AnnotationConfigApplicationContext(AppConfigOption3.class, InfrastructureConfig.class);


        //String declaration is case sensitive against @Bean definition in the @Configuration class
        userService = (UserService) appContext.getBean("MyUserService");
        //This method is used when there are 2 definitions in the @Configuration of the same type but with different functionality
        accountService = appContext.getBean("MyAccountService", AccountService.class);
        //Spring automatically resolves against class type in @Configuration class
        transactionService = appContext.getBean(TransactionService.class);

        bankNetwork = appContext.getBean(BankNetwork.class);
    }
```

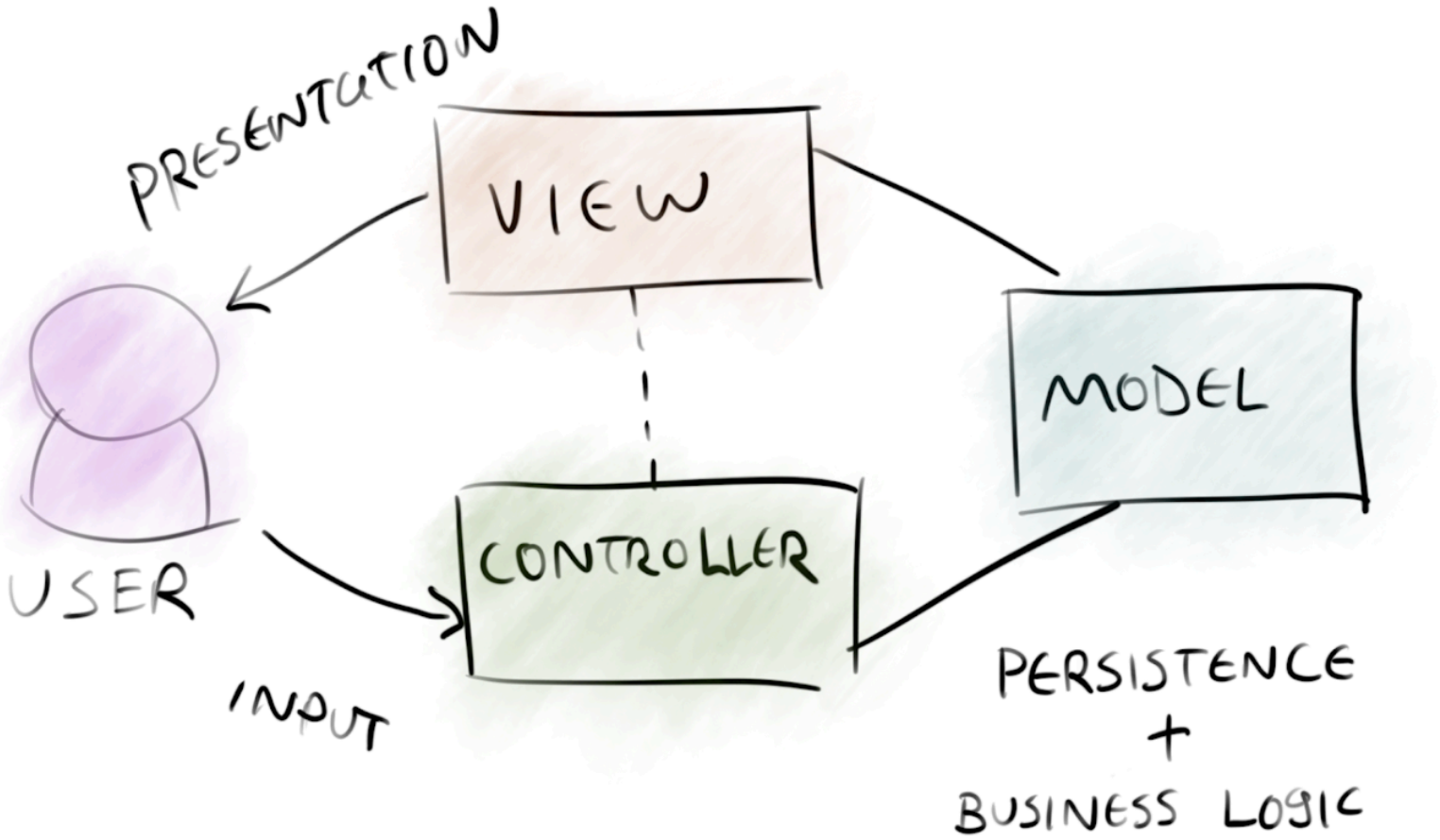
When to use what?

- Pros:
  - Is centralized in one (or a few) places
  - Write any Java code you need
  - Strong type checking enforced by compiler
  - Can be used for all classes

- Cons:
  - More verbose tan annotations

MVC

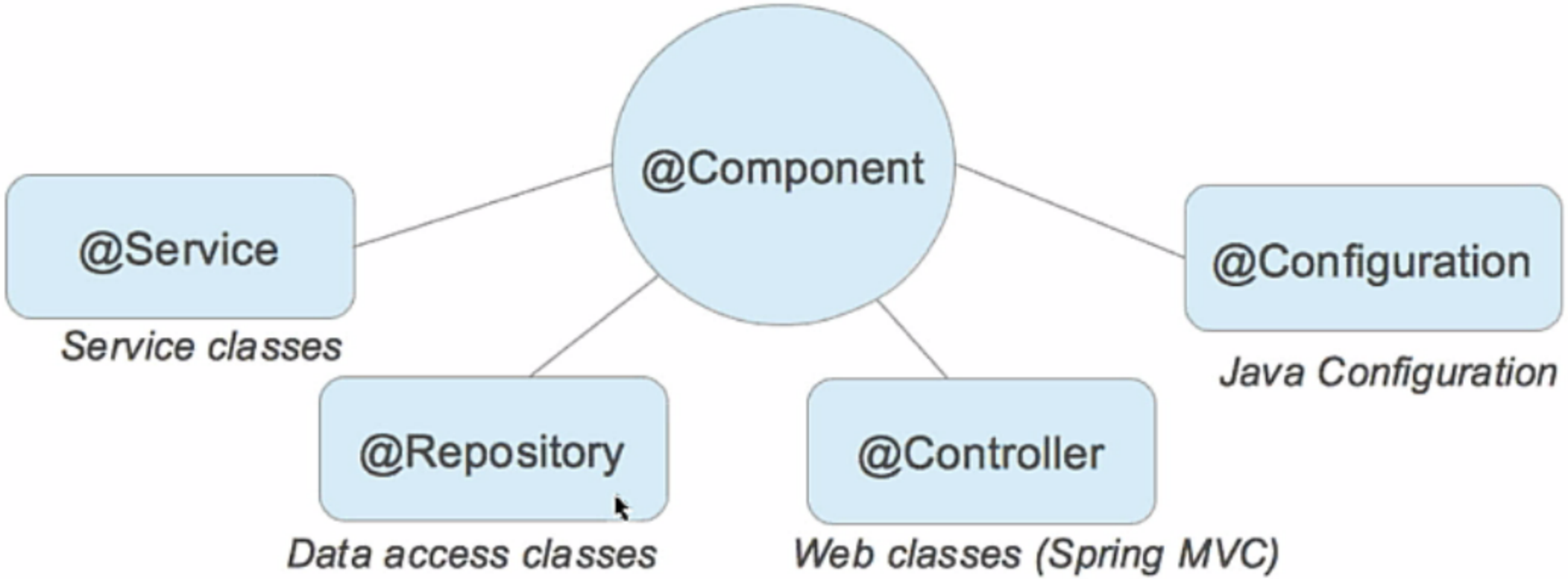
- Single Responsibility
- Separation of concerns



Spring - @nnotations – Part 2

- @Component / @Component(“…”)
- @Repository() / @Repository(“…”)
- @Service() / @Service(“…”)
- @Controller() / @Controller(“…”)

@Component


@Component
@Service
Service classes
@Repository
Data access classes
@Controller
Web classes (Spring MVC)
@Configuration
Java Configuration

@Component

```java
@Component
public class TransactionServiceImpl implements TransactionService {

    @Autowired
    private TransactionRepository transactionRepository;
```

```java
@Component("MyUserServiceImpl")
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;
```

- **@Component** annotation lets Spring know is a resource that is going to be used by Spring somewhere else by injection.

- **@Component(“…”)** might receive a String as parameter, that defines the ID of the resource in the Spring Application Context. This will work along with **@Qualifier(“…”)** annotation

- More generic uses

@Repository

```java
@Repository
public class JdbcTransactionRepository implements TransactionRepository {

    @Autowired
    private DataSource dataSource;
```

- **@Repository** annotation lets Spring know is a resource that is going to be used by Spring somewhere else by injection. **Normally for the repository/DAO layer in MVC architecture**.

- **@Repository("…")** might receive a String as parameter, that defines the ID of the resource in the Spring Application Context.

@Service

```java
@Service
public class TransactionServiceImpl implements TransactionService {

    @Autowired
    private TransactionRepository transactionRepository;
```

- **@Service** annotation lets Spring know is a resource that is going to be used by Spring somewhere else by injection. **Normally for the service layer in MVC architecture**.

- **@Service("…")** might receive a String as parameter, that defines the ID of the resource in the Spring Application Context.

@Controller

- **@Controller** annotation lets Spring know is a resource that is going to be used by Spring somewhere else by injection. **Normally for the controller layer in MVC architecture**.

- **@Controller(“…”)** might receive a String as parameter, that defines the ID of the resource in the Spring Application Context.