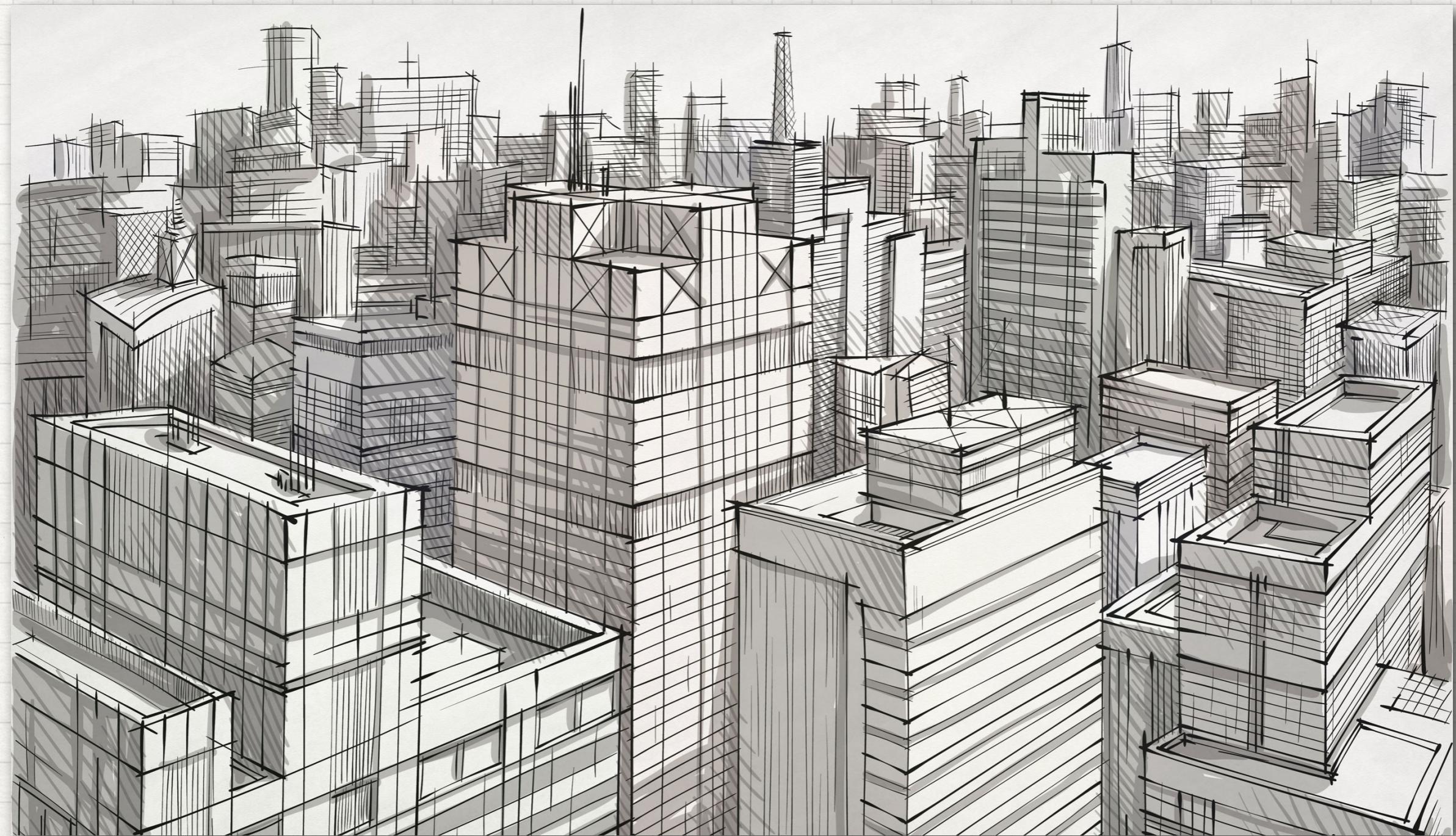


ВВЕДЕНИЕ В SPRING

воЩило юрий





колесо знает о
машине, которую
оно перемещает

колеса не знаю
друг о друге

Машина знает о
колесах, которые она
будет вращать

- IoC (Inversion of Control) — инверсия управления.
- не вы управляете процессом выполнения кода/программы, а фреймворк это делает за вас. Вы передали ему управление (инверсия управления).

- Под DI понимают то Dependency Inversion (инверсию зависимостей, то есть попытки не делать жестких связей между вашими модулями/классами, где один класс напрямую завязан на другой)
- Dependency Injection (внедрение зависимостей, это когда объекты котиков создаете не вы в main-е и потом передаете их в свои методы, а за вас их создает спринг)

- Bean (бин) - просто объект какого-то класса, созданный DI
- application context (Контекст) - это набор бинов (объектов).

```
package by.cources.spring.task1;

public class Car {

    private Wheel firstWheel;
    private Wheel secondWheel;

    public Car() {
        firstWheel = new Wheel(this);
        secondWheel = new Wheel(this);
    }

    public Wheel getFirstWheel() {
        return firstWheel;
    }

    public void setFirstWheel(Wheel firstWheel) {
        this.firstWheel = firstWheel;
    }

    public Wheel getSecondWheel() {
        return secondWheel;
    }

    public void setSecondWheel(Wheel secondWheel) {
        this.secondWheel = secondWheel;
    }

    public void go() {
        firstWheel.rotate();
        secondWheel.rotate();
    }
}
```

```
package by.cources.spring.task1;

public class Wheel {

    private Car owner;

    public Wheel(Car car) {
        setOwner(car);
    }

    public Car getOwner() {
        return owner;
    }

    public void setOwner(Car owner) {
        this.owner = owner;
    }

    public void rotate() {
        // change position of car
    }
}

public static void main(String[] args) {
    Car car = new Car();
    car.go();
}
```



колеса связаны с машиной

колеса связаны с валом, который их крутит

вал связан с машиной

```
package by.cources.spring.task1;

public class Car {

    private Wheel firstWheel;
    private Wheel secondWheel;

    public Car() {
        firstWheel = new Wheel(this);
        secondWheel = new Wheel(this);
    }

    public Wheel getFirstWheel() {
        return firstWheel;
    }

    public void setFirstWheel(Wheel firstWheel) {
        this.firstWheel = firstWheel;
    }

    public Wheel getSecondWheel() {
        return secondWheel;
    }

    public void setSecondWheel(Wheel secondWheel) {
        this.secondWheel = secondWheel;
    }

    public void go() {
        firstWheel.rotate();
        secondWheel.rotate();
    }
}
```

```
package by.cources.spring.task1;

public class Wheel {

    private Car owner;

    public Wheel(Car car) {
        setOwner(car);
    }

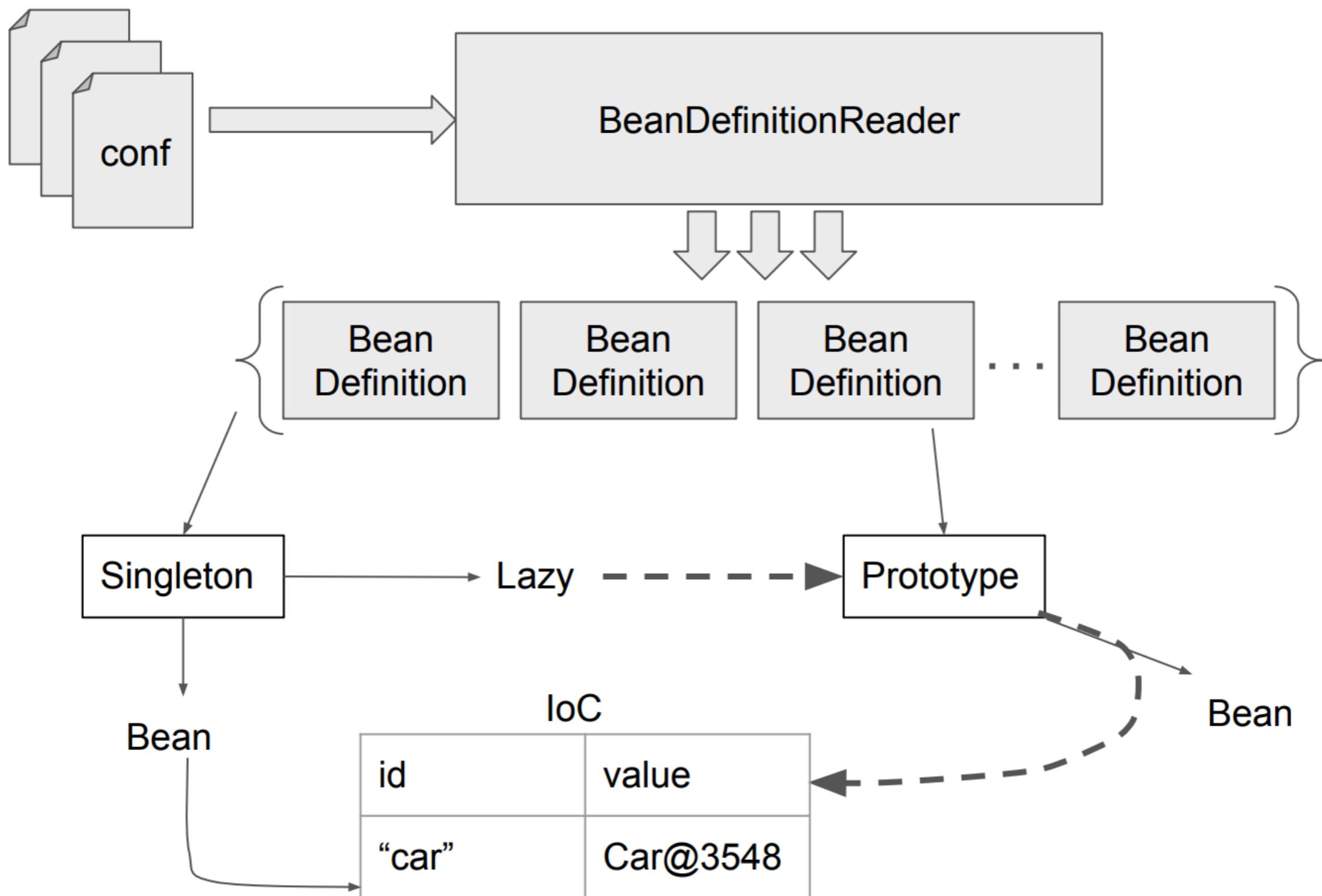
    public Car getOwner() {
        return owner;
    }

    public void setOwner(Car owner) {
        this.owner = owner;
    }

    public void rotate() {
        // change position of car
    }
}

public static void main(String[] args) {
    Car car = new Car();
    car.go();
}
```

Жизненный цикл Spring-приложения (упрощенный)



- property files
- xml files
- groovy config
- java config

- 1. BOOT
- 2. MOBILE
- 3. FOR ANDROID
- 4. FRAMEWORK
- 5. CLOUD
- 6. SOCIAL
- 7. DATA
- 8. INTEGRATION
- 9. BATCH
- 10. SECURITY
- 11. STATE MACHINE
- 12. SESSION
- 13. LDAP
- 14. WEB SERVICES
- 15. WEB FLOW
- 16. REST DOCS
- 17. CLOUD DATA FLOW
- 18. AMQP
- 19. HATEOAS
- 20. TEST



Spring Framework Runtime

Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

Web

WebSocket

Servlet

Web

Portlet

AOP

Aspects

Instrumentation

Messaging

Core Container

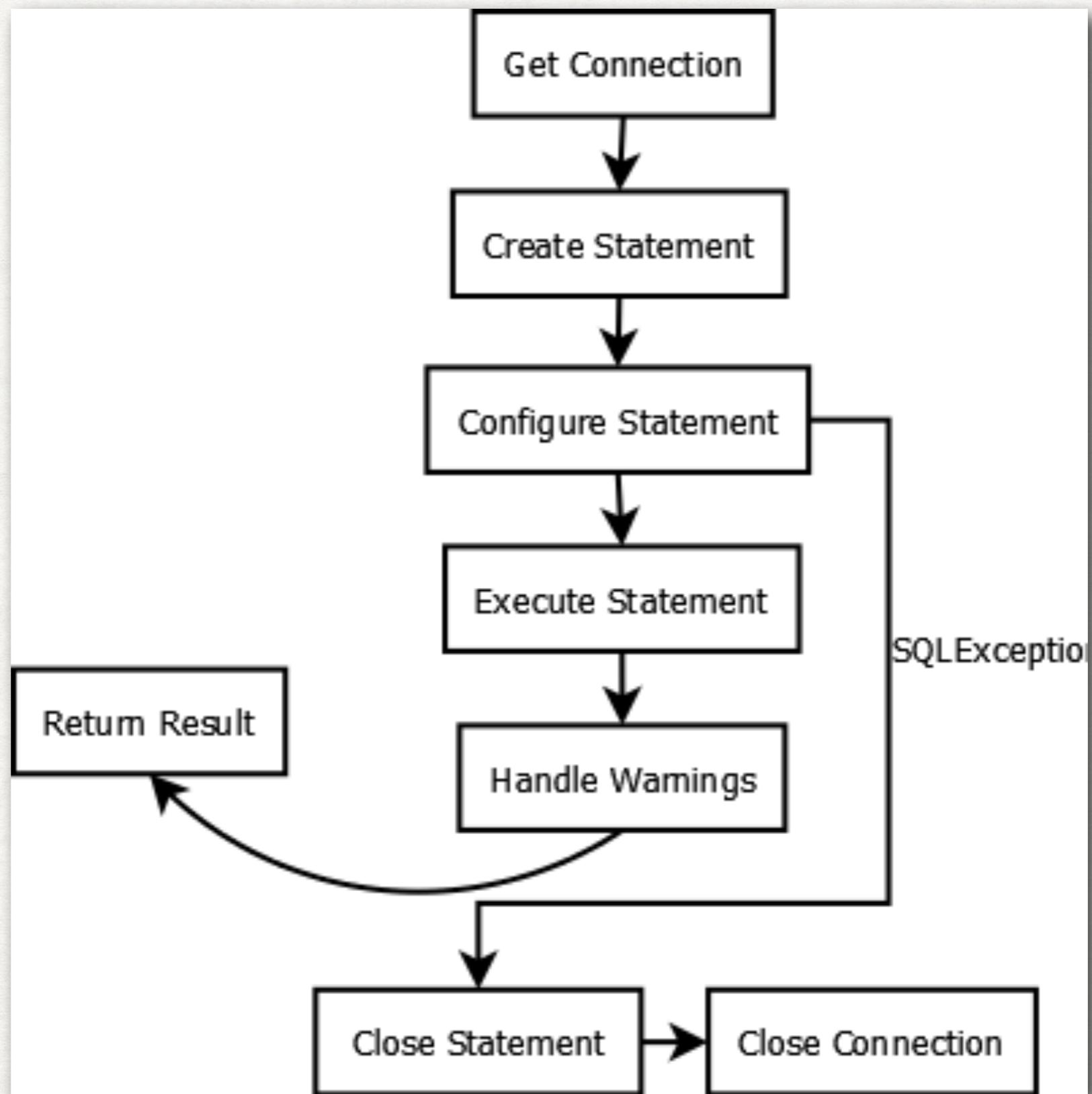
Beans

Core

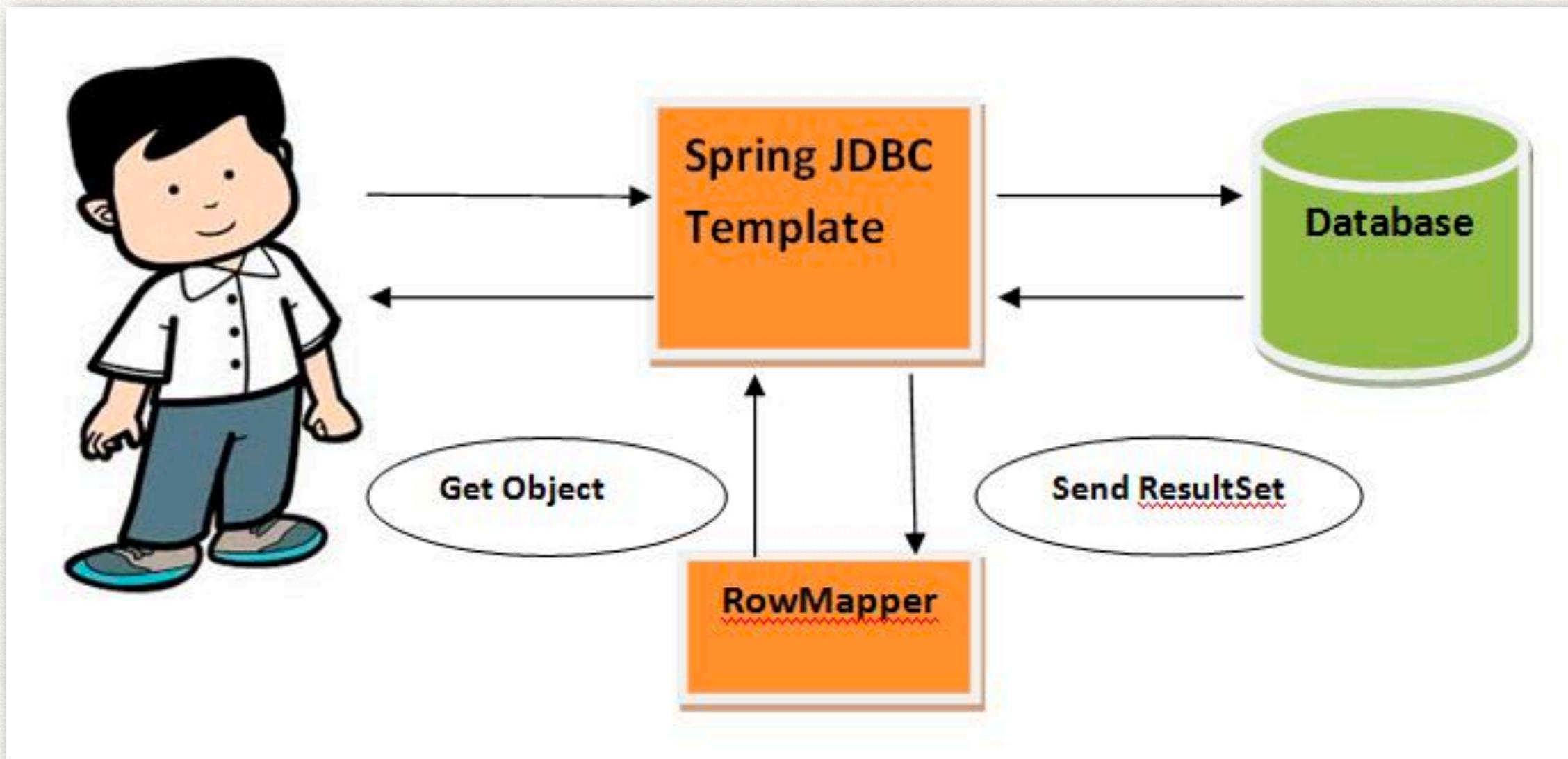
Context

SpEL

Test



SPRING JDBC



- data source
- row mapper
- jdbc template
- jdbc repository

```
@Bean  
DataSource dataSource() {  
    DriverManagerDataSource source = new DriverManagerDataSource();  
    source.setUrl(environment.getProperty("url"));  
    source.setUsername(environment.getProperty("dbuser"));  
    source.setPassword(environment.getProperty("dbpassword"));  
    source.setDriverClassName(environment.getProperty("driver"));  
    return source;  
}  
.
```

- data source
- row mapper
- jdbc template
- jdbc repository

```
package by.cources.spring.task2.spring.repository.mapping;

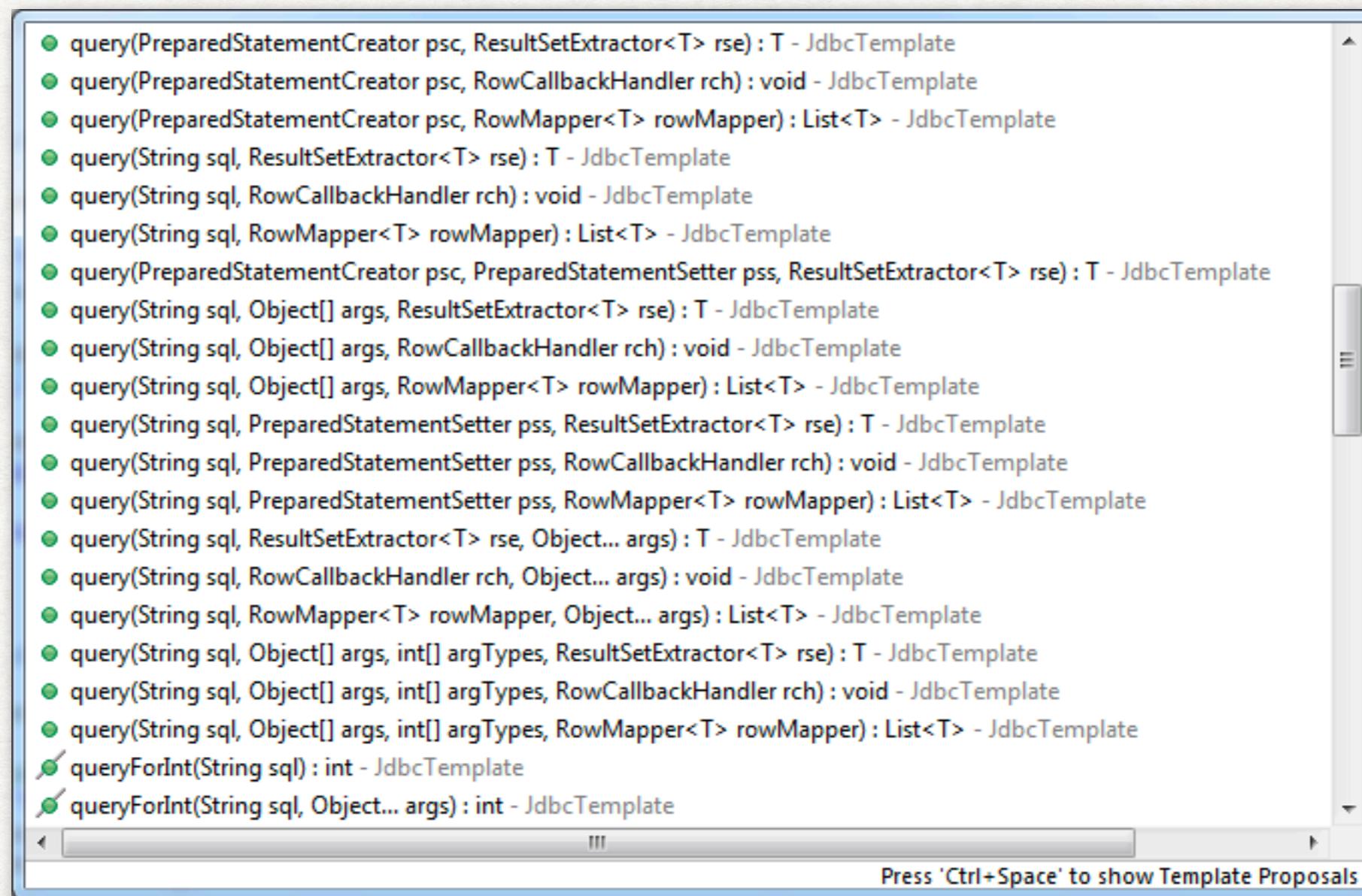
import by.cources.spring.task2.spring.model.Book;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.springframework.jdbc.core.RowMapper;

public class BookMapper implements RowMapper<Book> {

    public Book mapRow(ResultSet resultSet, int i) throws SQLException {
        Book person = new Book();
        person.setId(resultSet.getLong("id"));
        person.setName(resultSet.getString("name"));
        return person;
    }
}
```

- data source
- row mapper
- jdbc template
- jdbc repository

<https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html>



- data source
- row mapper
- jdbc template
- jdbc repository

```
package by.cources.spring.task2.spring.repository;

import by.cources.spring.task2.spring.model.Book;
import by.cources.spring.task2.spring.repository.mapping.BookMapper;
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcTemplateBookRepository implements BookRepository {

    private final JdbcTemplate jdbcTemplate;

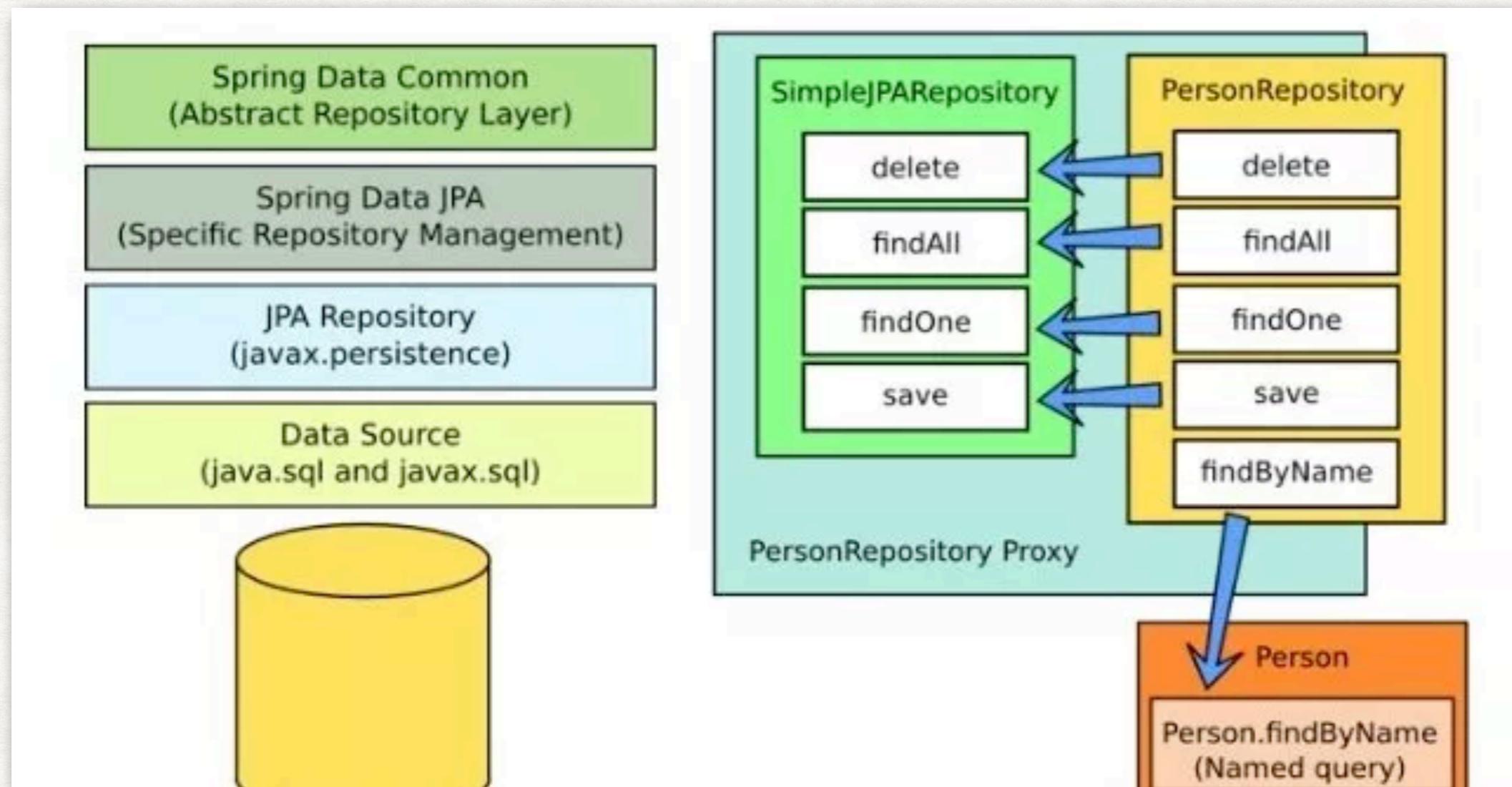
    public JdbcTemplateBookRepository(DataSource ds) {
        this.jdbcTemplate = new JdbcTemplate(ds);
    }

    @Override
    public List<Book> findAll() {
        return jdbcTemplate.query("select * from book", new BookMapper());
    }

    @Override
    public Book findById(Long id) {
        return jdbcTemplate.queryForObject("select * from book where id = ?", new
Object[]{id}, new BookMapper());
    }
}

.
```

SPRING DATA



- Domain / Entity
- Spring Repository
- Spring configuration

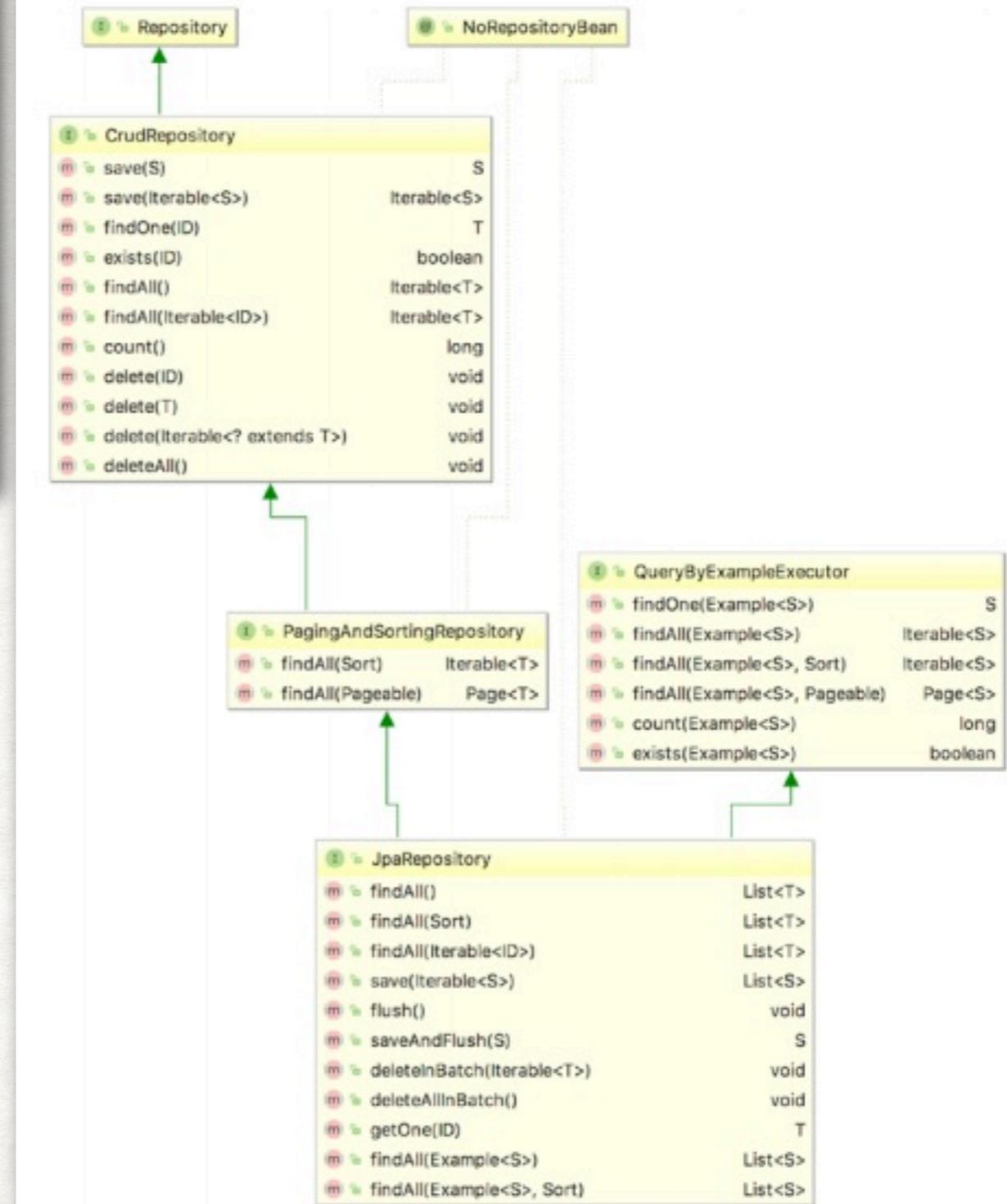
```
@Entity  
@Table(name = "book")  
public class Book {  
  
    @Id  
    @Column(name = "id")  
    private Long id;  
    @Column(name = "name")  
    private String name;  
    @ManyToOne  
    @JoinColumn(name = "author_id")  
    private Author author;
```

```
@Entity  
@Table(name = "author")  
public class Author {  
    @Id  
    @Column(name = "id")  
    private Long id;  
    @Column(name = "first_name")  
    private String firstName;  
    @Column(name = "last_name")  
    private String lastName;  
    @Column(name = "date_of_birth")  
    private LocalDate dateOfBirth;
```

```

I CrudRepository (org.springframework.data.repository)
I JpaRepository (org.springframework.data.jpa.repository)
I JpaRepositoryImplementation (org.springframework.data.jpa)
I PagingAndSortingRepository (org.springframework.data.repository)
C QuerydslJpaRepository (org.springframework.data.jpa.repository)
I ReactiveCrudRepository (org.springframework.data.repository)
I ReactiveSortingRepository (org.springframework.data.repository)
I RevisionRepository (org.springframework.data.repository)
I RxJava2CrudRepository (org.springframework.data.repository)
I RxJava2SortingRepository (org.springframework.data.repository)
C SimpleJpaRepository (org.springframework.data.jpa.repository)

```



```
@Bean  
LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    HibernateJpaVendorAdapter vendorAdapter = new  
    HibernateJpaVendorAdapter();  
    vendorAdapter.setDatabase(Database.HSQL);  
    vendorAdapter.setGenerateDdl(true);  
  
    LocalContainerEntityManagerFactoryBean factory = new  
    LocalContainerEntityManagerFactoryBean();  
    factory.setJpaVendorAdapter(vendorAdapter);  
    factory.setPackagesToScan(getClass().getPackage().getName());  
    factory.setDataSource(dataSource());  
    return factory;  
}
```

.

```
@Bean  
PlatformTransactionManager transactionManager() {  
    JpaTransactionManager txManager = new JpaTransactionManager();  
    txManager.setEntityManagerFactory(entityManagerFactory().get0bject());  
    return txManager;  
}  
.
```

- **Serializable** — транзакции полностью изолируются друг от друга и ни одна транзакция ни коим образом не влияет на другие. Самая низкая степень параллельности транзакций.
- **Repeatable read** — изменения данных, которые были прочитаны в транзакции ранее в транзакцию не попадают, другие транзакции не могут изменять данные, прочитанные этой транзакцией. Возможен эффект фантомного чтения, степерь параллельности транзакций выше, чем у Serializable.

- Read committed — транзакции получают изменения в данных от других транзакций, которые были успешно подтверждены. Возможны эффекты фантомного чтения и неповторяющегося чтения. Этот уровень изоляции обычно используется по умолчанию в базах данных и обеспечивает хорошую степень параллельности транзакций.
- Read uncommitted — самый низший уровень изоляции транзакций, который гарантирует только, что изменения, внесённые одной транзакцией, не будут перезаписаны другой транзакцией. Подвержен всем эффектам влияния транзакций друг на друга. Обеспечивает наивысшую степень параллельности транзакций.

```
@Override  
@Transactional(isolation = Isolation.SERIALIZABLE)  
public Author saveAuthor(Author author) {  
    return authorRepository.save(author);  
}  
.
```

- Propagation.REQUIRED — применяется по умолчанию. При входе в @Transactional метод будет использована уже существующая транзакция или создана новая транзакция, если никакой ещё нет
- Propagation.REQUIRES_NEW — второе по распространённости правило. Транзакция всегда создаётся при входе метод с Propagation.REQUIRES_NEW, ранее созданные транзакции приостанавливаются до момента возврата из метода.
- Propagation.NESTED — корректно работает только с базами данных, которые умеют savepoints. При входе в метод в уже существующей транзакции создаётся savepoint, который по результатам выполнения метода будет либо сохранён, либо откачен. Все изменения, внесённые методом, подтверждатся только позднее, с подтверждением всей транзакции. Если текущей транзакции не существует, будет создана новая.

- Propagation.MANDATORY — обратный по отношению к Propagation.REQUIRES_NEW: всегда используется существующая транзакция и кидается исключение, если текущей транзакции нет.
- Propagation.SUPPORTS — метод с этим правилом будет использовать текущую транзакцию, если она есть, либо будет исполняться без транзакции, если её нет.
- Propagation.NOT_SUPPORTED — одно из самых забавных правил. При входе в метод текущая транзакция, если она есть, будет приостановлена и метод будет выполняться без транзакции.
- Propagation.NEVER — правило, которое явно запрещает исполнение в контексте транзакции. Если при входе в метод будет существовать транзакция, будет выброшено исключение.

```
@Transactional(  
    isolation = Isolation.SERIALIZABLE,  
    propagation = Propagation.REQUIRES_NEW  
)  
@Override  
public Author saveAuthor(Author author) {  
    return authorRepository.save(author);  
}  
•
```

- `value` и `transactionManager` — указывают, какой именно экземпляр `PlatformTransactionManager` использовать, если их несколько.
- `readOnly` — указывает, что транзакция только читает данные, но не изменяет их. Это может быть использовано для оптимизации запросов или блокировок на уровне базы.
- `timeout` — задаёт максимальную длительность операции на стороне базы данных и если эта длительность будет превышена, метод прервётся и транзакция откатится
- `rollbackFor/ rollbackForClassName` — задают список классов исключений, которые вызовут откат транзакции, если метод их выбросит. В коде выше метод `updateGreet()` именно так откатывает транзакцию. По умолчанию, каждое исключение, имеющее в предках `RuntimeError`, вызывает откат транзакции.
- `noRollbackFor/ noRollbackForClassName` — задают список классов исключений, которые не вызовут откат транзакции, если метод их выбросит.
- `propagation` и `isolation` — управляют распространением транзакции и её уровнем изоляции. Я опишу эти параметры в отдельной статье.



Spring Framework Runtime

Data Access/Integration

JDBC

ORM

OXM

JMS

Transactions

Web

WebSocket

Servlet

Web

Portlet

AOP

Aspects

Instrumentation

Messaging

Core Container

Beans

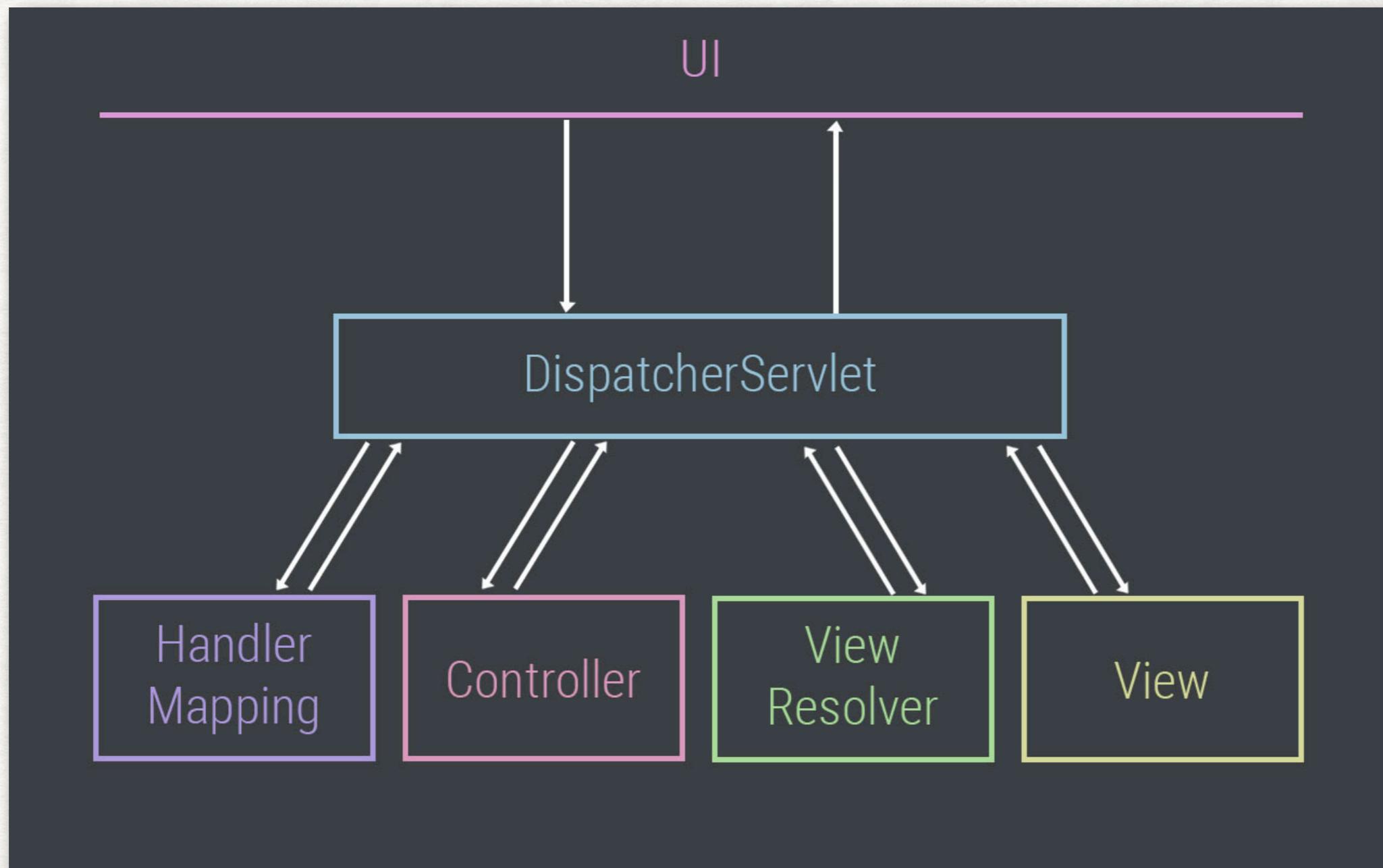
Core

Context

SpEL

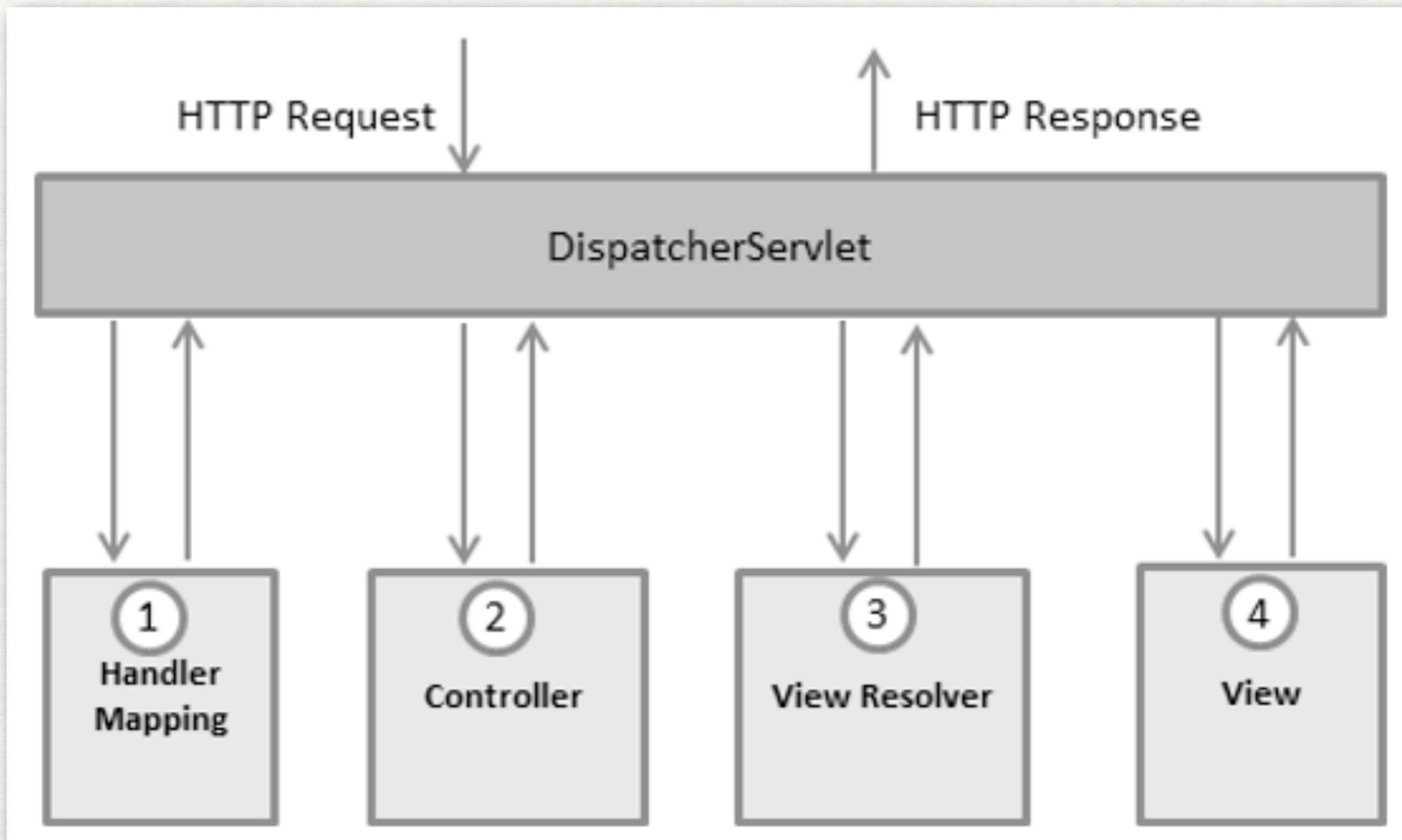
Test

SPRING MVC



- Model (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO («Старых добрых Java-объектов», или бинов).
- View (Отображение, Вид) отвечает за отображение данных Модели, — как правило, генерируя HTML, которые мы видим в своём браузере.
- Controller (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую Модель и передаёт её для отображения в Вид.

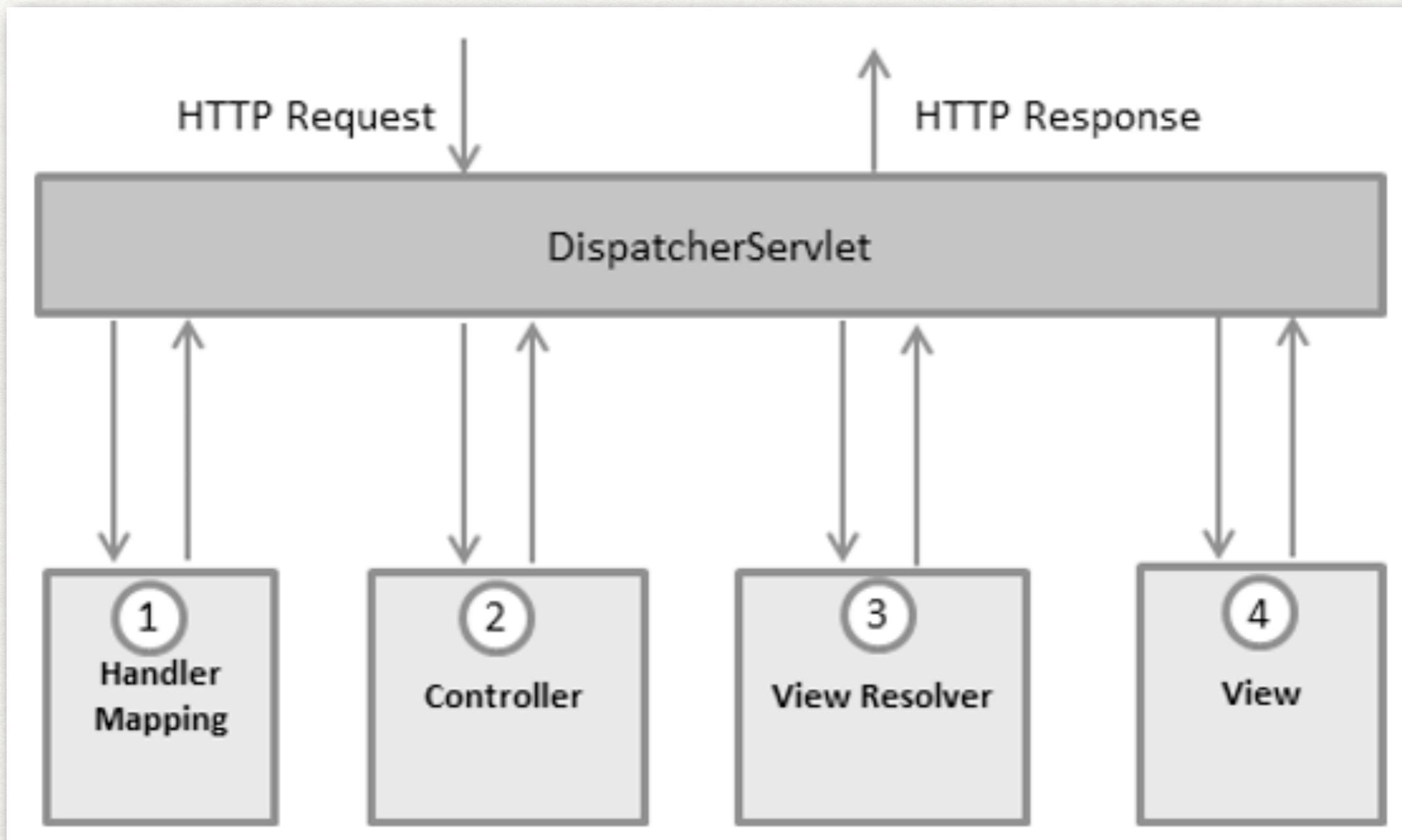
DISPATCHERSERVLET



DISPATCHERSERVLET

- После получения HTTP-запроса DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет, какой Контроллер должен быть вызван, после чего, отправляет запрос в нужный Контроллер.

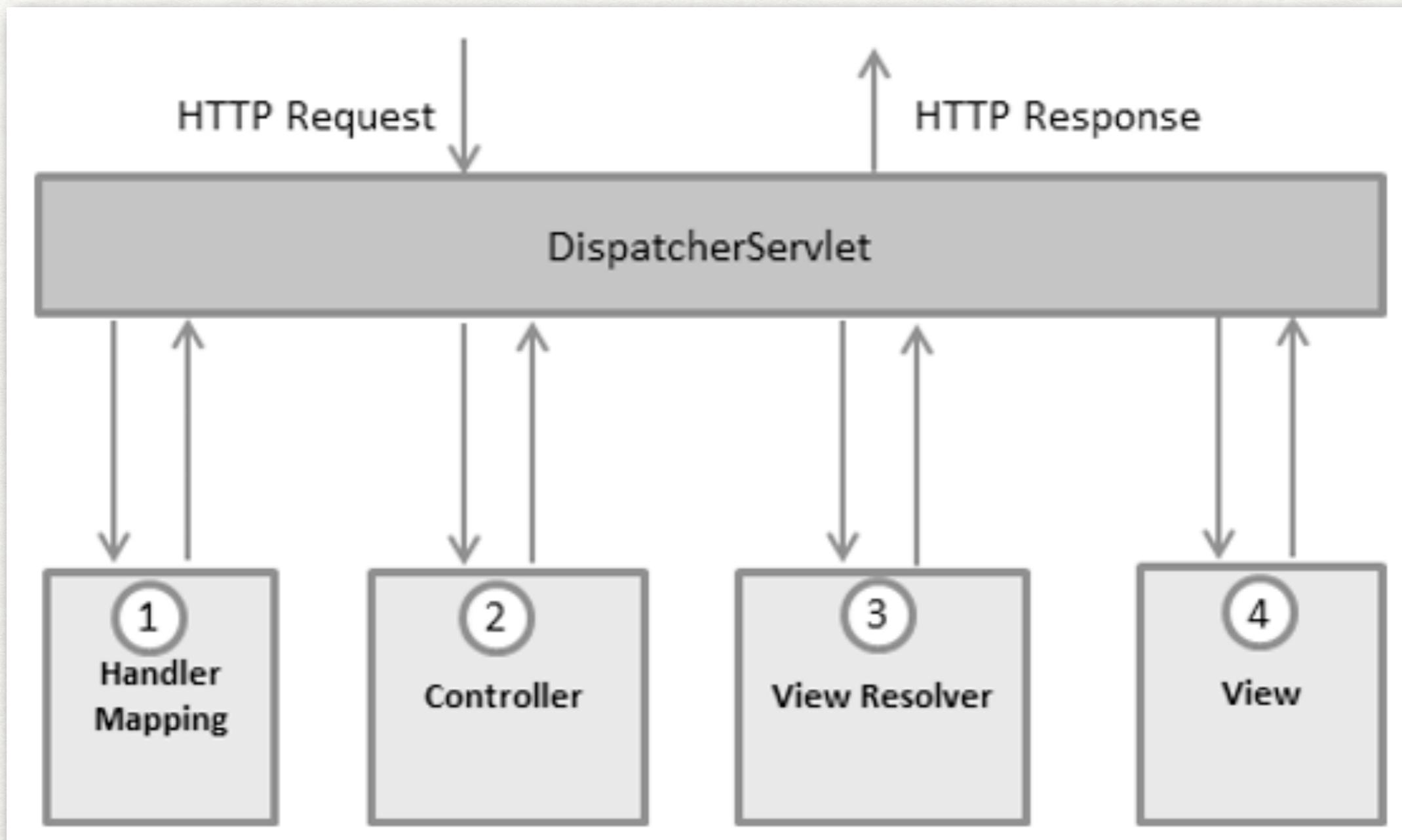
DISPATCHERSERVLET



DISPATCHERSERVLET

- Контроллер принимает запрос и вызывает соответствующий служебный метод, основанный на GET или POST. Вызванный метод определяет данные Модели, основанные на определённой бизнес-логике и возвращает в DispatcherServlet имя Вида (View).

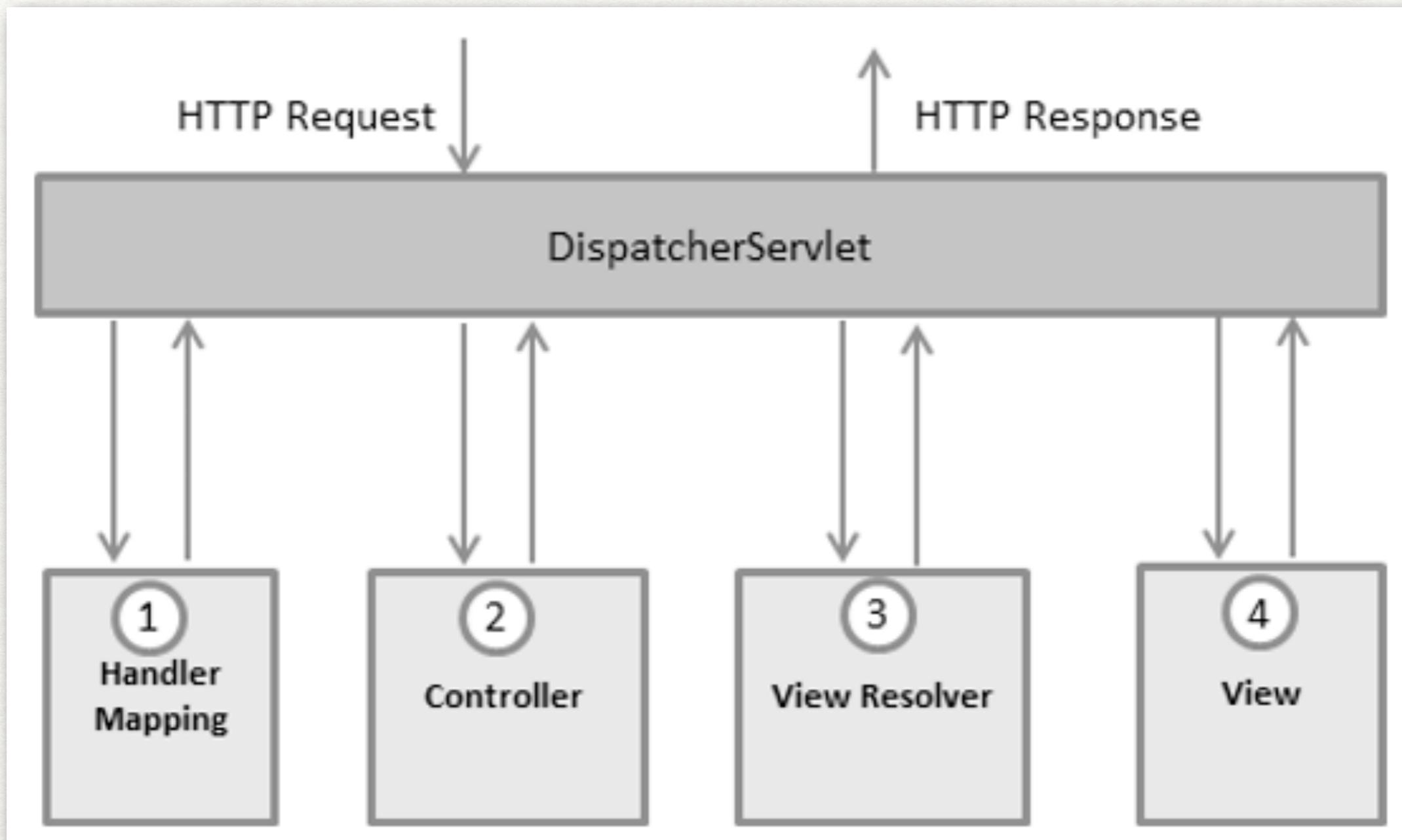
DISPATCHERSERVLET



DISPATCHERSERVLET

- При помощи интерфейса ViewResolver DispatcherServlet определяет, какой Вид нужно использовать на основании полученного имени.

DISPATCHERSERVLET



DISPATCHERSERVLET

- После того, как Вид (View) создан, DispatcherServlet отправляет данные Модели в виде атрибутов в Вид, который в конечном итоге отображается в браузере.

DISPATCHERSERVLET

- Controller определяет класс как Контроллер Spring MVC.
- @RequestMapping указывает, что все методы в данном Контроллере относятся к URL-адресу "/hello".
- @RequestMapping(method = RequestMethod.GET) используется для объявления метода printHello() как дефолтного метода для обработки HTTP-запросов GET (в данном Контроллере).