SPRING BOOT

Spring boot

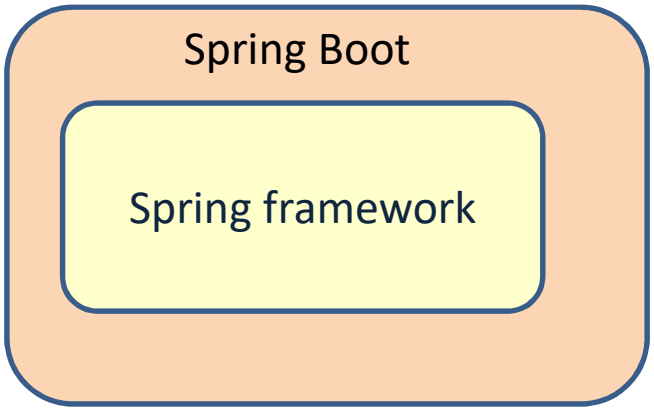
- Framework that makes it easy to configure and run spring applications
- Simple maven configuration
- Default/auto spring configuration
  - Opinionated framework
    - Convention over configuration

Has all the default configuration

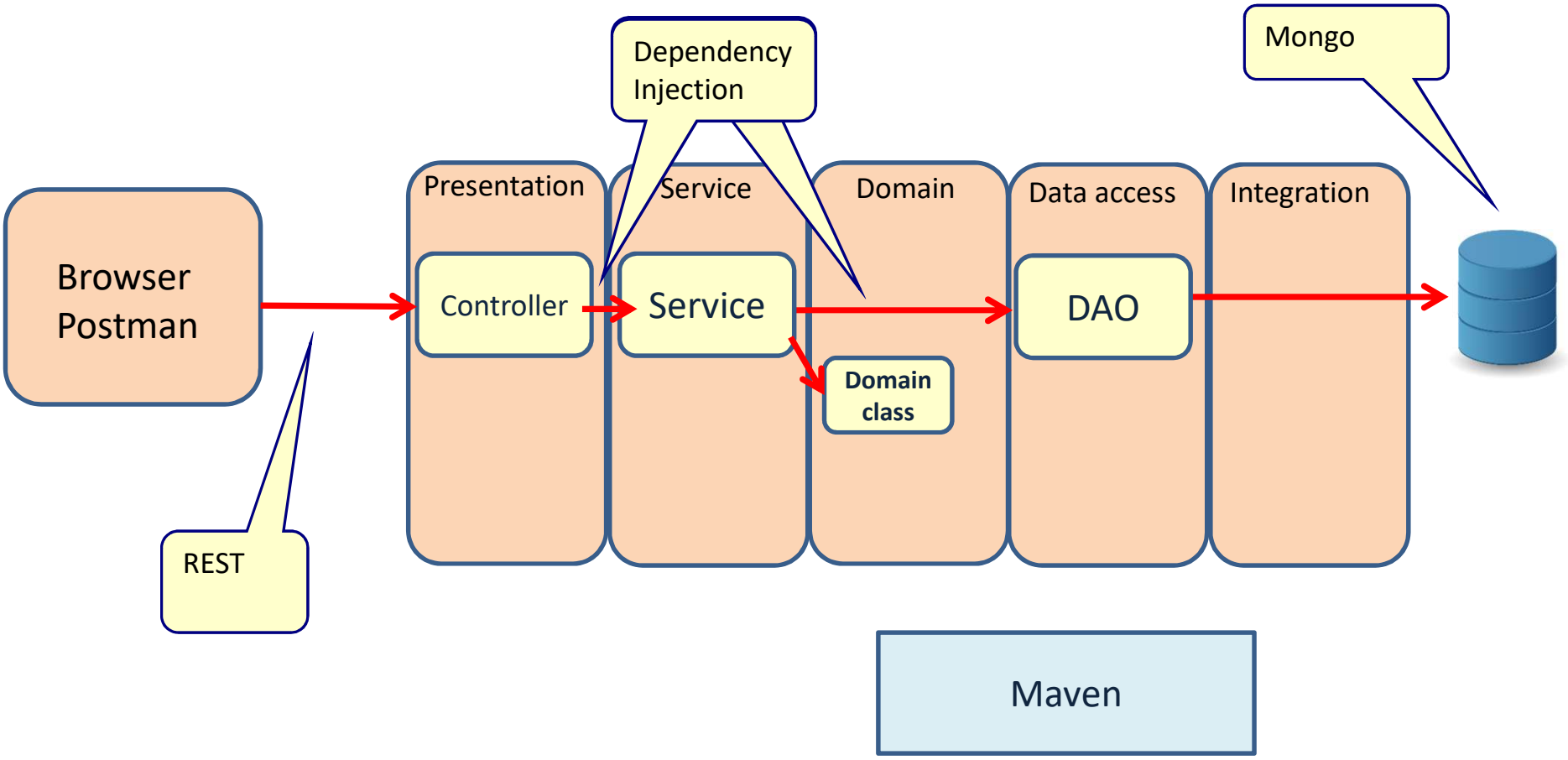
- Containerless deployment

Web container inside App

Creates Jar file from app (which needs just virtual machine to run) rather than War file (which needs web container)

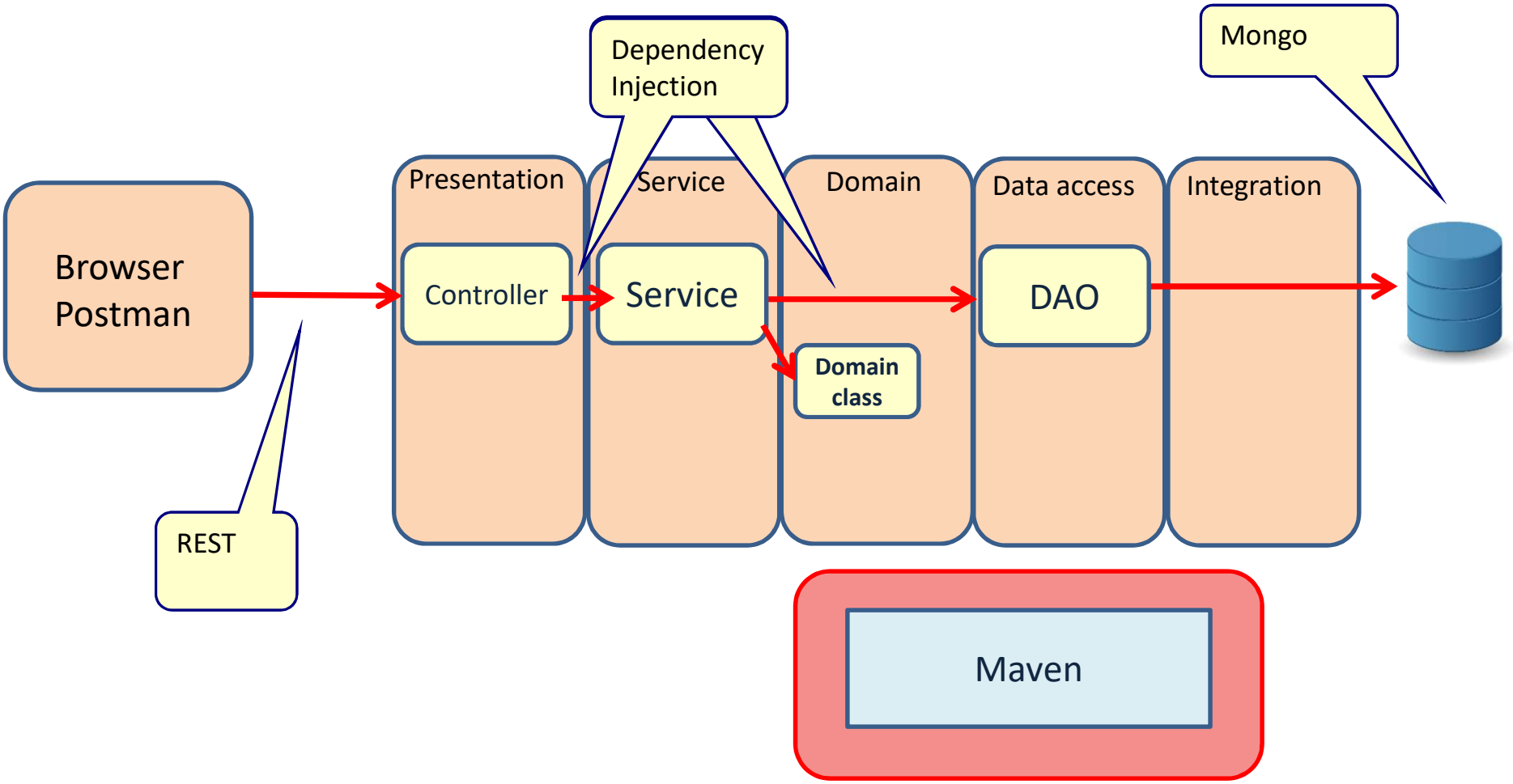


Parts of Spring Boot we will study



MAVEN

Parts of Spring Boot we will study



What is Maven?

- Build tool
- Dependency management tool
- Documentation tool

Libraries

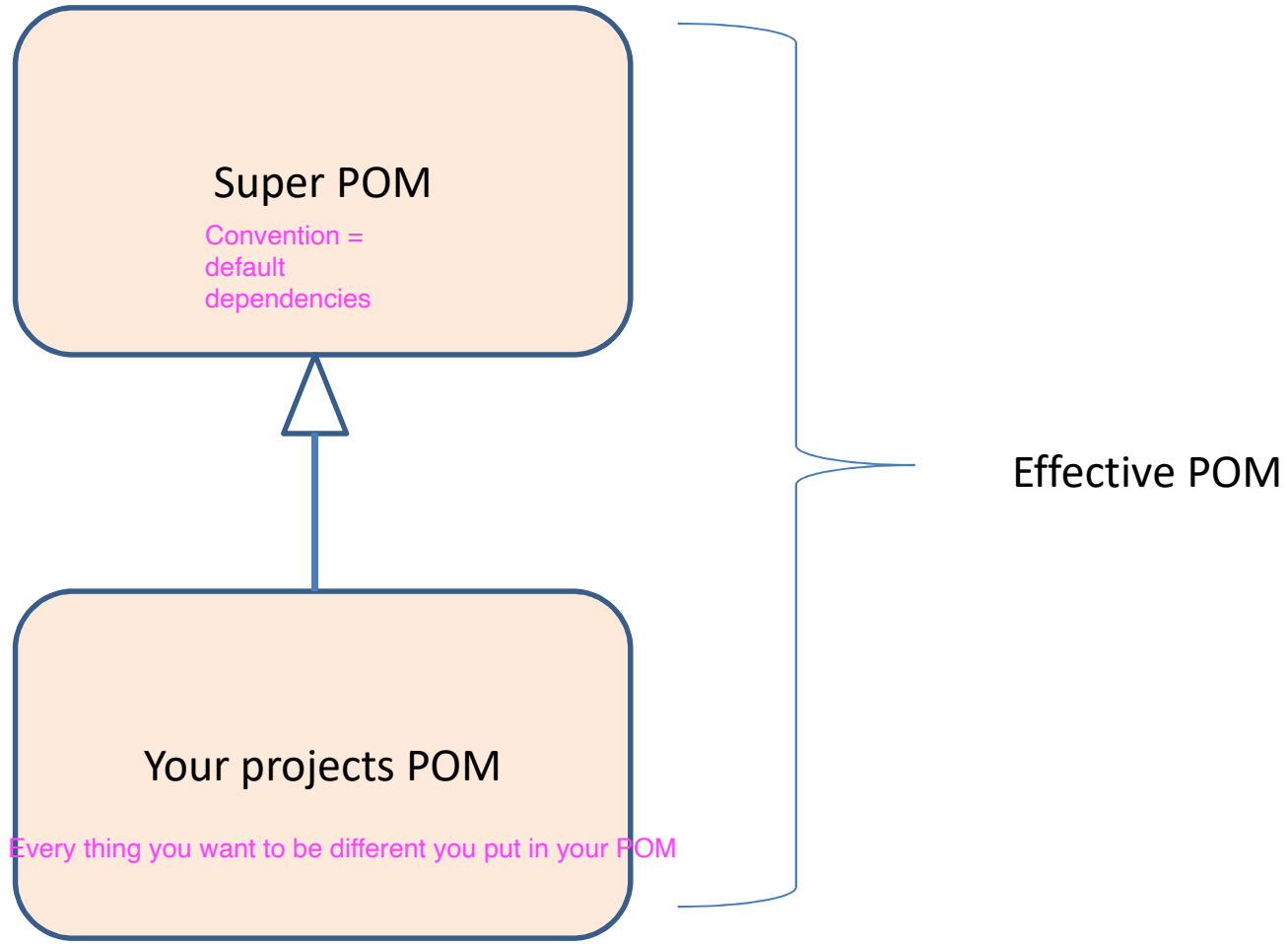
Maven configuration: POM.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/maven-v4_0_0.xsd">

    <!-- PROJECT DESCRIPTION -->
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.myproject</groupId>
    <artifactId>myapp</artifactId>
    <packaging>war</packaging>
    <name>Killer application</name>
    <version>1.0</version>

    <!-- PROJECT DEPENDENCIES -->
    <dependencies>
        <!-- JUnit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1.</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

General project information

This project depends on the JUnit3.8.1.jar file

Super POM

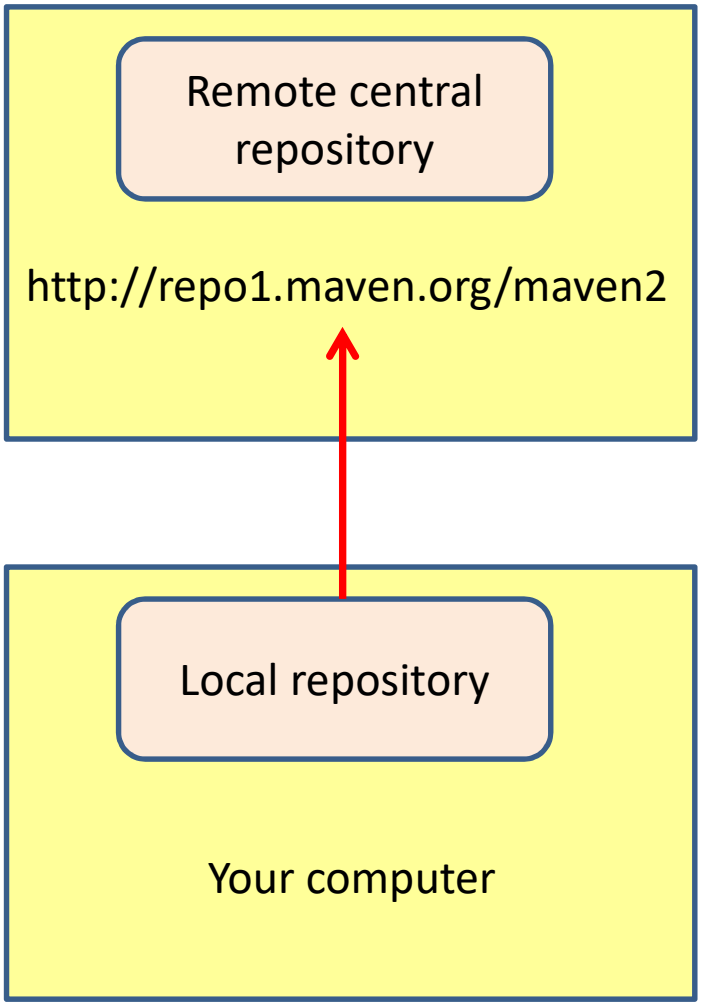


Maven conventions

- A developer familiar with Maven will quickly get familiar with a new project
- No time wasted on re-inventing directory structures

src/main/java	Java source files goes here
src/main/resources	Other resources your application needs
src/main/filters	Resource filters (properties files)
src/main/config	Configuration files
src/main/webapp	Web application directory for a WAR project
src/test/java	Test sources like unit tests (not deployed)
src/test/resources	Test resources (not deployed)
src/test/filters	Test resource filter files (not deployed)
src/site	Files used to generate the Maven project website

Repositories



- If you cannot find the library in your local repository, get it from the remote repository
- Remote repository is defined in the super POM
- Default Local repository is located in USER_HOME/.m2/repository
- Can be configured in settings.xml

Maven dependencies

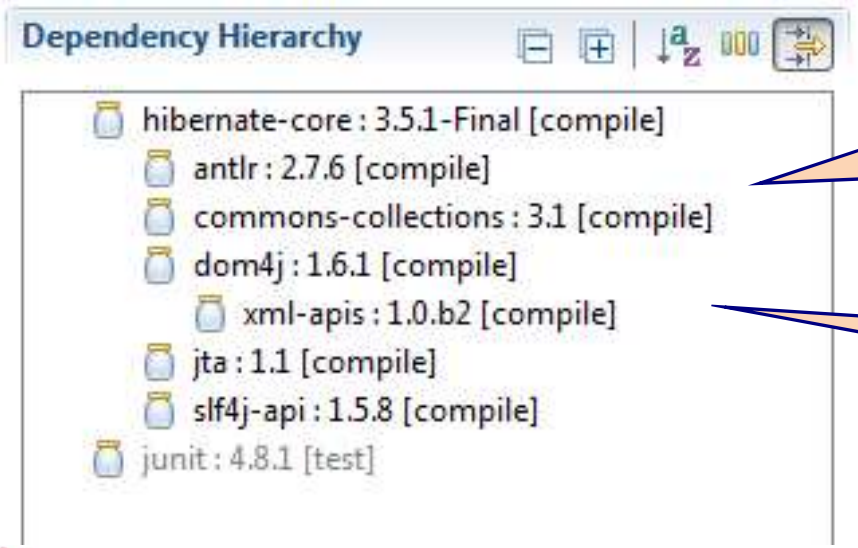
- Define your dependencies in POM.xml
- Maven does the following:
  - Download all defined Jar files in the repository
  - Find all transitive dependencies
  - Add all required Jar files to your project classpath

```
<dependencies>
    <!-- JUnit -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1.</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Transitive dependencies

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>3.5.1-Final</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.1</version>
  </dependency>
</dependencies>
```

Our project depends on 2 jars: hibernate and junit



Hibernate depends on: antlr, commons-collections, dom4j, jta and slf4j-api
dom4j depends on xml-apis

All these jar files are automatically downloaded

BASIC SPRING BOOT APPLICATION

Spring boot POM file

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.M6</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>


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

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

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Inherit Spring Boot default dependencies and versions

Starter POM

Contains goals for packaging the application

Service class

```
public interface ICustomerService {
  void addCustomer(String name, String email);
}
```

```
@Service
public class CustomerService implements ICustomerService {

  public void addCustomer(String name, String email) {
    Customer customer = new Customer(name, email);
    System.out.println(customer.getName()+":"+customer.getEmail());
  }
}
```

```
public class Customer {
  private String name;
  private String email;
...
}
```

Every class with the annotations

- @Service
- @Controller
- @Repository
- @Component

will be instantiated by Spring

Spring Boot Application

Implement the CommandLineRunner

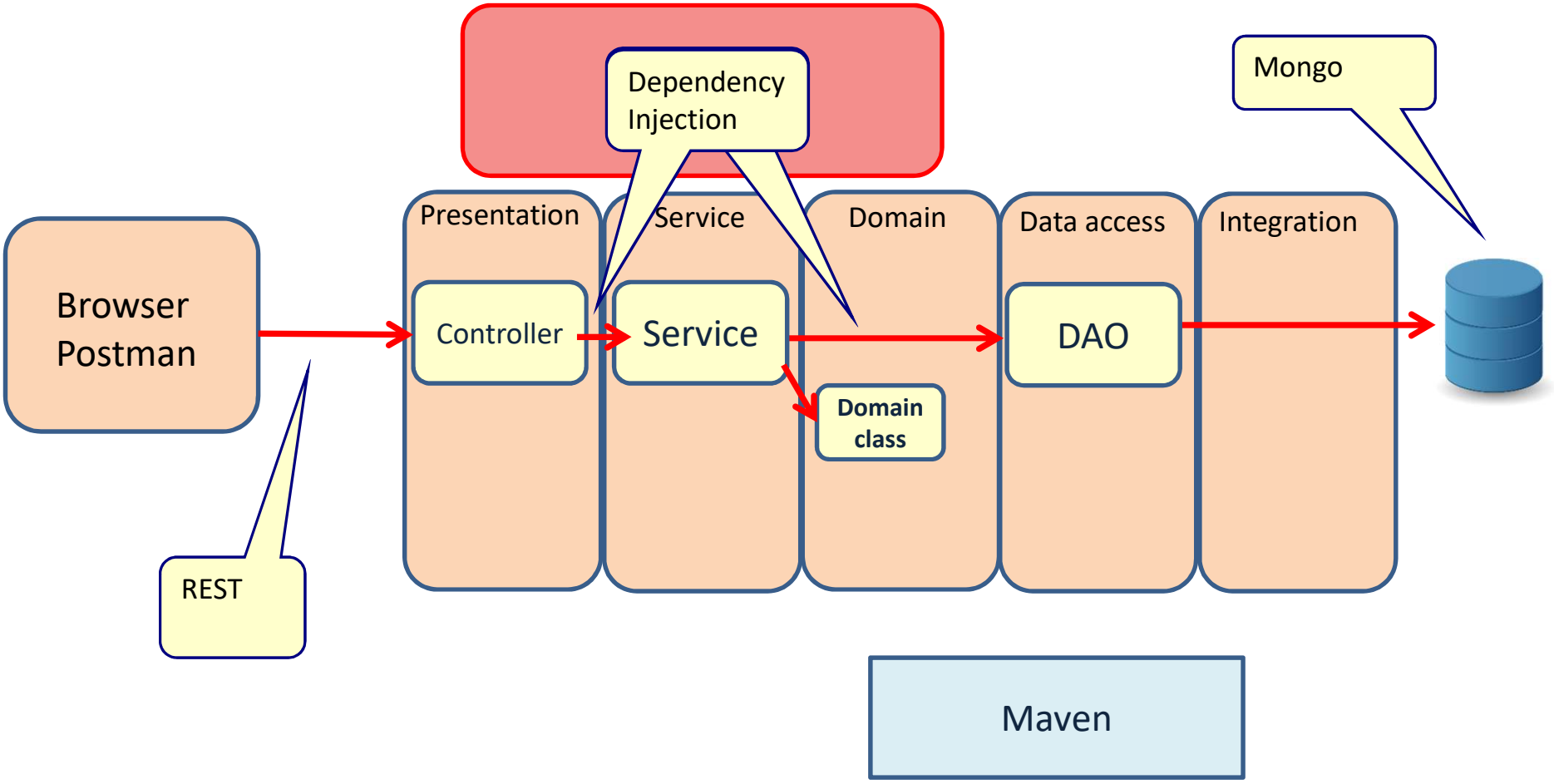
```
@SpringBootApplication
public class SpringBootProjectApplication implements CommandLineRunner {
  @Autowired
  private CustomerService customerService;
  give the reference from object from the context
  public static void main(String[] args) {
    SpringApplication.run(SpringBootProjectApplication.class, args);
  }
        Creates the context and read all the configuration and
        annotation and creates instance of all spring objects

  @Override
  public void run(String... args) throws Exception {
    customerService.addCustomer();
  }
}
```

Inject the customerService

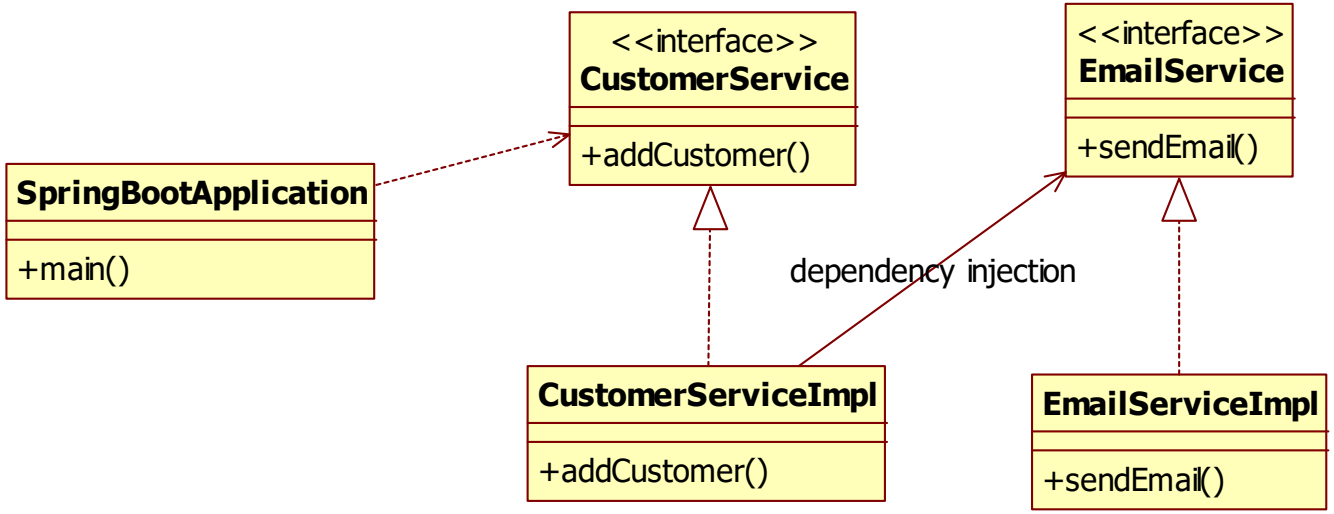
Implement the run() method

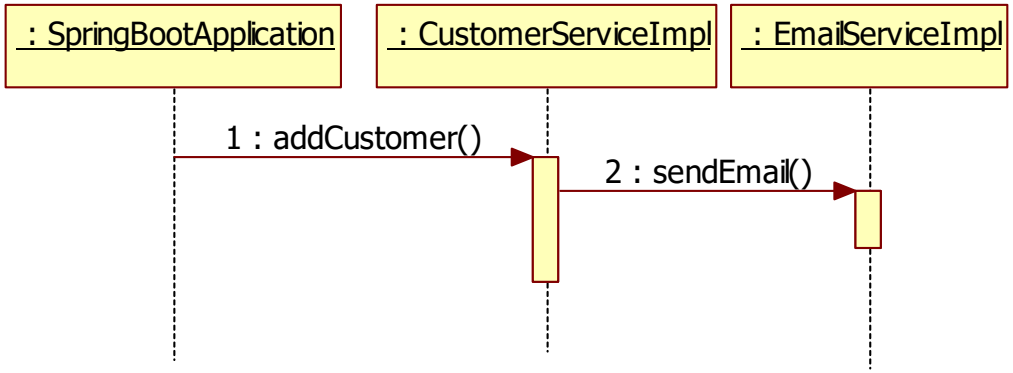
SPRING BOOT & DEPENDENCY INJECTION

Parts of Spring Boot we will study



Dependency injection





Dependency Injection

```
public interface EmailService {
  void sendEmail();
}
```

```
@Service
public class EmailServiceImpl implements EmailService{
  public void sendEmail() {
    System.out.println("Sending email");
  }
}
```

```
public interface CustomerService {
  void addCustomer();
}
```

```
@Service
public class CustomerServiceImpl implements CustomerService{
  @Autowired
  private EmailService emailService;

  public void addCustomer() {
    emailService.sendEmail();
  }
}
```

Inject the EmailService in the CustomerService

Spring Boot Application

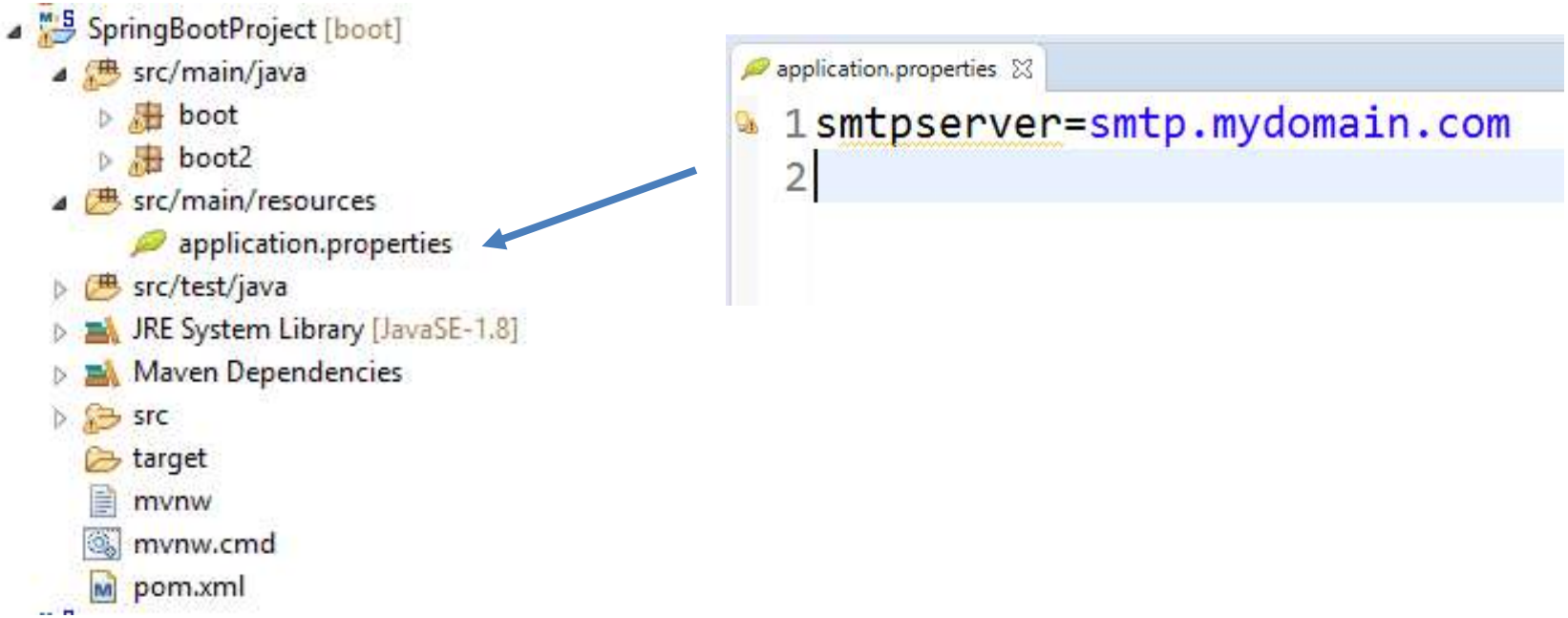
```
@SpringBootApplication
public class SpringBootProjectApplication implements CommandLineRunner {
  @Autowired
  private CustomerService customerService;

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

  @Override
  public void run(String... args) throws Exception {
    customerService.addCustomer();
  }
}
```

Implement the CommandLineRunner

Implement the run() method

Spring Boot configuration

- Spring Boot uses **application.properties** as the default configuration file



application.properties

```java
public interface EmailService {
  void sendEmail();
}
```

```java
@Service
public class EmailServiceImpl implements EmailService{
  @Value(" ${smtpserver}")
  String smtpServer;

  public void sendEmail() {
    System.out.println("Sending email using smtp server "+smtpServer);
  }
}
```

Inject the value from the properties file

application.properties

```
1 smtpserver=smtp.mydomain.com
2
```

Set the logging level in application.properties

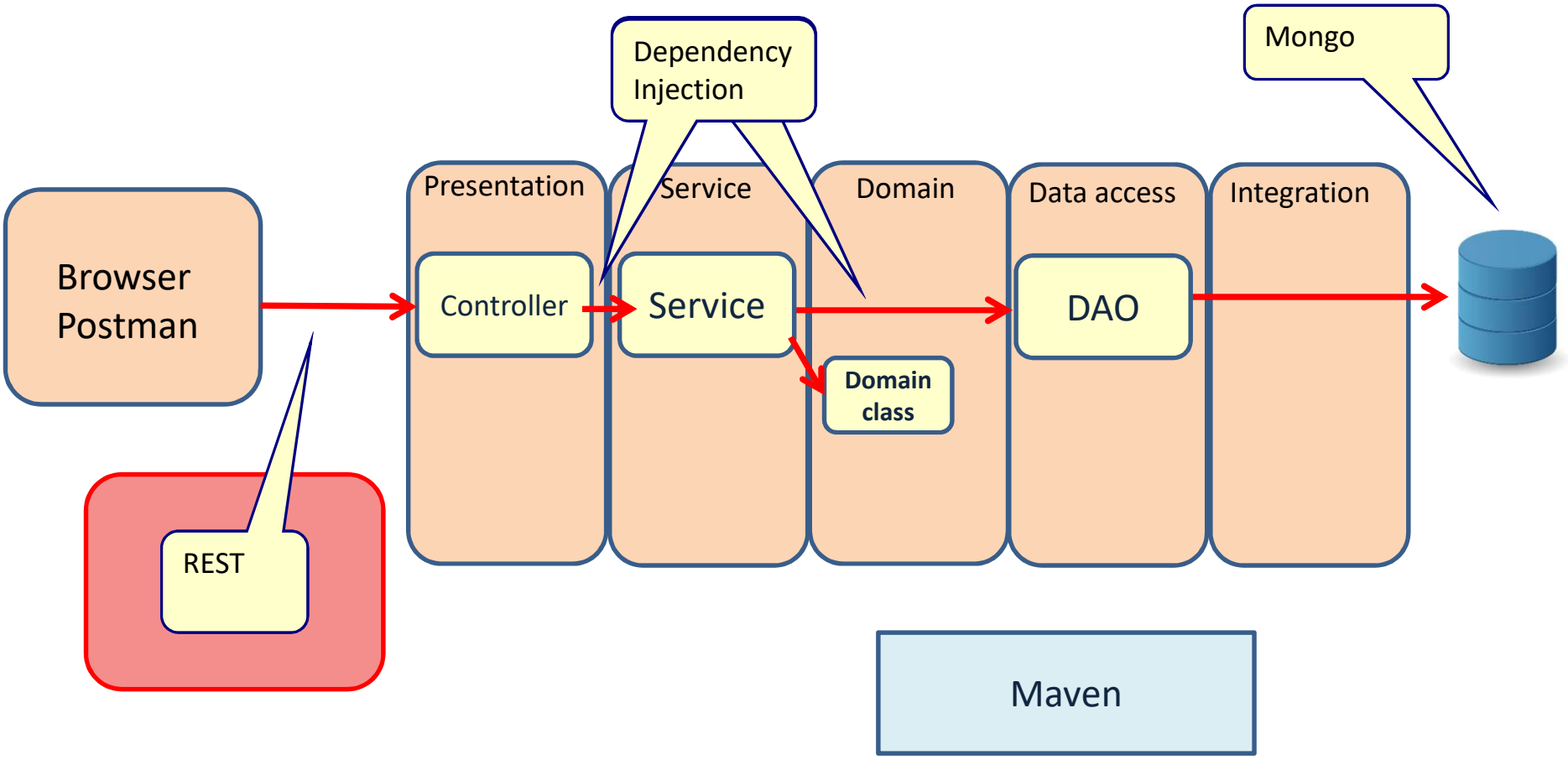
```
logging.level.root=ERROR
logging.level.org.springframework=ERROR
```

Main point

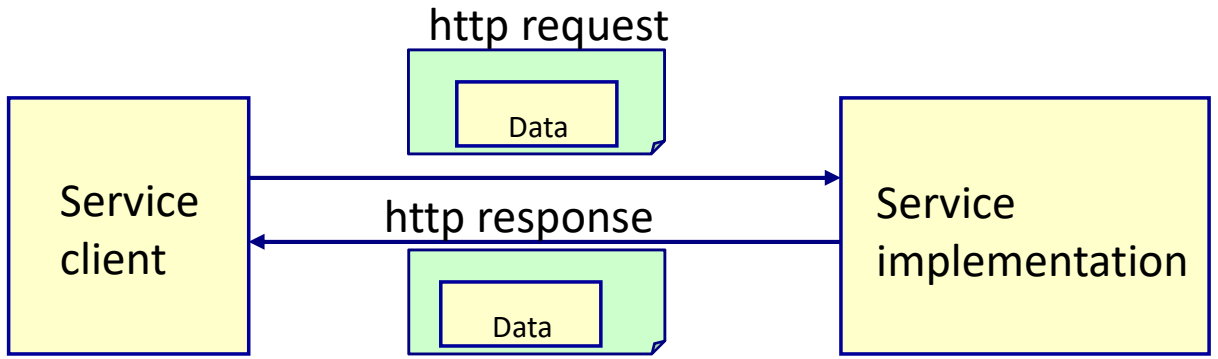
- Dependency injection is a flexible technique to connect objects together by configuration.
- Everything in creation is connected with everything else in its source, the Unified Field, the home of all the laws of nature.

SPRING REST WEBSERVICES

Parts of Spring Boot we will study



RESTful Web Services



- The URL specifies the resource to act on
- Not bound to a specific data format
- Data in HTTP messages
  - GET message for retrieving data
  - POST message for creating data
  - PUT message for updating data
  - DELETE message for deleting data

In service we should create a mapping between httpmethods and service methods

Spring REST libraries

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

Simple Rest Example: the controller

@RestController tells Spring that this class is a controller that is called by sending HTTP REST requests, and that returns HTTP response messages

```
@RestController
public class GreetingController {

    @RequestMapping("/greeting")
    public String greeting() {
        return "Hello World";
    }
}
```

The URL to call this method ends with **/greeting**

Simple Rest Example: configuration

```
@SpringBootApplication
public class GreetingRestApplication {

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

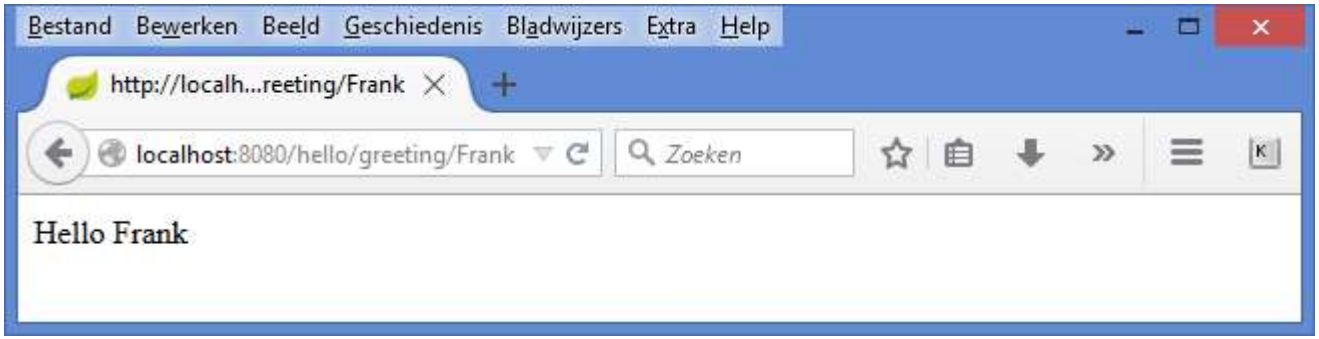
Simple Rest Example: calling the service

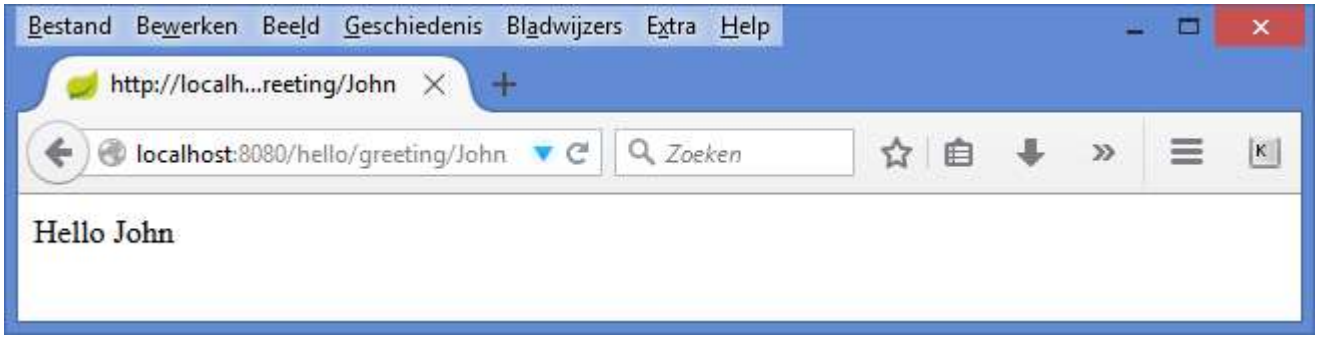

```
@RestController
public class GreetingController {

    @RequestMapping("/greeting")
    public String greeting() {
        return "Hello World";
    }
}
```

Different URL


```
@RestController
@RequestMapping("/hello")
public class GreetingController {

    @RequestMapping(value="/greeting")
    public  String greetingJSON() {
        return "Hello World";
    }
}
```

Path variables






```
@RestController
@RequestMapping("/hello")
public class GreetingController {

    @RequestMapping(value="/greeting/{name}")
    public  String greeting(@PathVariable String name) {
        return "Hello "+name;
    }
}
```

normally classes return string

Returning a class


```
@RestController
public class GreetingController {

    @RequestMapping("/greeting")
    public Greeting greeting() {
        return new Greeting("Hello World");
    }
}
```

Serialize the class to JSON

Return a Greeting class

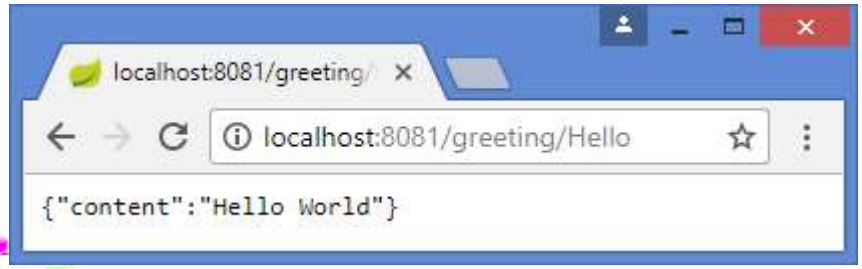
```
public class Greeting {
  private final String content;

  public Greeting(String content) {
    this.content = content;
  }

  public String getContent() {
    return content;
  }
}
```

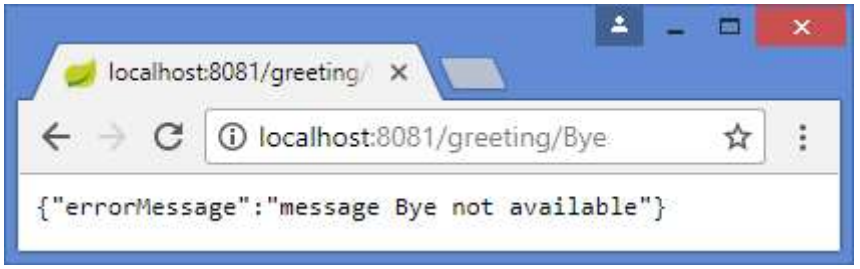
ResponseEntity

```
@RestController
public class GreetingController {

    @RequestMapping("/greeting/{message}")
    public ResponseEntity<?> getGreeting(@PathVariable("message") String message) {
      Greeting greeting = new Greeting("");
      if (message.equals("Hello")) {
        greeting.setContent("Hello World");
      }
      else{
        return new ResponseEntity<CustomErrorType>(new CustomErrorType("message " + message+
          " not available"), HttpStatus.NOT_FOUND);
      }
        return new ResponseEntity<Greeting>(greeting, HttpStatus.OK);
    }
}
```

Create responseEntity of the same type,
when we want to have control over header and the response type

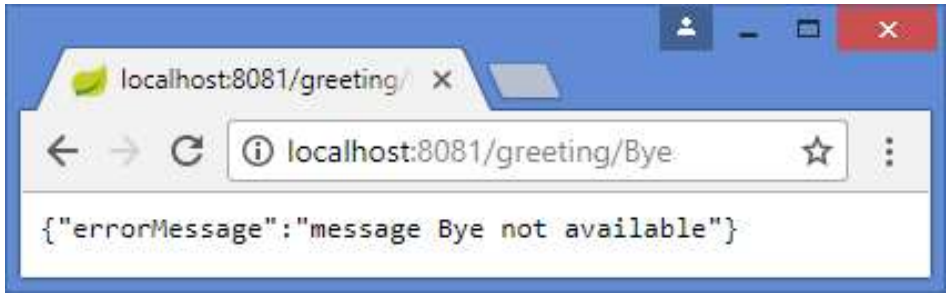
Set the content and the HttpStatus





CustomErrorType

```
public class CustomErrorType {
  private String errorMessage;

  public CustomErrorType(String errorMessage) {
    this.errorMessage = errorMessage;
  }

  public String getErrorMessage() {
    return errorMessage;
  }
}
```

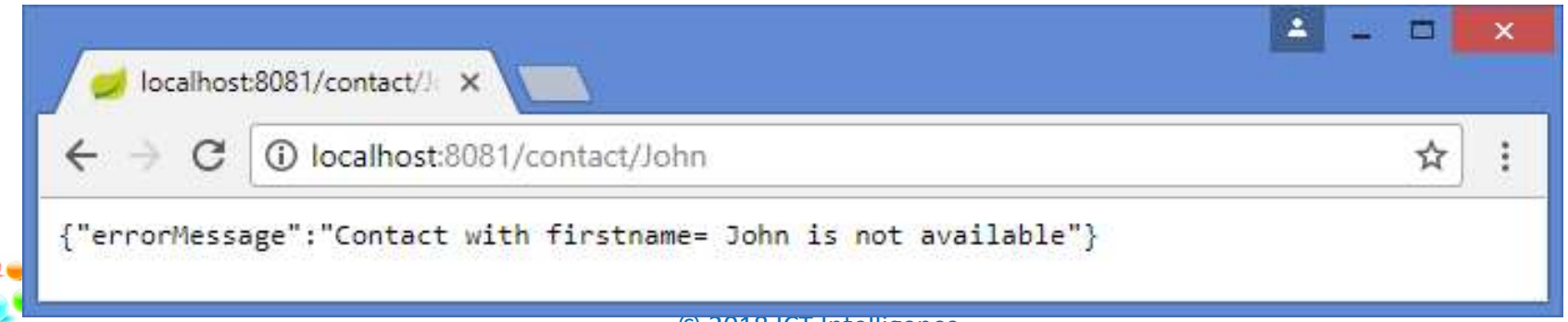


ContactController

```
@RestController
public class ContactController {

  private Map<String, Contact> contacts = new HashMap<String, Contact>();

  public ContactController() {
    contacts.put("Frank", new Contact("Frank", "Brown", "fbrown@acme.com", "2341678453"));
    contacts.put("Mary", new Contact("Mary", "Jones", "mjones@acme.com", "2341674376"));
  }



  @RequestMapping("/contact/{firstName}")
  public ResponseEntity<?> getContact(@PathVariable String firstName) {
    Contact contact = contacts.get(firstName);
    if (contact == null) {
      return new ResponseEntity<CustomErrorType>(new CustomErrorType("Contact with firstname= "
        + firstName + " is not available"),HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<Contact>(contact, HttpStatus.OK);
  }
}
```

@PathVariable: Get the value of the variable

ContactController



Add a contact

```
@RestController
public class ContactController {
  private Map<String, Contact> contacts = new HashMap<String, Contact>();

  public ContactController() {
    contacts.put("Frank", new Contact("Frank", "Brown", "fbrown@acme.com", "2341678453"));
    contacts.put("Mary", new Contact("Mary", "Jones", "mjones@acme.com", "2341674376"));
  }

  @RequestMapping("/contact/{firstName}")
  public ResponseEntity<?> getContact(@PathVariable String firstName) {
    Contact contact = contacts.get(firstName);
    if (contact == null) {
      return new ResponseEntity<CustomErrorType>(new CustomErrorType("Contact with
          firstname= " + firstName + " is not available"),HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<Contact>(contact, HttpStatus.OK);
  }

  @RequestMapping(value="/contact", method=RequestMethod.POST)
  public ResponseEntity<?> addContact(@RequestBody Contact contact) {
    contacts.put(contact.getFirstName(), contact);
    return new ResponseEntity<Contact>(contact, HttpStatus.OK);
  }

}
```

GET request is default

POST request

Get the Contact class from the HTTP request message

Completing the Contact server application

```
@RestController
public class ContactController {

...

    @RequestMapping(value="/contact/{firstName}", method=RequestMethod.DELETE)
    public ResponseEntity<?> deleteContact(@PathVariable String firstName) {
      Contact contact = contacts.get(firstName);
      if (contact == null) {
        return new ResponseEntity<CustomErrorType>(new CustomErrorType("Contact with
           firstname= " + firstName + " is not available"),HttpStatus.NOT_FOUND);
      }
      contacts.remove(firstName);
      return new ResponseEntity<Contact>(HttpStatus.NO_CONTENT);
    }


    @RequestMapping(value="/contact", method=RequestMethod.PUT)
    public ResponseEntity<?> updateContact(@RequestBody Contact contact) {
      contacts.put(contact.getFirstName(), contact);
      return new ResponseEntity<Contact>(contact, HttpStatus.OK);
    }
}
```

DELETE request

Response is empty

PUT request

Get all contacts

```
@RequestMapping(value="/contact/all", method=RequestMethod.GET)
public ResponseEntity<?> getAllContacts() {
    return new ResponseEntity<Collection<Contact>>(contacts.values(), HttpStatus.OK);
}
```

Mapping annotations

```
@RequestMapping(value = "/add", method = RequestMethod.GET)
```

```
@GetMapping("/add")
```

Same

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
```

```
@PostMapping("/add")
```

Same

```
@RequestMapping(value = "/del", method = RequestMethod.DELETE)
```

```
@DeleteMapping("/del")
```

Same

```
@RequestMapping(value = "/mod", method = RequestMethod.PUT)
```

```
@PutMapping("/mod")
```

Same

Containerless deployment

Container Deployments

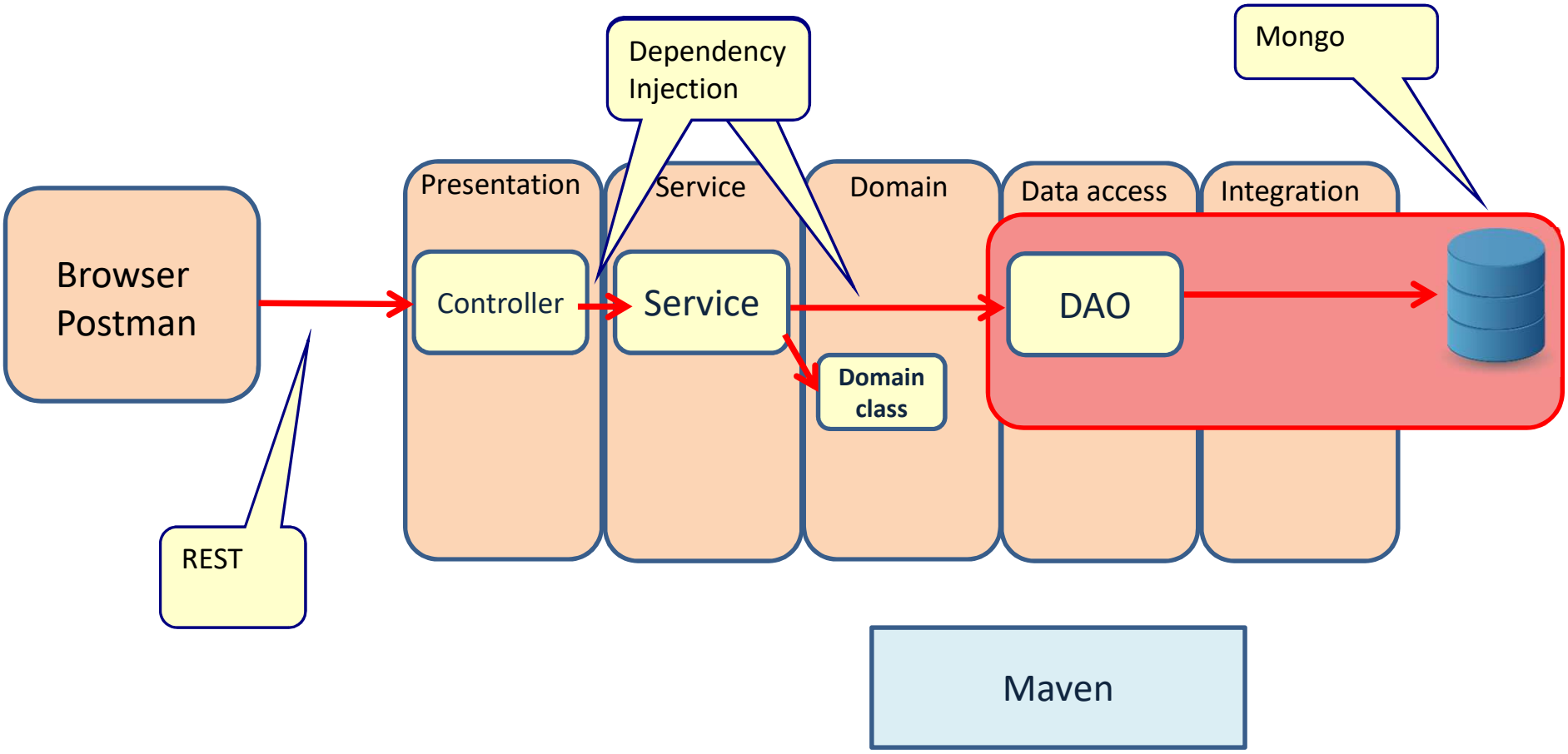
- Pre-setup and configuration
- Need to use files like web.xml to tell container how to work
- Environment configuration is external to your application

Application Deployments

- Runs anywhere Java is setup (think cloud deployments)
- Container is embedded and the app directs how the container works
- Environment configuration is internal to your application

SPRING MONGO

Parts of Spring Boot we will study



Spring Mongo libraries

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

The Mongo Documents

```
@Document    = @Entity for relational DB
public class Student {

    @Id
    private String studentId;
    private String name;
    List<Course> courses = new ArrayList();

    public void addCourse(Course course) {
      courses.add(course);
    }

    public void print() {
      System.out.println( "Student{" + "studentId=" + studentId
      + ", name=" + name + ", [" );
      for (Course course : courses) {
        course.print();
        System.out.print(" ,");
      }
      System.out.println( "]}");
    }
    ...
}
```

```
@Document
public class Course {
  @Id
  private String courseId;
  private String name;

  public void print() {
    System.out.println( "Course{" +
    "courseId=" + courseId + ", name="
    + name + "}" );
  }
     ...
}
```

The repository = Data Access Object (DAO)

```
public interface StudentRepository extends MongoRepository<Student, String> {

}
```

application.properties

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=testdb
```

The application

```
@SpringBootApplication
public class SpringDataMongoApplication implements CommandLineRunner {
  @Autowired
  private StudentRepository studentRepository;

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

  @Override
  public void run(String... args) throws Exception {
    Student student1 = new Student("11", "Frank Brown");
    Student student2 = new Student("14", "John Doe");
    Course course1 = new Course("CS101", "Programming 1");
    Course course2 = new Course("CS108", "Algorithms");
    Course course3 = new Course("CS450", "Computer Networks");
    Course course4 = new Course("CS545", "Software Architecture");
    student1.addCourse(course1);
    student1.addCourse(course2);
    student2.addCourse(course1);
    student2.addCourse(course3);
    student2.addCourse(course4);
    studentRepository.save(student1);
    studentRepository.save(student2);
    studentRepository.findAll().forEach((course) -> course.print());
  }
}
```

Queries

```
public interface StudentRepository extends MongoRepository<Student, String> {
  Student findByName(String name);
}
```

Just by name of method it creates the query findBy+propertyName

Is called query method

Define your own method based on a standard convention

default Method

```
Optional<Student> studentOpt = studentRepository.findById("11");
Student student11 = studentOpt.get();
System.out.println(student11.getName());
```

findById is a standard method on a repository

```
Student student14 = studentRepository.findByName("John Doe");
System.out.println(student14.getStudentId());
```

Use the new method

Query methods rules

- The name of our query method must start with one of the following prefixes:
  - find…By, read…By, query…By, count…By, and get…By.
- If we want to specify the selected property, we must add the name of the property before the first By word.
  - findTitleBy
- If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word.
  - If we want to get more than one result, we have to append the optional numeric value to the First and the Top keywords.
  - findTopBy, findTop1By, findFirstBy, findFirst2By
- If we want to select unique results, we have to add the Distinct keyword before the first By word.
  - findTitleDistinctBy, findDistinctTitleBy
- We must add the search criteria of our query method after the first By word.
  - findByEmailAddressAndLastname
- If our query method specifies x search conditions, we must add x method parameters to it.
  - The number of method parameters must be equal than the number of search conditions.
  - The method parameters must be given in the same order than the search conditions.

Supported keywords

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

@Query

```
public interface StudentRepository extends MongoRepository<Student, String> {
  Student findByName(String name);
          name of the attr in Student class
  @Query("{'courses.name': ?0}")
  List<Student> findByCourseName(final String courseName);
}
```

?0 is the first parameter

```
List<Student> students = studentRepository.findByCourseName("Programming 1");
students.forEach(student -> System.out.println(student.getName()));
```

Main point

- In Spring Boot, a DAO (repository) is nothing more than a simple interface. Spring will create the implementation of the interface.

- Do less and accomplish more.

desire is like interface

Connecting the parts of knowledge with the wholeness of knowledge

1. Spring Boot is an opinionated framework that uses convention over configuration.
2. Spring Boot makes it as simple as possible to write an enterprise Java application.

3. **Transcendental consciousness** is the field of all knowledge.
4. **Wholeness moving within itself:** In Unity Consciousness, all of the intelligence and structure at the basis of the universe is realized as the lively qualities of one's own inner intelligence.