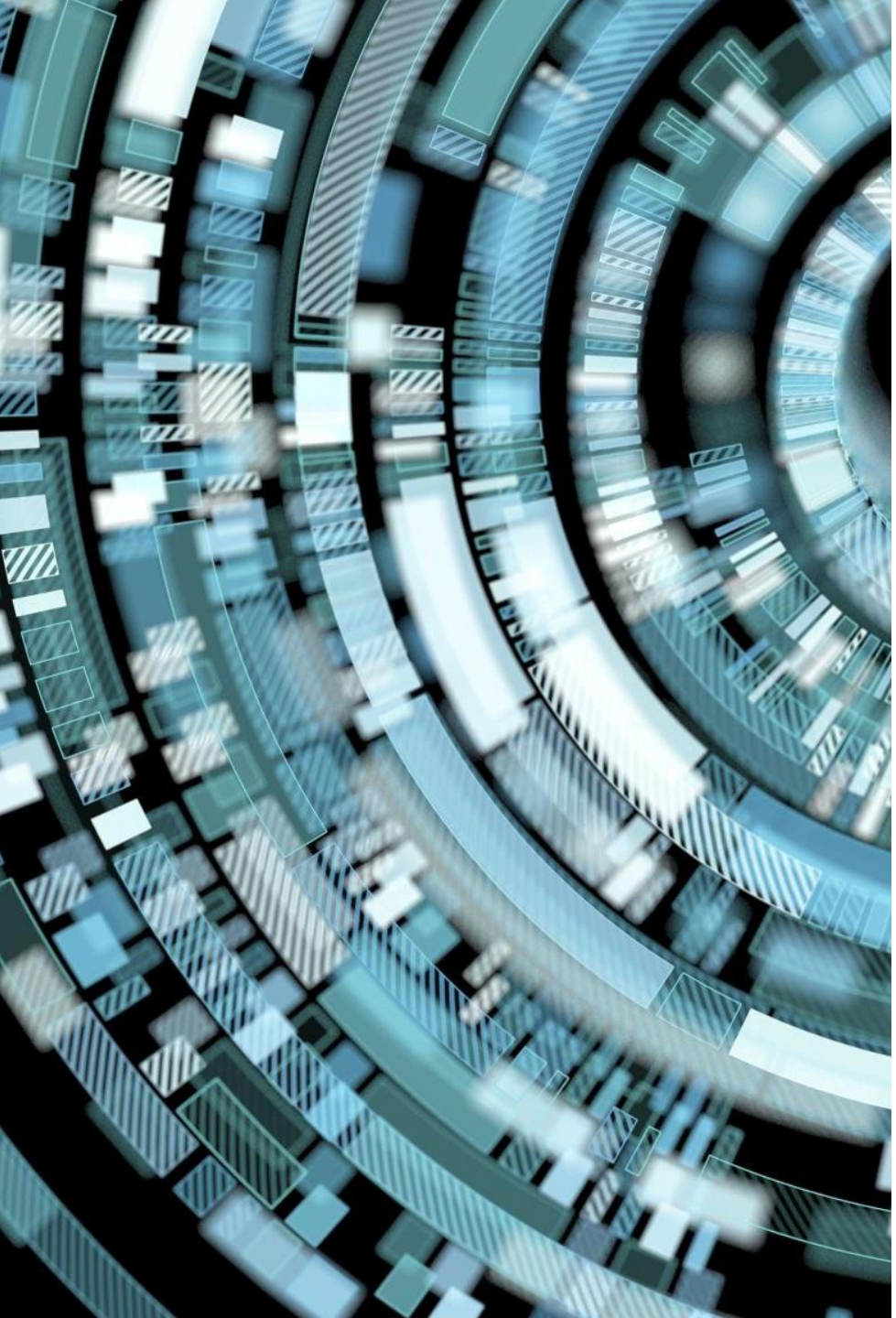
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 12

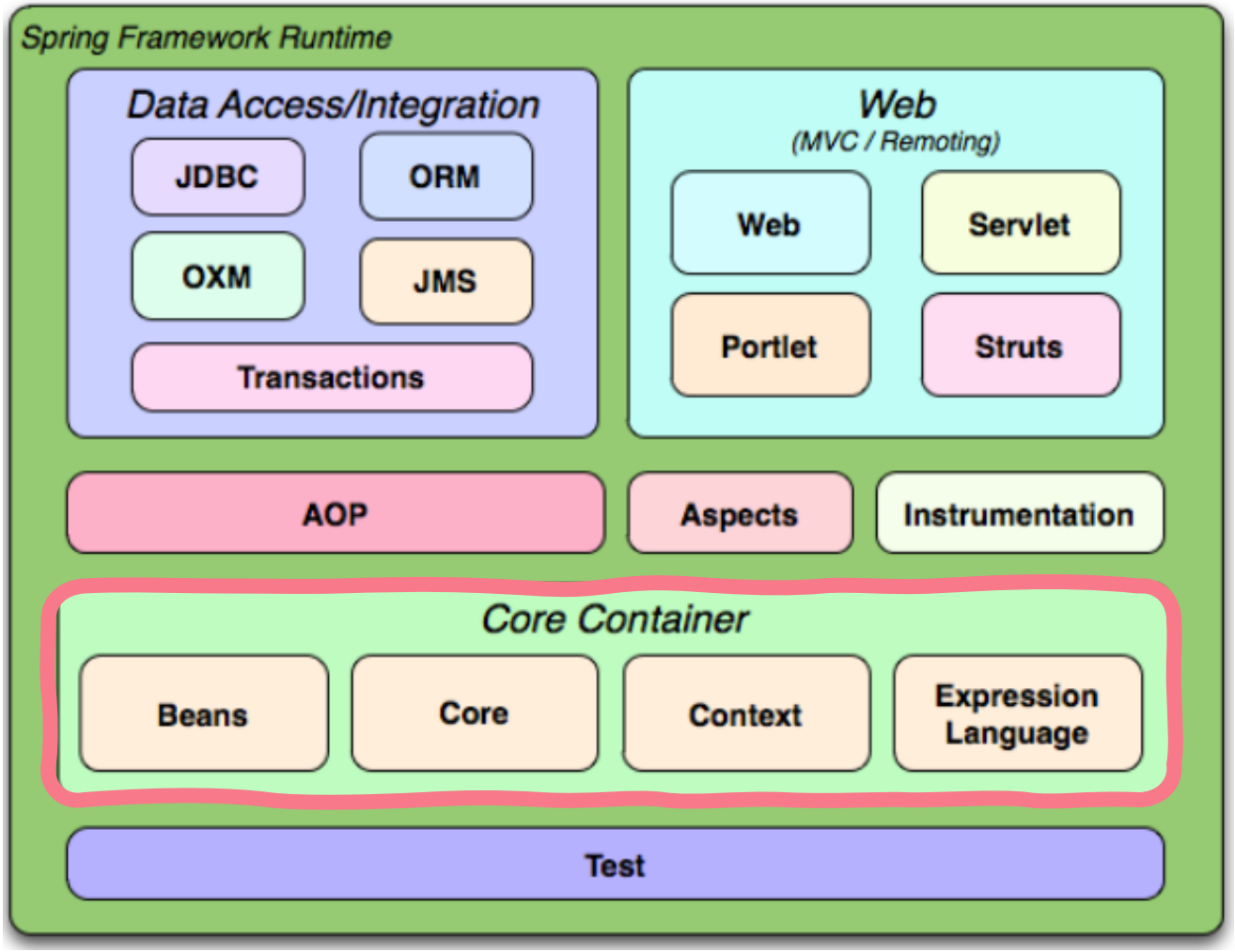
- The Spring Framework
  - IoC & Dependency Injection
  - Spring AOP
  - Spring MVC
- Spring Boot
  - Overview
  - Building a MVC web application
  - Building a RESTful web service
  - Microservices

The Spring Framework

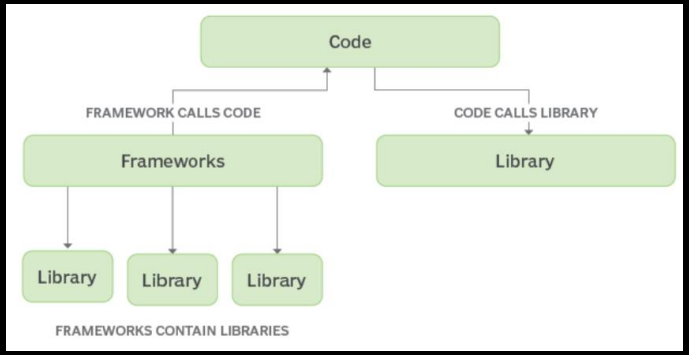
- The Spring Framework is an open-source, lightweight framework that enables developers to develop enterprise-class applications using Plain Old Java Object (POJO), instead of EJB
- It also offers tons of extensions that are used for building all sorts of large-scale applications on top of the Java EE platform

The Spring Framework

- The Spring Framework consists of features organized into about 20 modules, as shown in the diagram
- Spring Core Container is required, other modules are optional
- Core Container is based on IoC and Dependency Injection



Core Concepts in Spring



- Inversion of Control (IoC, 控制反转): a principle in SE which transfers the control of objects or portions of a program to a container or framework
- Traditionally, our custom code makes calls to a library; In contrast, IoC enables a framework to take control of the flow of a program and make calls to our custom code.
- To use a framework, you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

Core Concepts in Spring

- Dependency Injection (DI, 依赖注入): how IoC concept is implemented in Spring.
- The Spring IoC container takes control by injecting necessary dependencies
- Dependency injection makes a class independent of its dependencies. It achieves that by decoupling (解耦) the usage of an object from its creation
- DI aims to separate the concerns of constructing objects (by IoC container) and using them (by callers), leading to loosely coupled programs.

Dependency Injection

```
class Car{
  private Wheel wh = new NepaliRubberWheel();
  private Battery bt = new ExcideBattery();

  //The rest
}
```

Without DI:

- The Car object is responsible for creating the dependent objects Wheel and Battery.
- The code is highly coupled (Car breaks if Battery's constructor changes)
- Hard to test (how to test Car?)

```
class Car{
  private Wheel wh; // Inject an Instance of Wheel (dependency of car) at runtime
  private Battery bt; // Inject an Instance of Battery (dependency of car) at runtime
  Car(Wheel wh,Battery bt) {
      this.wh = wh;
      this.bt = bt;
  }
  //Or we can have setters
  void setWheel(Wheel wh) {
      this.wh = wh;
  }
}
```

With DI:

- Injecting the dependencies (Wheel and Battery) at runtime.
- DI can be done by setter injection or constructor injection.

https://stackoverflow.com/a/6085922

Dependency Injection

"Dependency Injection" is a 25-dollar term for a 5-cent concept. [...] Dependency injection means giving an object its instance variables.

- James Shore

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself.

- Martin Fowler

https://www.jamesshore.com/v2/blog/2006/dependency-injection-demystified
https://martinfowler.com/articles/injection.html

```
class Car{
  private Wheel wh; // Inject an Instance of Wheel (dependency of car) at runtime
  private Battery bt; // Inject an Instance of Battery (dependency of car) at runtime
  Car(Wheel wh,Battery bt) {
      this.wh = wh;
      this.bt = bt;
  }
  //Or we can have setters
  void setWheel(Wheel wh) {
      this.wh = wh;
  }
}
```

With DI:

- Injecting the dependencies (Wheel and Battery) at runtime.
- DI can be done by setter injection or constructor injection.

Example

```java
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        System.out.println("Driving with " + engine.getName() + " engine.");
    }
}
```

```java
public class Engine {
    private String name;

    public Engine() {}

    public Engine(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void start() {
        System.out.println(name + " engine is starting.");
    }
}
```

- Business logics: `Car` and `Engine` (POJOs)
- `Car` depends on `Engine`

@Component

- `@Component` is used for automatic bean detection
- Without having to write any code, Spring container will:
  - Scan our application for classes annotated with `@Component`
  - Instantiate them and inject any specified dependencies into them
  - Inject them wherever needed

```java
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        System.out.println("Driving with " + engine.getName() + " engine.");
    }
}
```

https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html

@Autowired

- `@Autowired` can be applied on setter methods and constructors.
- The `@Autowired` annotation injects object dependency implicitly.
- Autowiring allows the Spring container to automatically resolve dependencies between collaborating beans by inspecting the beans that have been configured

```java
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void drive() {
        System.out.println("Driving with " + engine.getName() + " engine.");
    }
}
```

https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html

Configurations

- @Configuration
  - A Java class annotated with @Configuration is a configuration by itself
  - Classes with @Configuration define and instantiate beans

- @ComponentScan
  - We use the @ComponentScan annotation along with the @Configuration annotation to specify the packages that we want to be scanned

```java
public class Engine {
    private String name;

    public Engine() {}

    public Engine(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void start() {
        System.out.println(name + " engine is starting.");
    }
}
```

```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

    @Bean
    public Engine engine() {
        return new Engine("V8");
    }

}
```

@Bean

- A bean is an object that is instantiated, assembled, and managed by a Spring IoC container
- @Bean annotation works with @Configuration to create Spring beans
- Methods annotated with @Bean create and return the actual bean

```java
public class Engine {
    private String name;

    public Engine() {}

    public Engine(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void start() {
        System.out.println(name + " engine is starting.");
    }
}
```

```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

    @Bean
    public Engine engine() {
        return new Engine("V8");
    }

}
```

Spring IoC Container

- Spring IoC container is responsible for instantiating, configuring and assembling objects/beans (using DI), as well as managing their life cycles (hence *the inversion of control*).
- The `ApplicationContext` interface is the commonly used Spring IoC Container
- Your application classes are combined with configuration metadata so that after the `ApplicationContext` is created and initialized, you have a fully configured and executable system or application.

```java
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        Car car = context.getBean(Car.class);
        car.drive();
    }
}
```

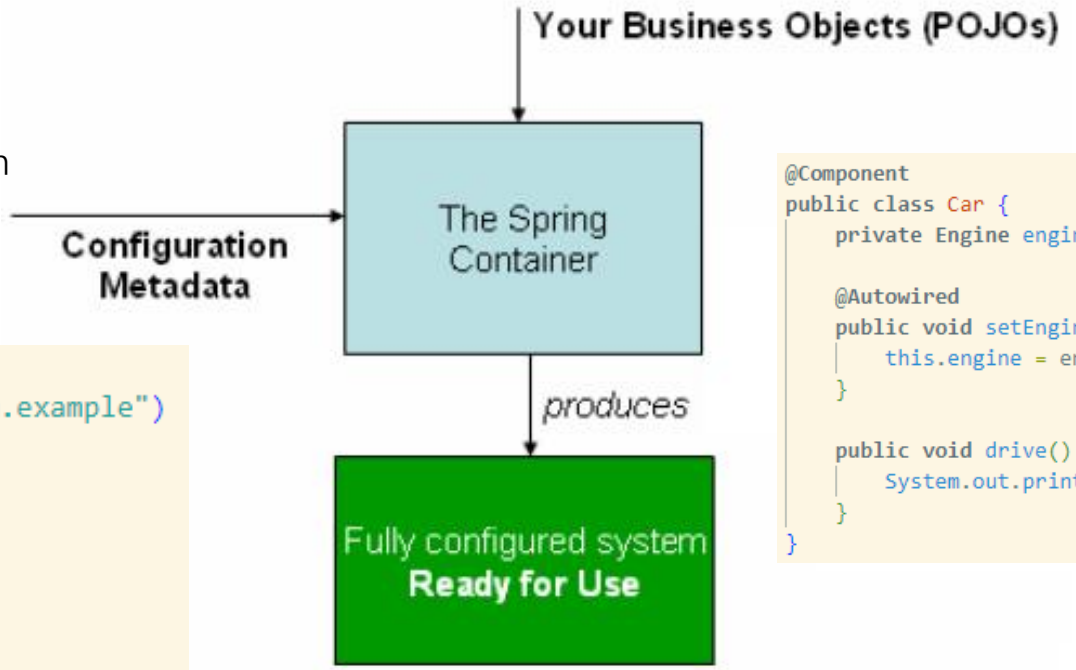
To Put it Together

Spring container will:

- Scan our application for classes annotated with @Component
- Instantiate them by injecting any specified dependencies into them at runtime

Configuration could be:

- XML-based configuration
- Annotation-based configuration




```java
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {

    @Bean
    public Engine engine() {
        return new Engine("V8");
    }

}
```

```java
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine)
        this.engine = engine;
    }

    public void drive() {
        System.out.println("Driving with " + engine.getName() + " engine.");
    }
}
```

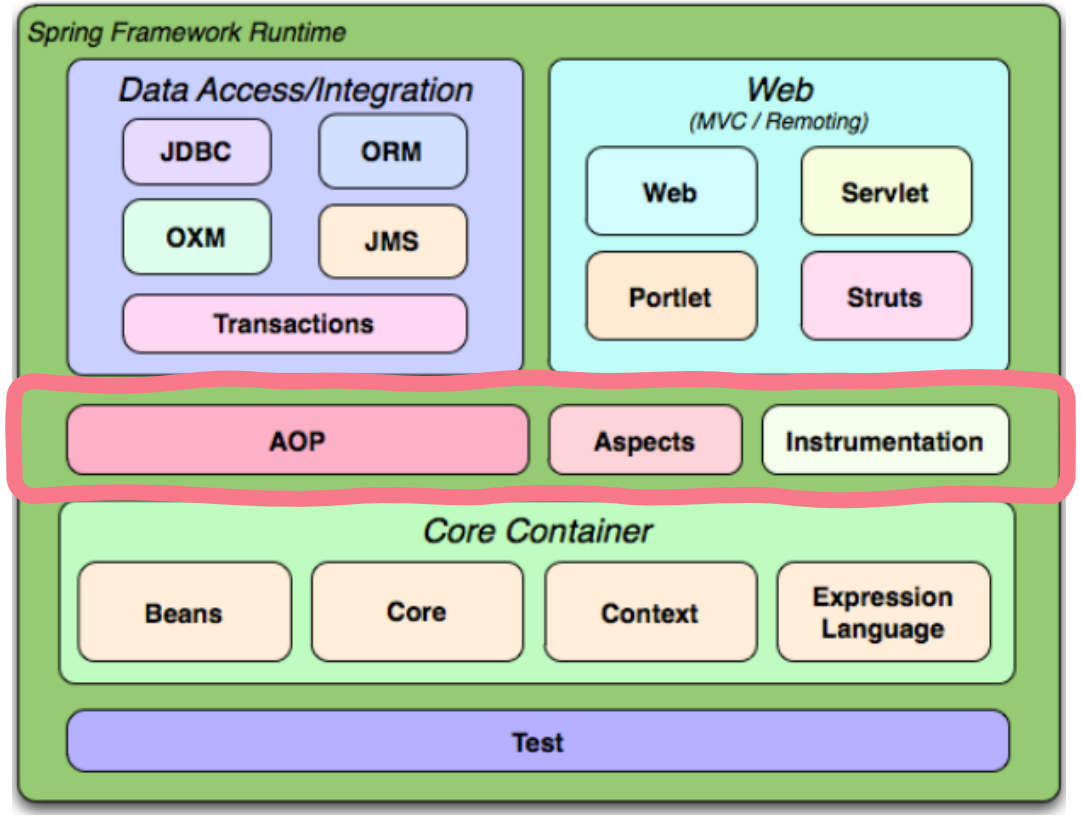
```java
public class Engine {
    private String name;

    public Engine() {}

    public Engine(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void start() {
        System.out.println(name + " engine is starting.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        Car car = context.getBean(Car.class);
        car.drive();
    }
}
```

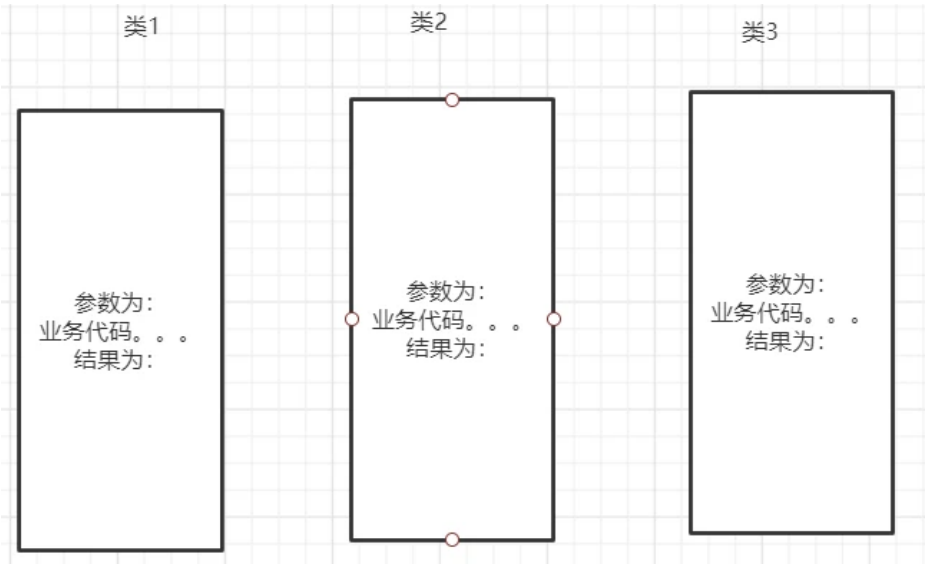
Spring AOP

- AOP (Aspect-Oriented Programming, 面向切面编程): a programming paradigm that complements OOP by allowing the separation of cross-cutting concerns (i.e., we could add additional cross-cutting behavior to existing code without modifying the code itself)

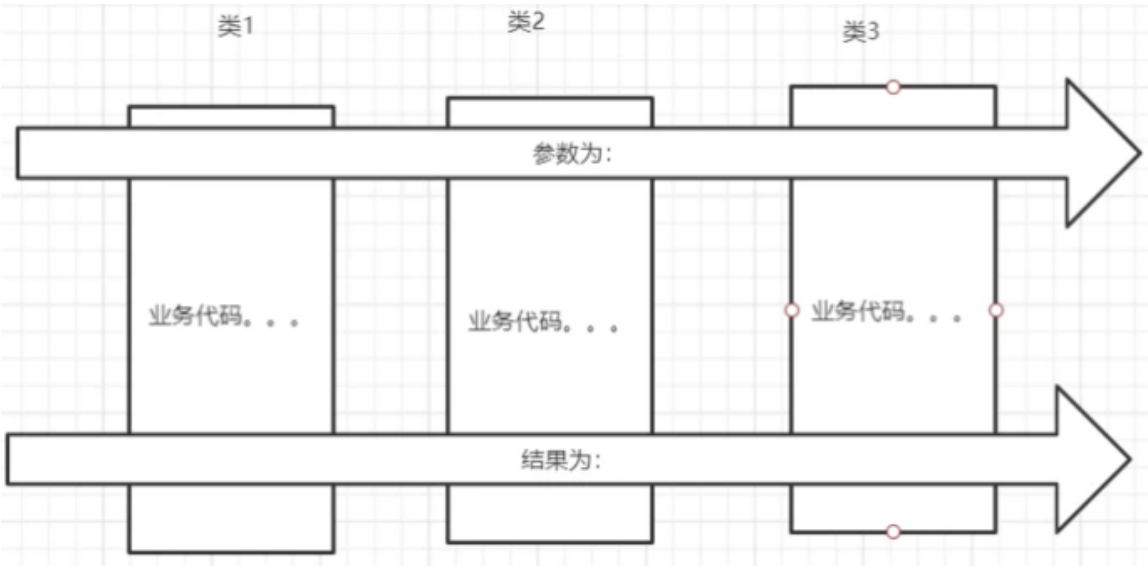
- Cross-cutting concerns (横切关注点): a piece of logic or code that is going to be written in multiple classes/layers but is not business logic
  - Logging
  - Security
  - Transaction management
  - …



Spring AOP



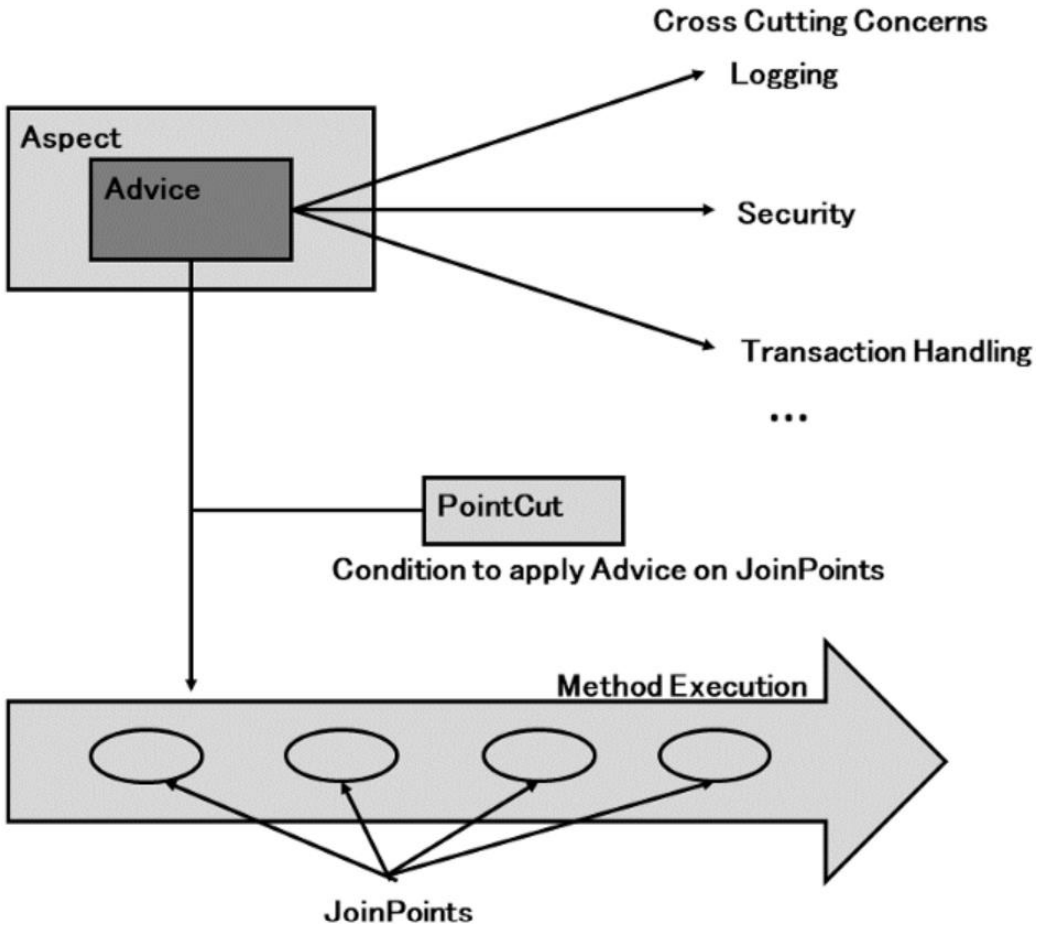
Without AOP: business code and non-business code are tangled together



With AOP: business code and non-business code are decoupled and can be managed independently.

AOP Terminology

- **Aspect**: cross-cutting concerns. In Spring AOP, aspects are typically implemented using regular classes annotated with @Aspect
- **Join point**: a point during the execution of a program. In Spring AOP, a join point always represents a method execution.
- **Advice**: action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.
- **Pointcut**: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (e.g., the execution of a method with a certain name)



https://docs.spring.io/spring-framework/docs/2.5.5/reference/aop.html

Spring AOP Example

`logBeforeV1()` will be executed before `getEmployeeById()` during runtime

```java
@Component
public class EmployeeManager
{
    public EmployeeDTO getEmployeeById(Integer employeeId) {
        System.out.println("Method getEmployeeById() called");
        return new EmployeeDTO();
    }
}
```

Join point: Business logic

```java
@Aspect
public class EmployeeCRUDAspect {

    @Before("execution(* EmployeeManager.getEmployeeById(..))")     //point-cut expression
    public void logBeforeV1(JoinPoint joinPoint)
    {
        System.out.println("EmployeeCRUDAspect.logBeforeV1() : " + joinPoint.getSignature().getName());
    }
}
```

Aspect: Cross-cutting logic (logging)

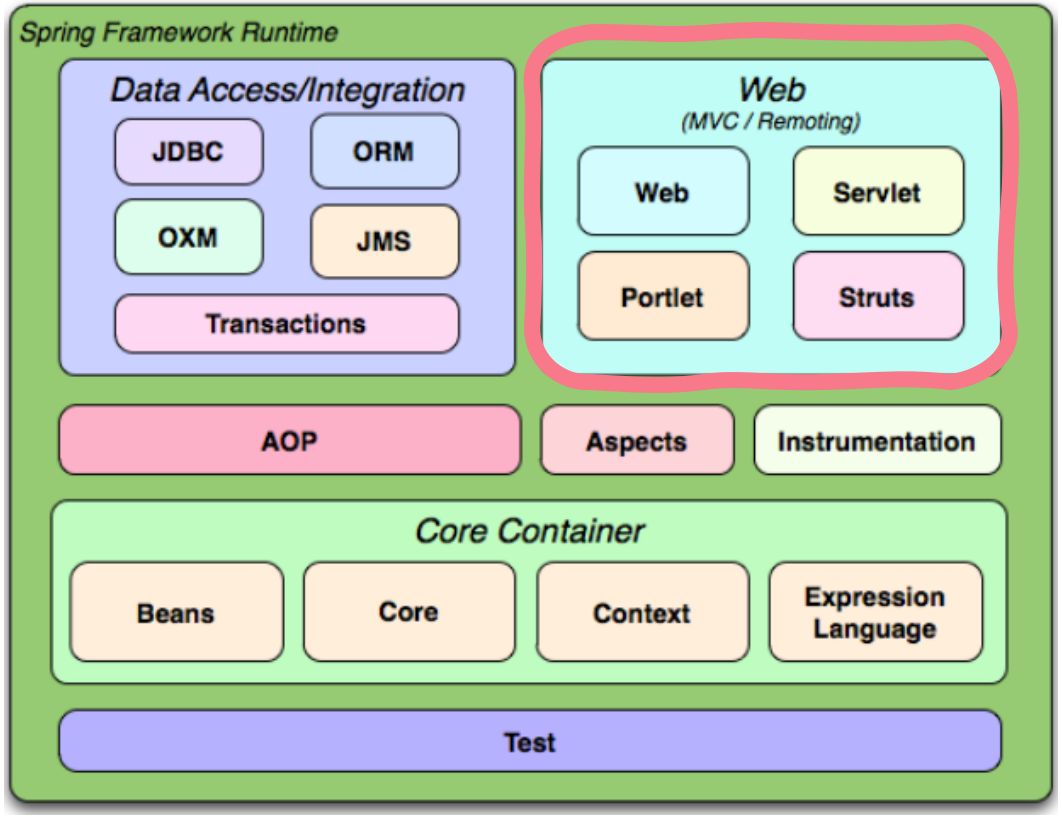
Pointcut: expressions to match joint-point methods

Before Advice: action taken at a join point

https://howtodoinjava.com/spring-aop-tutorial/

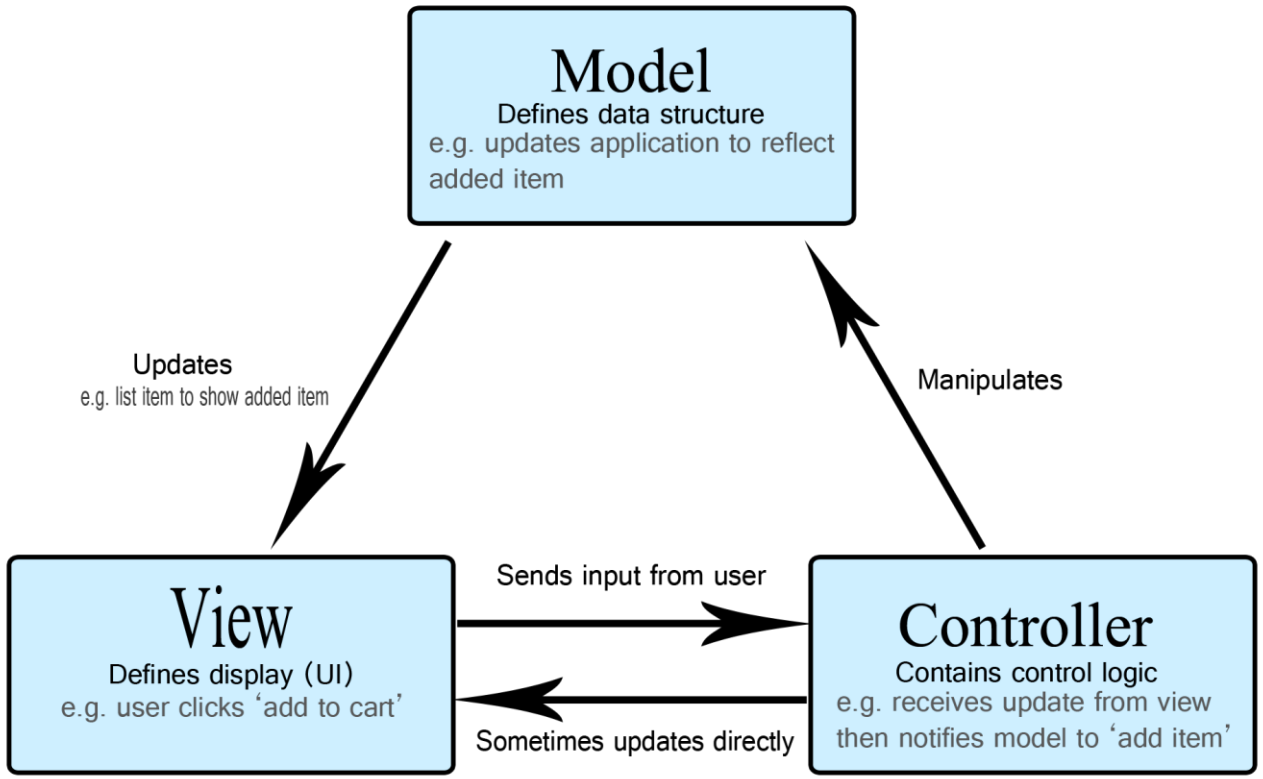
Spring MVC

- The Web layer consists of the `spring-web`, `spring-webmvc`, `spring-websocket`, and `spring-webmvc-portlet` modules.
- Spring MVC is an integrated version of the Spring framework and Model View Controller
  - It has all the basic features of the core Spring framework like Dependency Injection and Inversion of Control
  - The MVC pattern segregates the application’s different aspects (input logic, business logic, and UI logic)
- Spring MVC (`spring-webmvc`) contains Spring’s model-view-controller (MVC) and REST Web Services implementation for web applications.



MVC Design Pattern

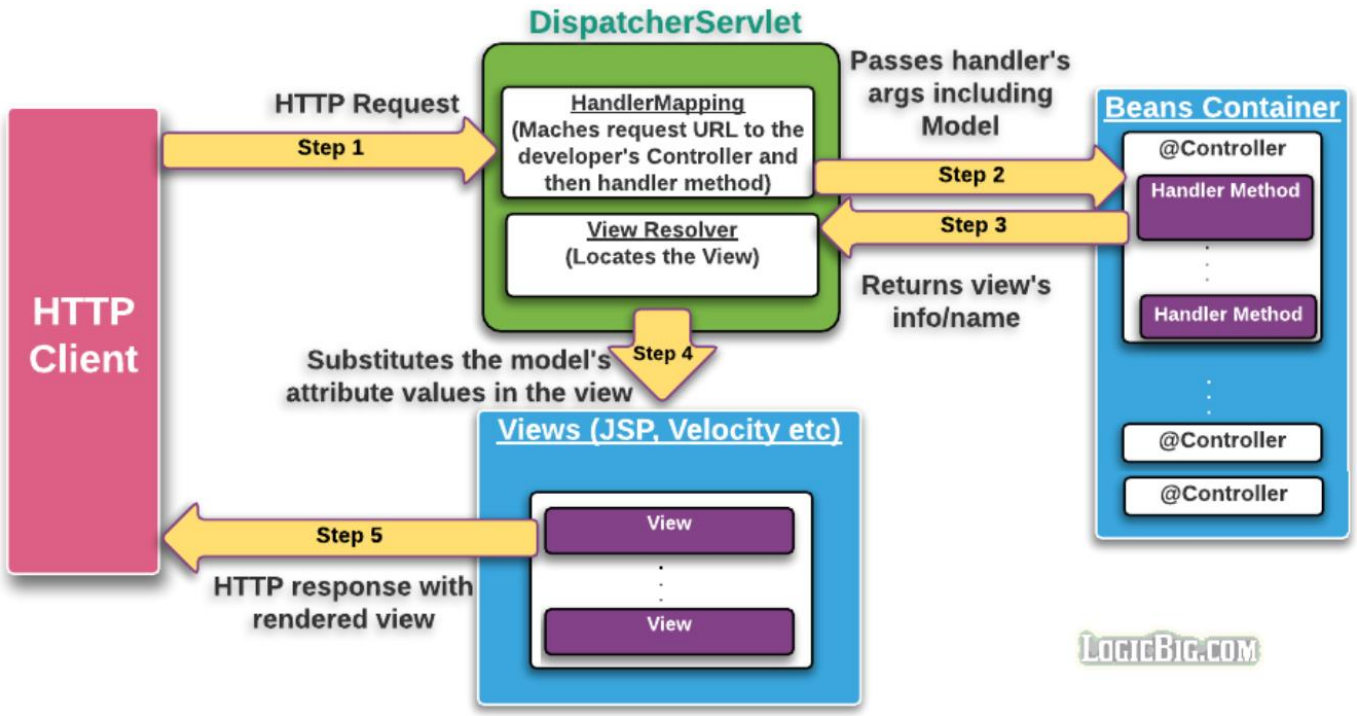
- Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements.
  - **Model** directly manages the data, logic and rules of the application
  - **View** represents the visualization of the data that model contains.
  - **Controller** accepts input and converts it to commands for the model or view



Spring MVC Workflow – The Controller

`DispatcherServlet` (Frontend controller) receives the request and delegates the requests to the controllers based on the requested URI (internally using the `HandlerMapping` object)

A Spring controller is a Java class while its methods are known as handlers. The controller and/or its methods are mapped to request URI using `@RequestMapping`.

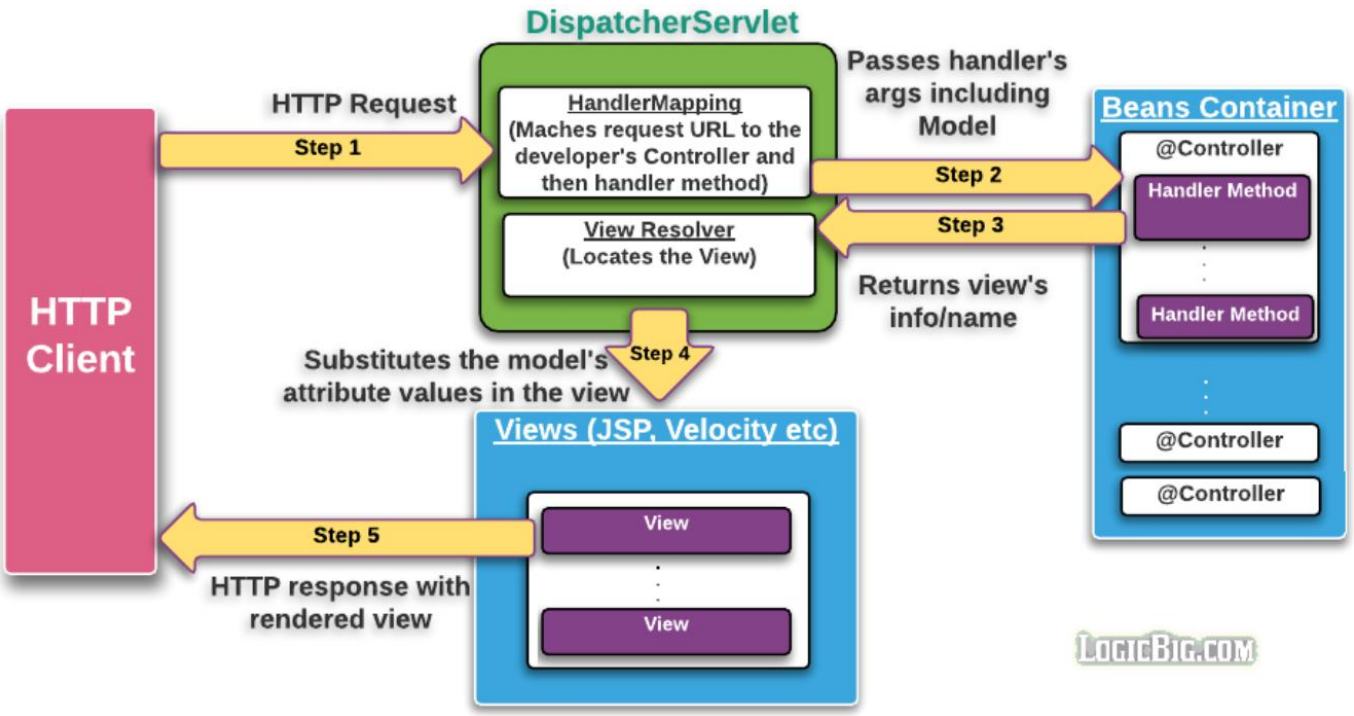


https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/spring-mvc-intro.html

Spring MVC Workflow – The Model

The Model binds the view attributes with application specific values. It's used to transfer data between the view and controller of the Spring MVC application
If the handler method parameters list has Model type, its instance is passed by Spring.

```
@Controller
public class MyMvcController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String prepareView(Model model) {
        //bind msg variable to a value which our jsp view
        //will be using
        model.addAttribute("msg", "Spring quick start!!");
        //return  the name of our jsp page.
        return "my-page";
    }
}
```



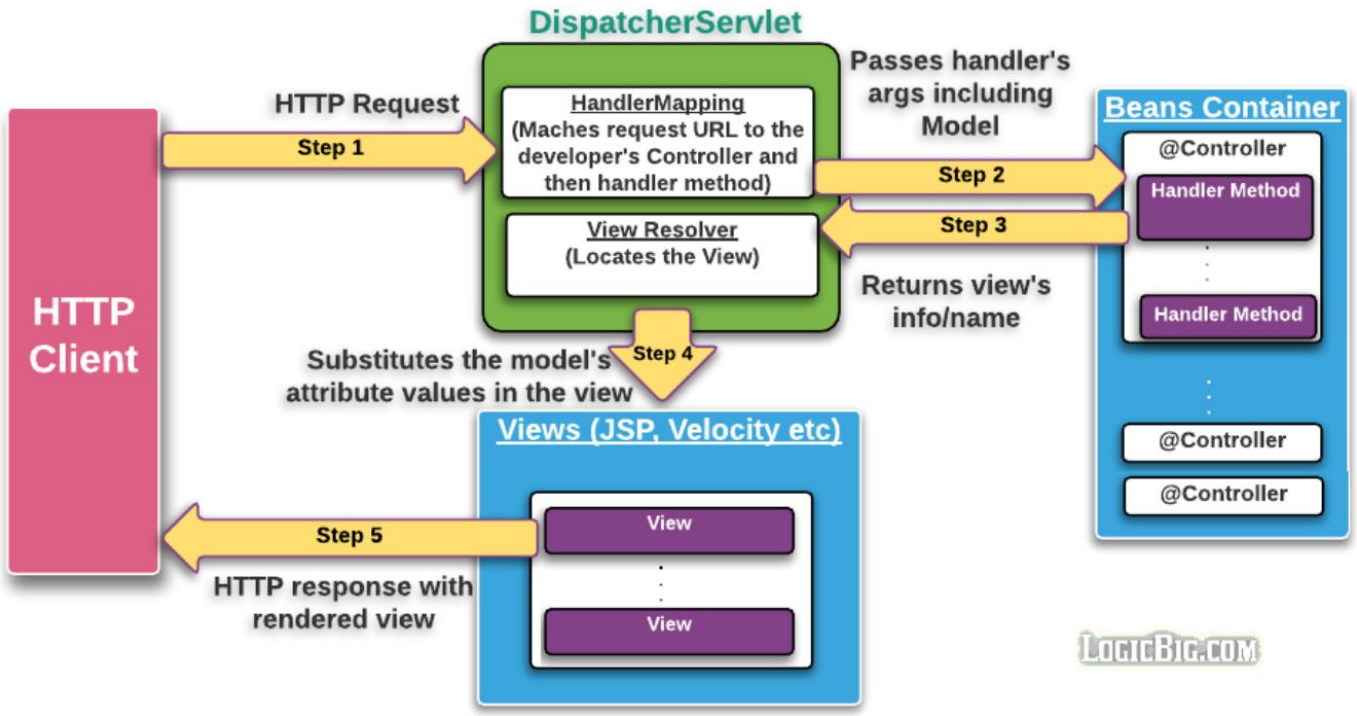
https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/spring-mvc-intro.html

Spring MVC Workflow – The View

```java
@Configuration
public class MyWebConfig {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver =
                        new InternalResourceViewResolver();

        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

```jsp
<%@ page language="java"
    contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<html>
<body>
    Message : ${msg}
</body>
</html>
```

- webapp
  - WEB-INF
    - views
      - my-page.jsp

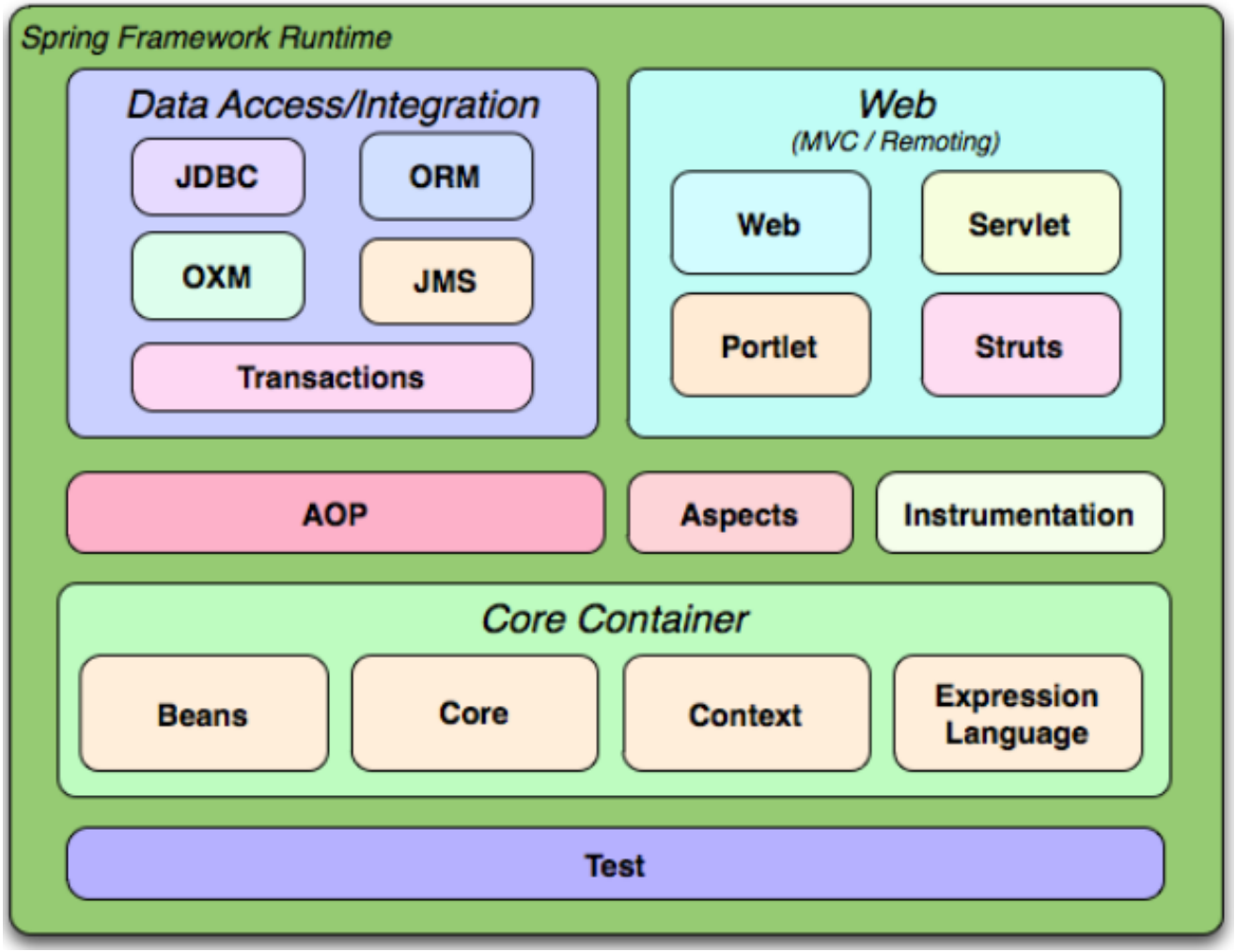


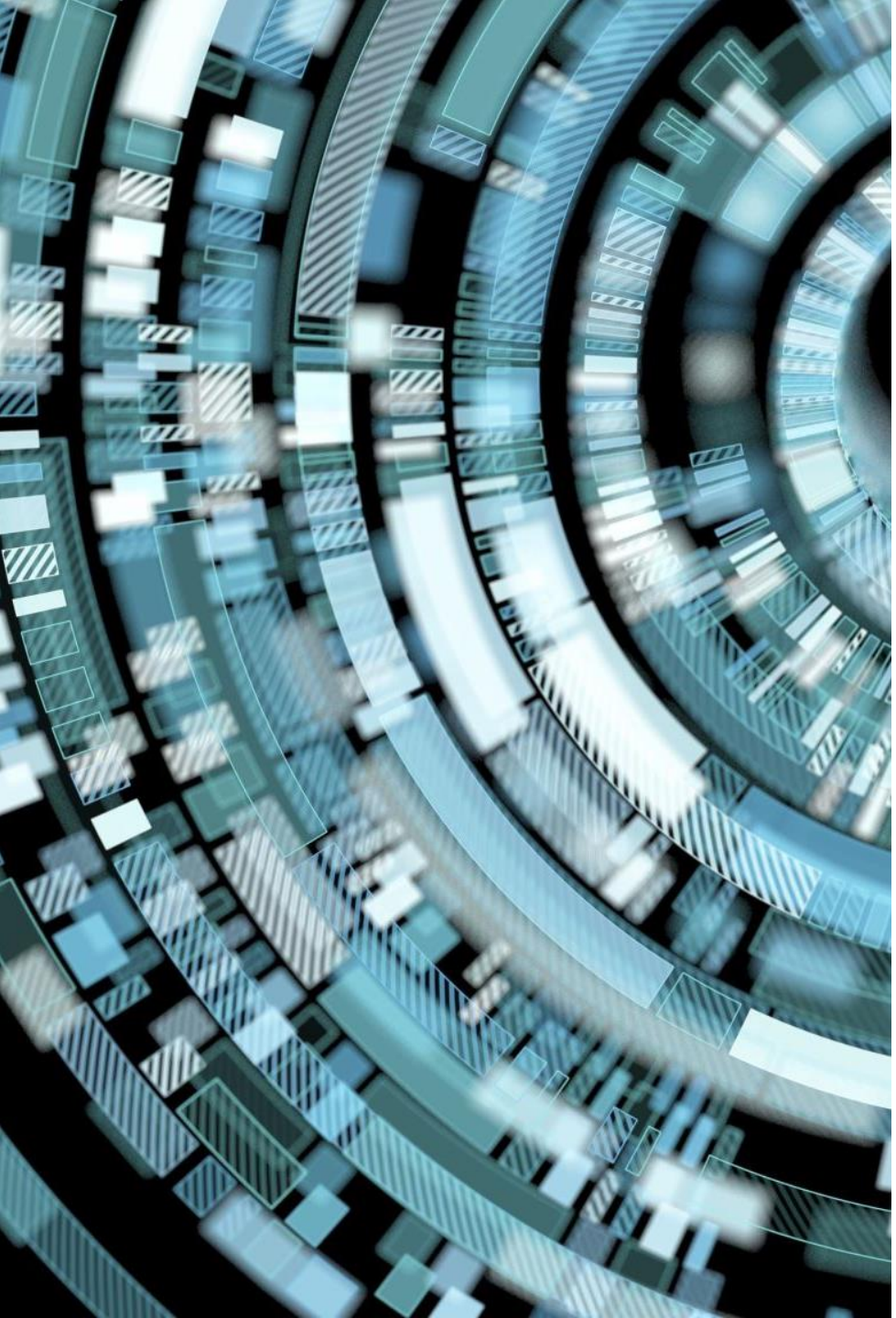
https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/spring-mvc-intro.html

The Spring Framework

- Core Container
- AOP
- Web

- Data Access/Integration
- Test





Lecture 12

- The Spring Framework
  - IoC & Dependency Injection
  - Spring AOP
  - Spring MVC
- Spring Boot
  - Overview
  - Building a MVC web application
  - Building a RESTful web service
  - Microservices

Spring Boot: The History

In October 2012, Mike Youngstrom created a feature request in spring jira asking for support for containerless web application architectures in spring framework. He talked about configuring web container services within a spring container bootstrapped from the main method! Here is an excerpt from the jira request,

> *I think that Spring's web application architecture can be significantly simplified if it were to provided tools and a reference architecture that leveraged the Spring component and configuration model from top to bottom. Embedding and unifying the configuration of those common web container services within a Spring Container bootstrapped from a simple main() method.*

This request lead to the development of spring boot project starting sometime in early 2013. In April 2014, spring boot 1.0.0 was released. Since then a number of spring boot minor versions came out,

https://www.quickprogrammingtips.com/spring-boot/history-of-spring-framework-and-spring-boot.html

spring boot

- `The Spring Framework` can still be quite complex since developers need to perform many configurations manually (and repetitively!)
- `Spring Boot` simplifies and automates the configuration process and speeds up the creation and deployment of Spring applications (e.g., you could create standalone applications with less or almost no configuration overhead)

https://www.fusion-reactor.com/blog/the-difference-between-spring-framework-vs-spring-boot/

spring boot

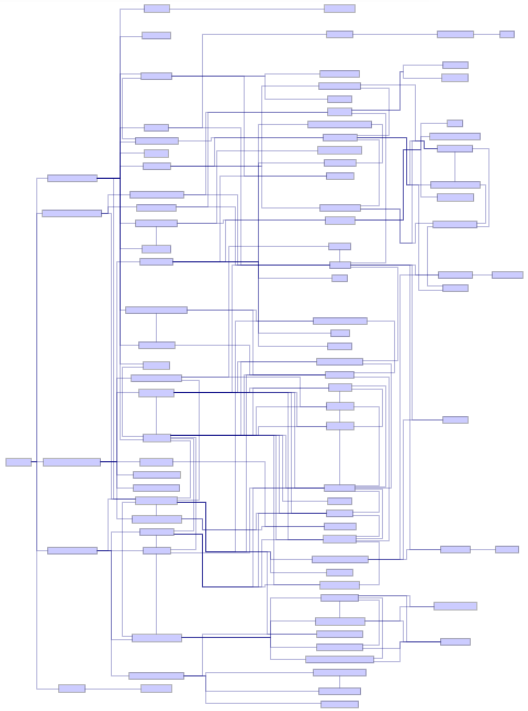
- Spring Boot means bootstrapping a Spring application in such a way that it contains almost everything needed to run a full application.
- Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added.
- Spring Boot takes an **opinionated** view to guide you into their way of configuring things
  - Spring Boot “thinks” that it is the good starting point

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.auto-configuration

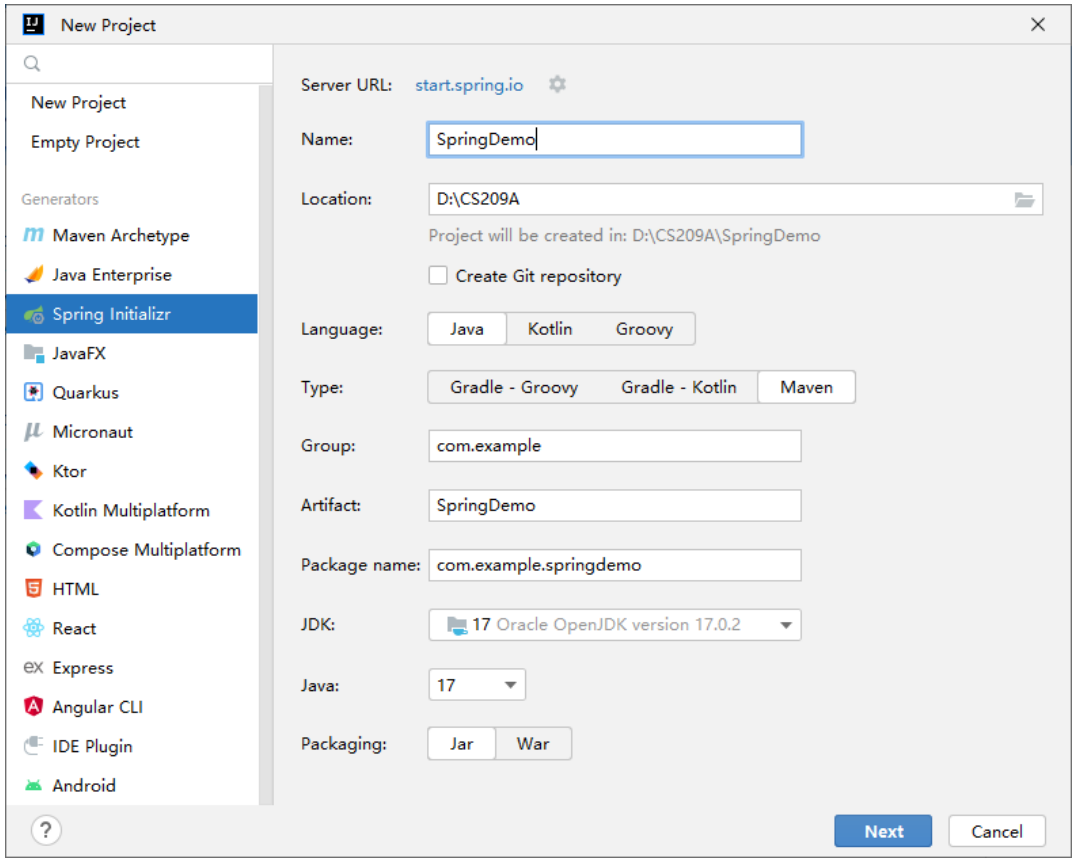
Creating a web application

- Using `Spring Boot`
  - Create a Spring Boot application using Spring initializer
  - Select dependencies (e.g., Spring Web)
  - Done ☺

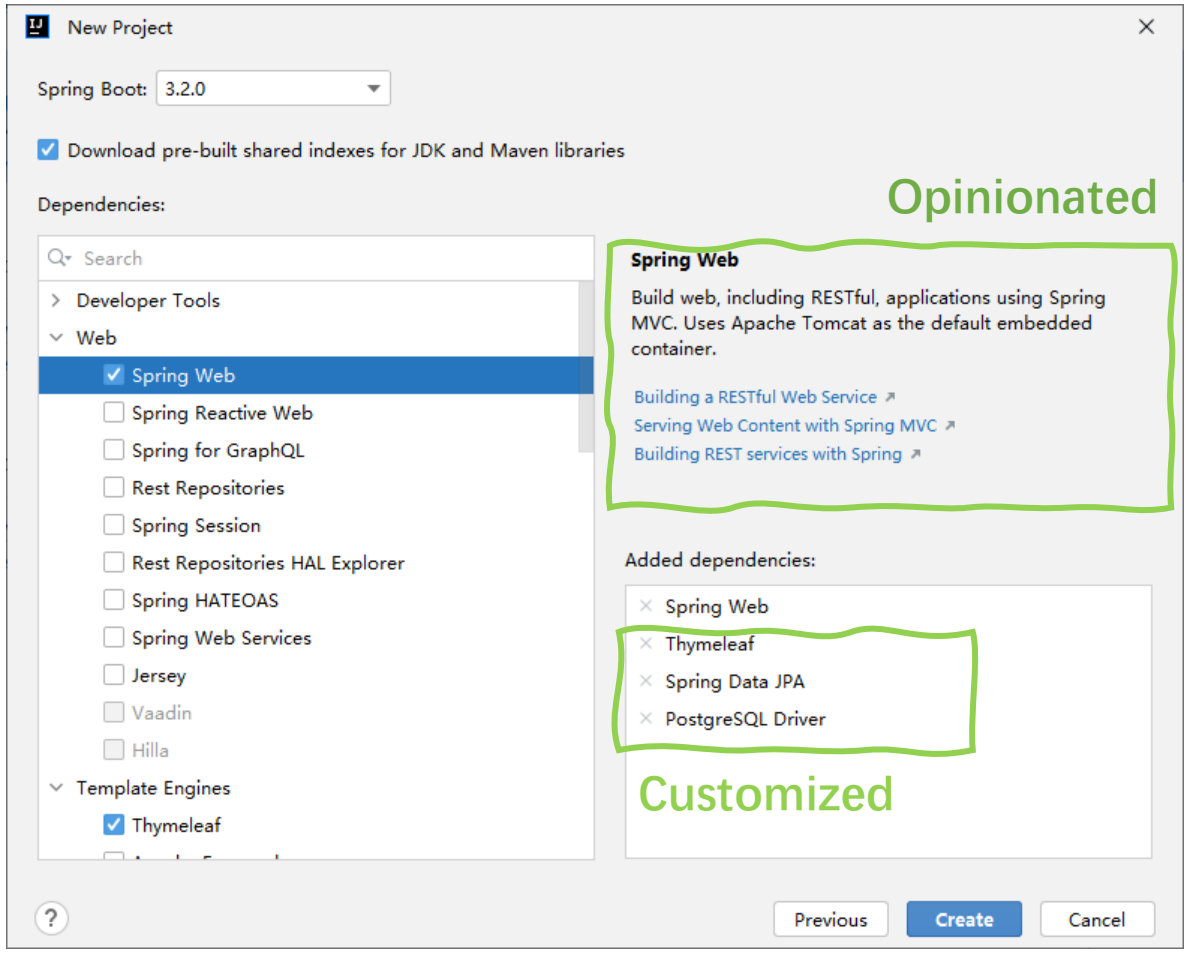
- Using `Spring MVC`
  - Download and confi
  - Manually add mave
    - spring-core
    - spring-context
    - spring-aop
    - spring-webmvc
    - spring-web
    - …
  - Configurations
  - More configuration
  - …



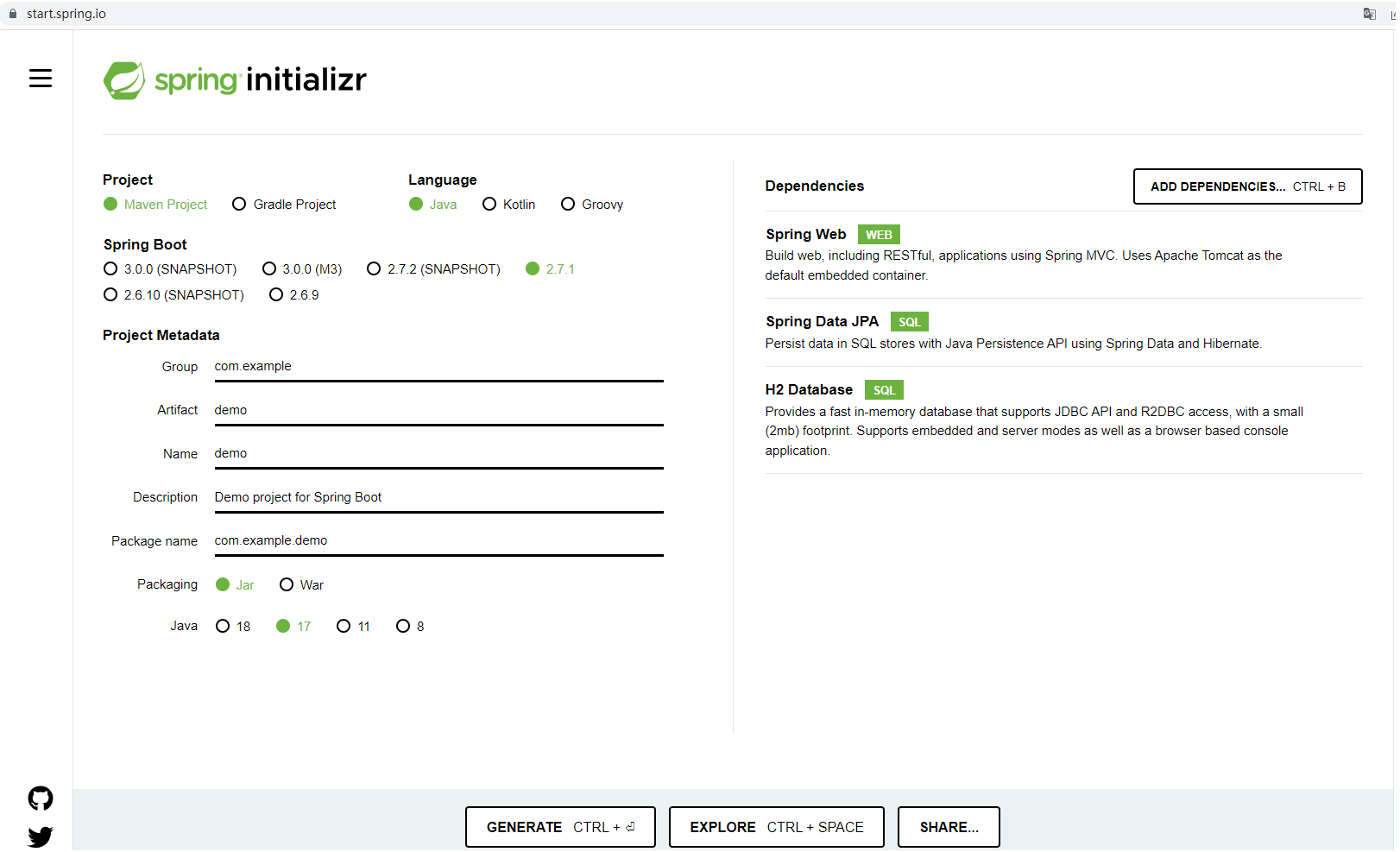
Creating a web app with Spring Initializer



Supported by IntelliJ Ultimate



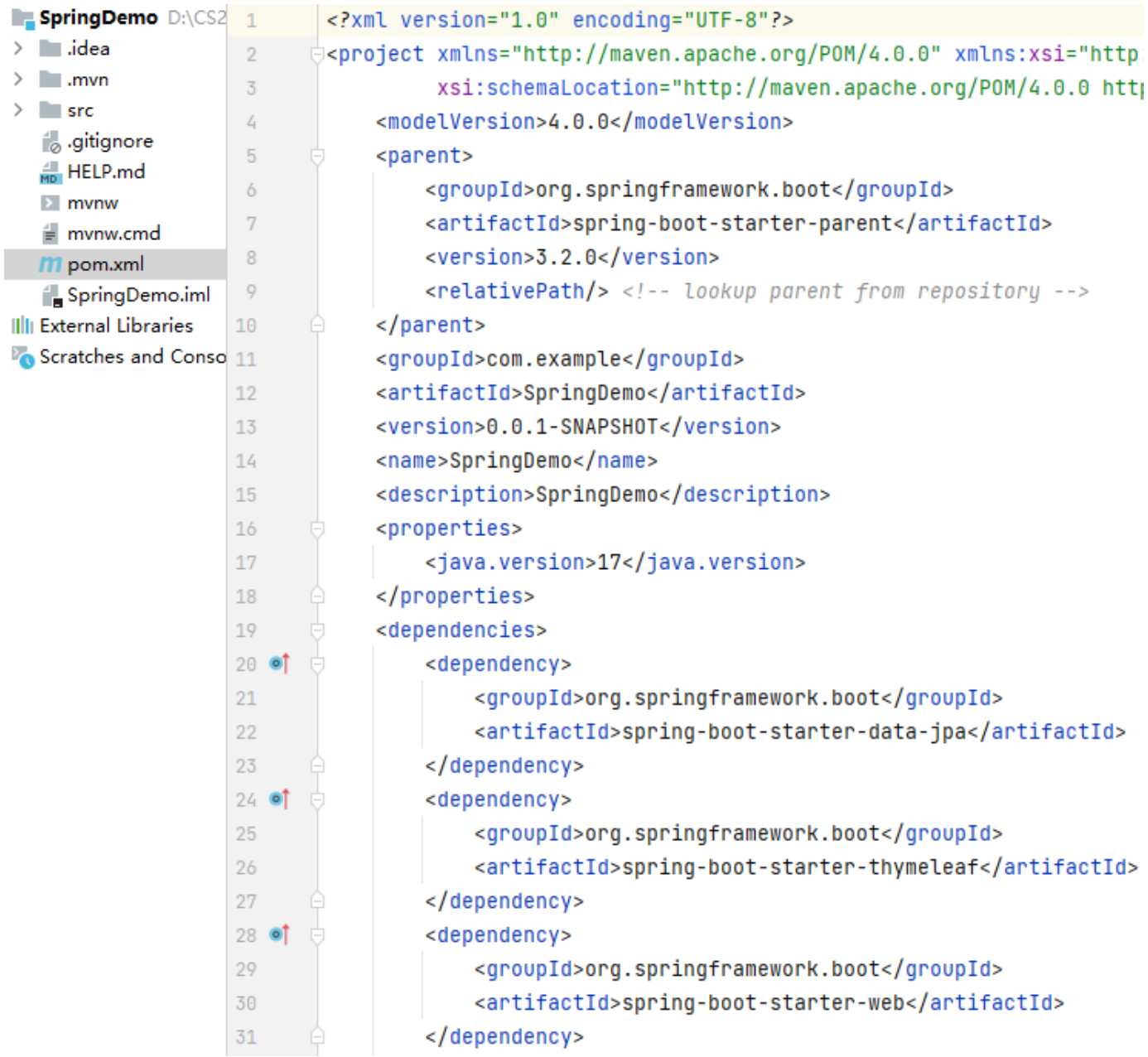
Creating a web app with Spring Initializer



Generate & download the project, then open in IntelliJ

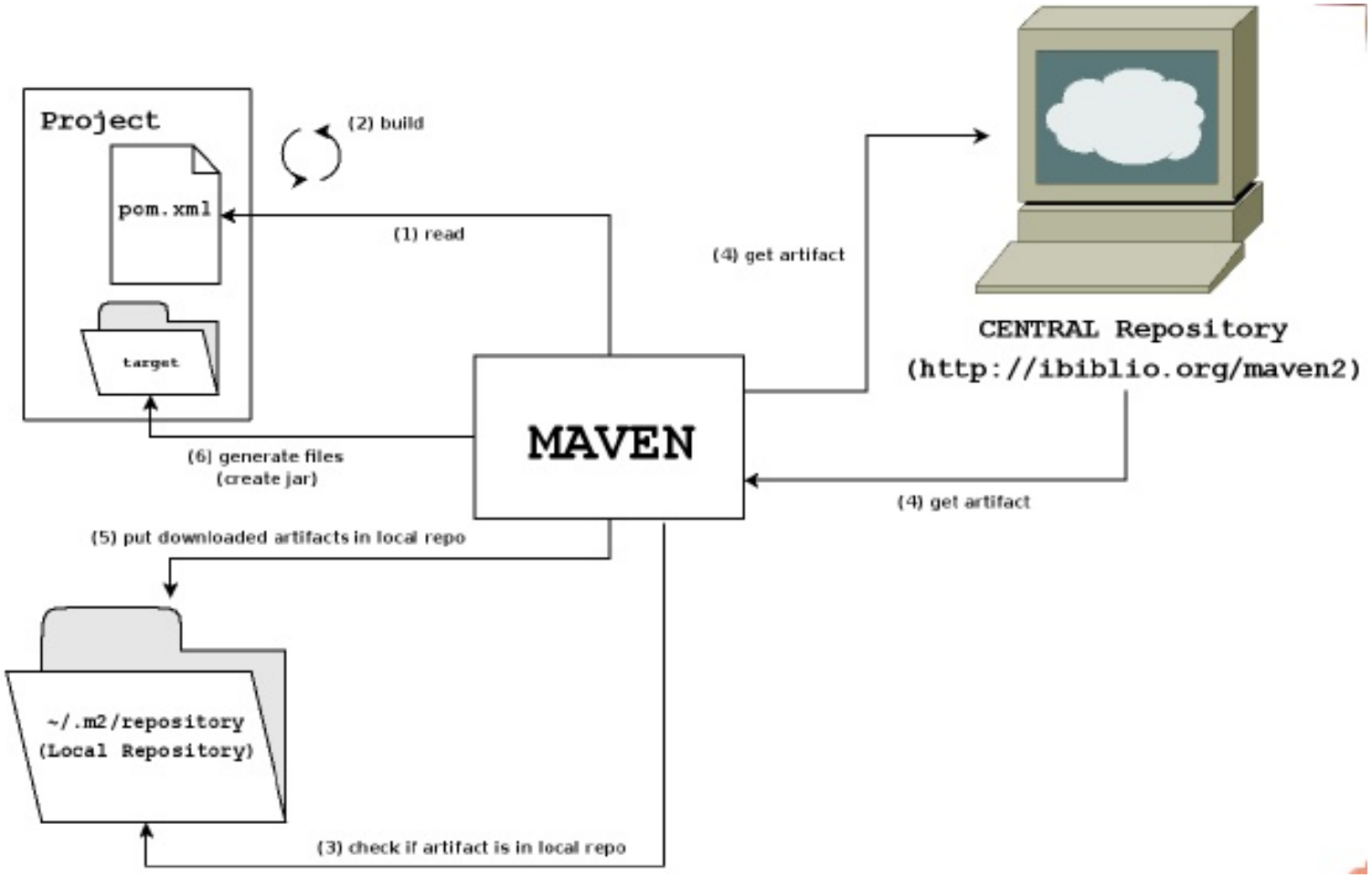
Maven Dependencies

- POM stands for "Project Object Model". It is an XML representation of a Maven project held in a file named `pom.xml`.
- The `pom.xml` file is the core of a project's configuration in Maven.
- It is a single configuration file that contains the majority of information and dependency required to build a project in just the way you want.




```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 htt
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>SpringDemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>SpringDemo</name>
    <description>SpringDemo</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

Maven Workflow



https://www.slideshare.net/sandeepchawla/maven-introduction

spring-boot-starter-parent

A special starter project that sets up

- Default Maven plugins
- Default dependencies & version management
- Default properties & configurations
- …

```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
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>SpringDemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>SpringDemo</name>
    <description>SpringDemo</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

spring-boot-starter-web

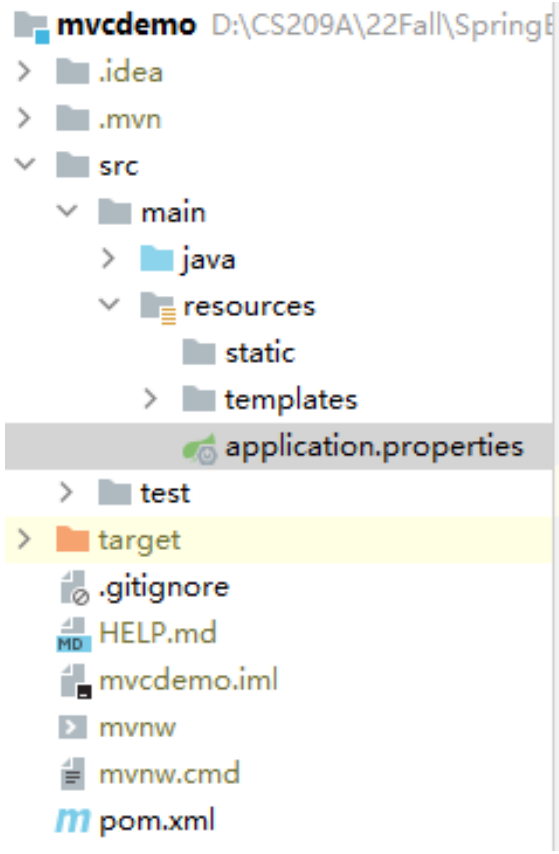
transitively pulls in all dependencies related to web development

```
+- org.springframework.boot:spring-boot-starter-web:jar:2.7.1:compile
|  +- org.springframework.boot:spring-boot-starter-json:jar:2.7.1:compile
|  |  +- com.fasterxml.jackson.core:jackson-databind:jar:2.13.3:compile
|  |  |  +- com.fasterxml.jackson.core:jackson-annotations:jar:2.13.3:compile
|  |  |  \- com.fasterxml.jackson.core:jackson-core:jar:2.13.3:compile
|  |  +- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:jar:2.13.3:comp
|  |  +- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar:2.13.3:co
|  |  \- com.fasterxml.jackson.module:jackson-module-parameter-names:jar:2.13
|  +- org.springframework.boot:spring-boot-starter-tomcat:jar:2.7.1:compile
|  |  +- org.apache.tomcat.embed:tomcat-embed-core:jar:9.0.64:compile
|  |  +- org.apache.tomcat.embed:tomcat-embed-el:jar:9.0.64:compile
|  |  \- org.apache.tomcat.embed:tomcat-embed-websocket:jar:9.0.64:compile
|  +- org.springframework:spring-web:jar:5.3.21:compile
|  \- org.springframework:spring-webmvc:jar:5.3.21:compile
|     \- org.springframework:spring-expression:jar:5.3.21:compile
```

```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
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>SpringDemo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>SpringDemo</name>
    <description>SpringDemo</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

Application Properties

1.Core Properties
2.Cache Properties
3.Mail Properties
4.JSON Properties
5.Data Properties
6.Transaction Properties
7.Data Migration Properties
8.Integration Properties
9.Web Properties
10.Templating Properties
11.Server Properties
12.Security Properties
13.RSocket Properties
14.Actuator Properties
15.DevTools Properties
16.Testing Properties




```
spring.datasource.url=jdbc:postgresql://localhost:5432/cs209a
spring.datasource.username=postgres
spring.datasource.password=123456
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.
spring.jpa.properties.hibernate.format_sql=true

server.error.include-message=always
```

We use this `application.properties` file to configure our Spring Boot application

Application Class

- `@SpringBootApplication` annotation enables 3 features:

  - `@Configuration`: allow to register extra beans in the context or import additional configuration classes
  - `@ComponentScan`: enable @Component scan on the package where the application is located
  - `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism

```java
@SpringBootApplication
public class MvcDemoApplication {
    yidatao
    public static void main(String[] args) {
        SpringApplication.run(MvcDemoApplication.class, args);
    }

}
```

MvcdemoApplication

Console | Actuator

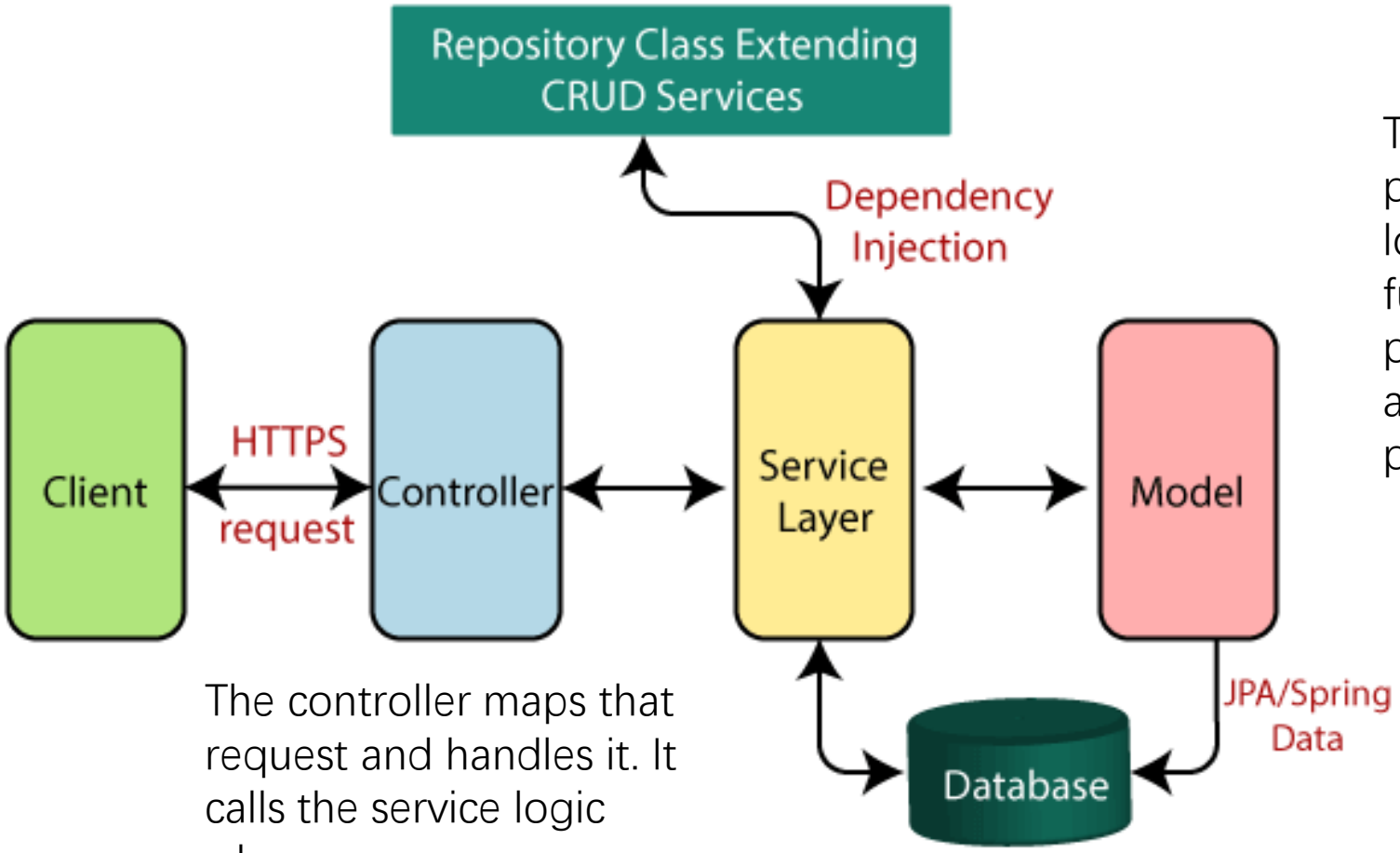
```
C:\Users\admin\.jdks\openjdk-17.0.2\bin\java.exe ...
OpenJDK 64-Bit Server VM warning: Options -Xverify:none and -noverif

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.1)
```

Convention over Configuration

- Convention over Configuration (programming by convention), is a software design paradigm that aims to reduce the number of decisions software developers have to make, with the benefits of simplicity without losing flexibility.
- Developers only need to specify the non-conforming parts of the application
- E.g., when we import a spring-boot-starter-web.jar, Spring Boot automatically imports Spring MVC dependencies and configures a built-in Tomcat container.

Spring Boot Flow Architecture

The client makes HTTP requests, which go to the controllers

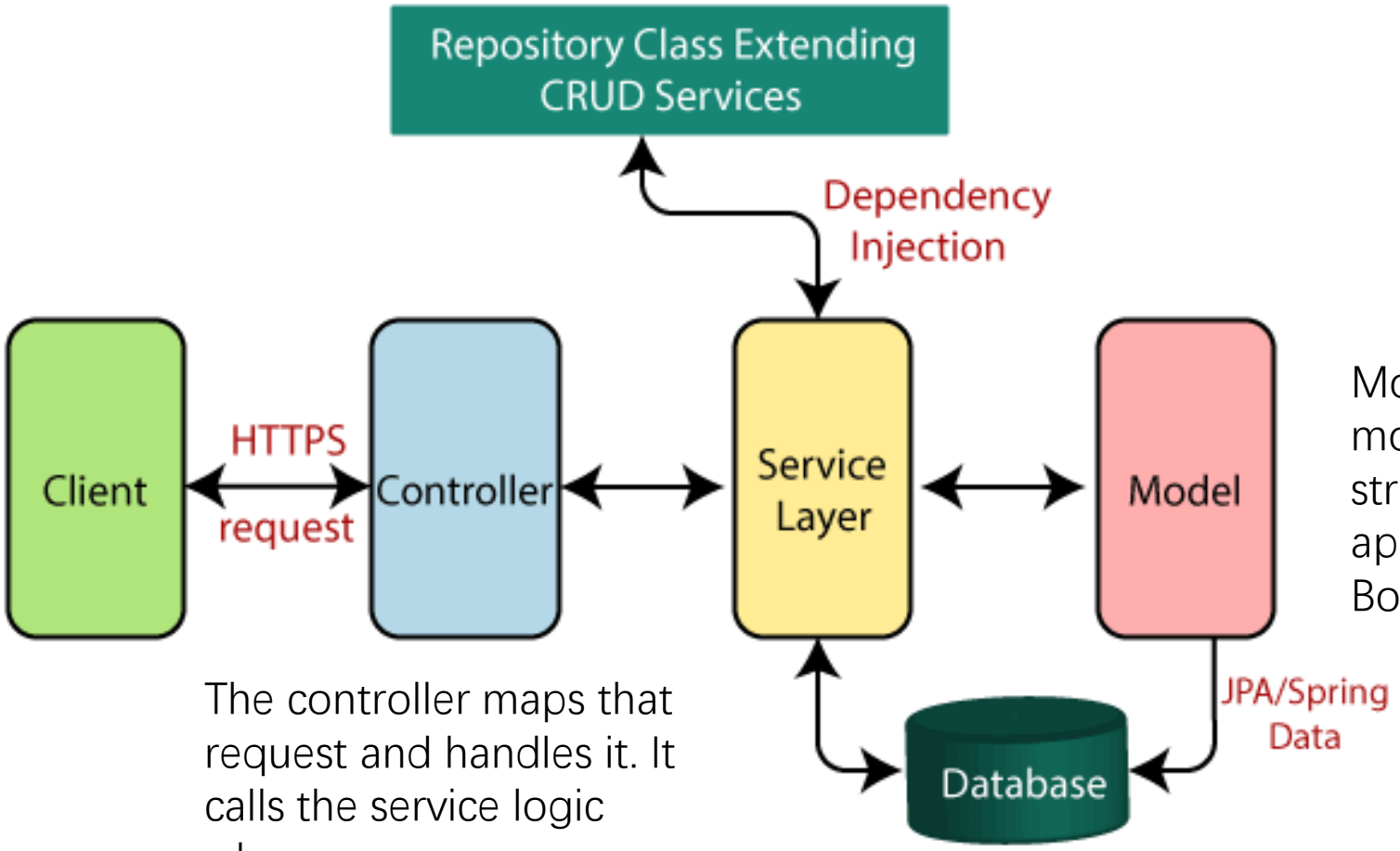


The controller maps that request and handles it. It calls the service logic when necessary.

The service layer performs the business logic. It defines which functionalities you provide, how they are accessed, and what to pass and get in return

Spring Boot Flow Architecture

The client makes HTTP requests, which go to the controllers



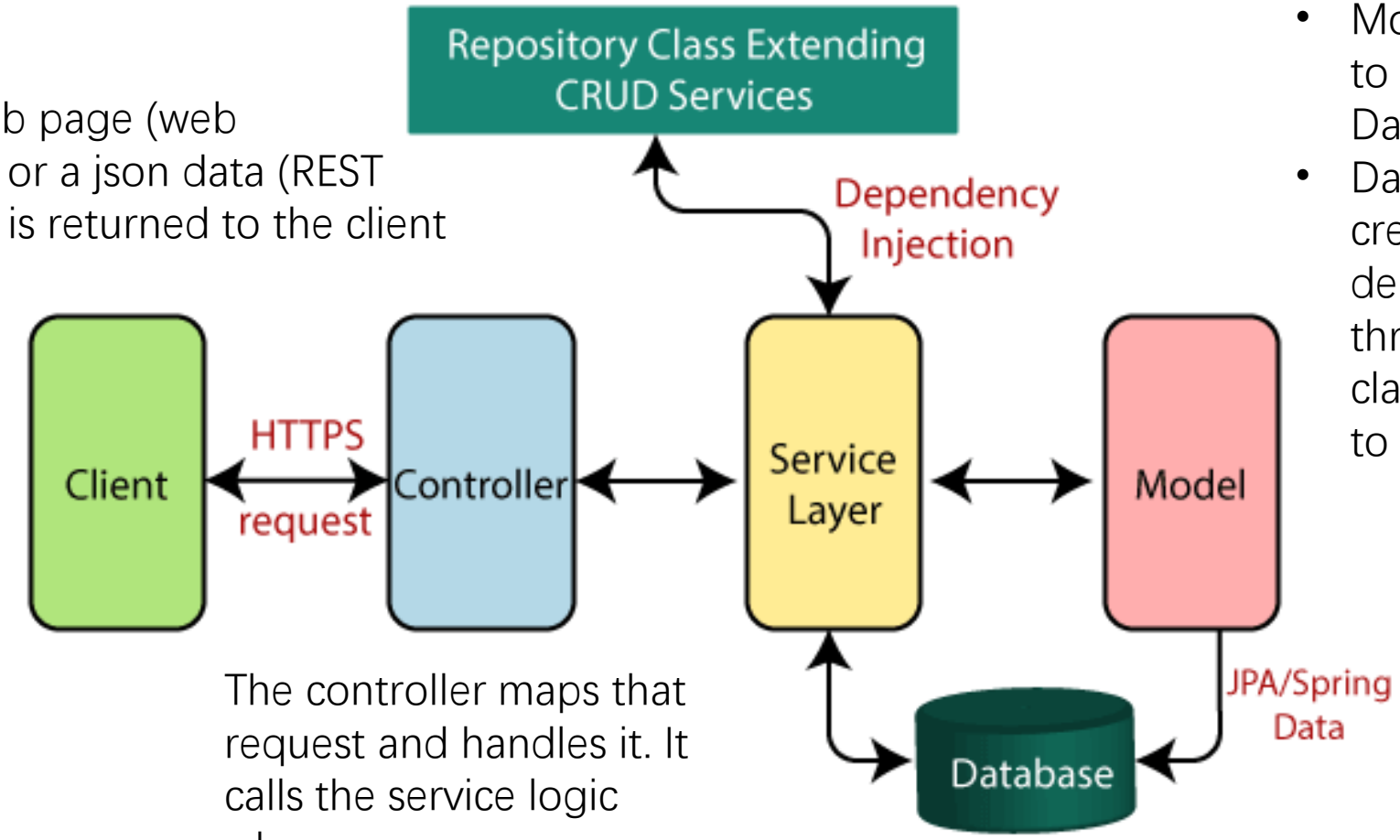
Model includes data model/entities/data structures used in the application, e.g., User, Book, Product, Account

The controller maps that request and handles it. It calls the service logic when necessary.

Spring Boot Flow Architecture

Finally, a web page (web application) or a json data (REST application) is returned to the client

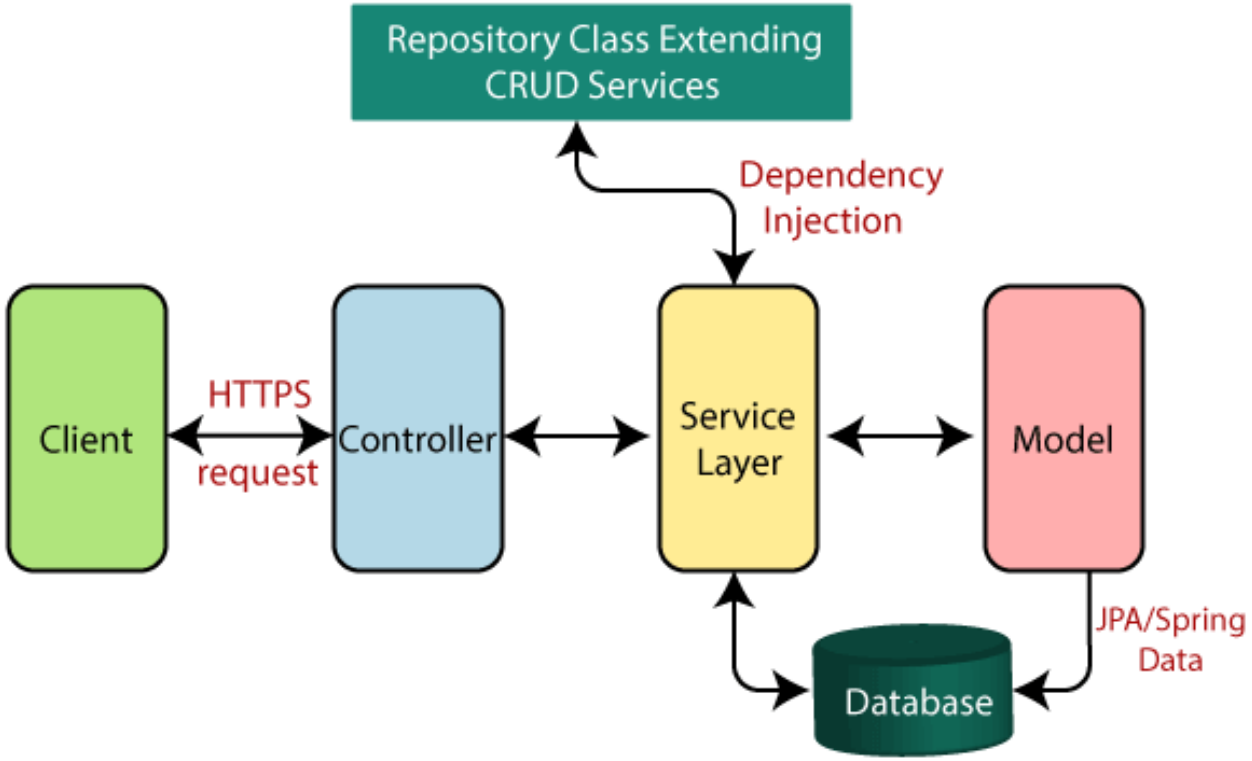
The client makes HTTP requests, which go to the controllers

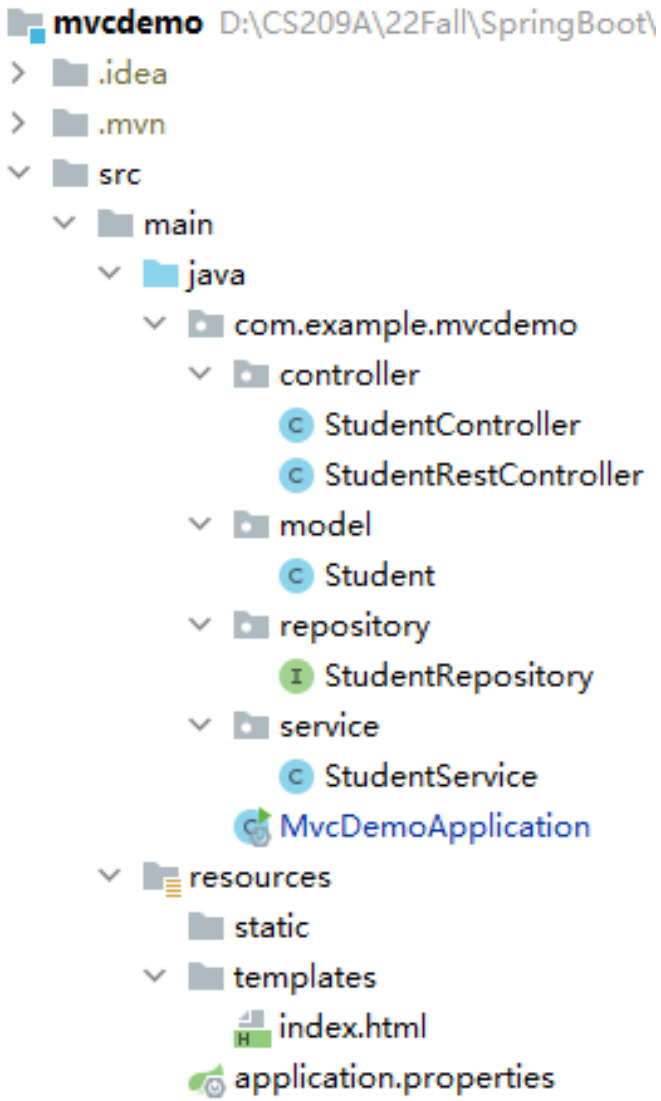


The controller maps that request and handles it. It calls the service logic when necessary.

- Model object is mapped to database by Spring Data JPA
- Data operations (e.g., create, read, update, delete) are performed through a repository class, which is injected to the service class

A Simple Student Management Web Application





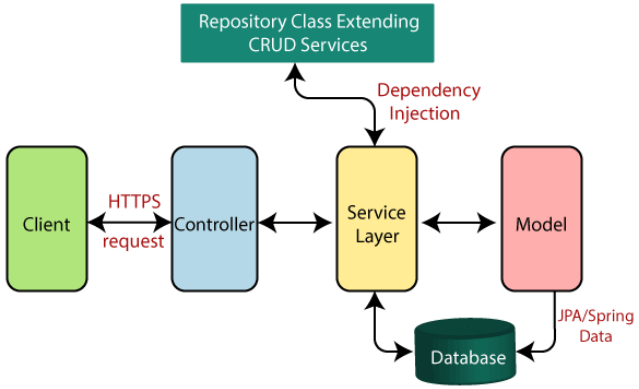
Model

- mvcdemo D:\CS209A\22Fall\SpringBoot\
  - .idea
  - .mvn
  - src
    - main
      - java
        - com.example.mvcdemo
          - controller
            - StudentController
            - StudentRestController
          - model
            - Student
          - repository
            - StudentRepository
          - service
            - StudentService
          - MvcDemoApplication
      - resources
        - static
        - templates
          - index.html
        - application.properties

- Student
  - Student()
  - Student(String, String)
  - Student(Long, String, String)
  - getId(): Long
  - setId(Long): void
  - getName(): String
  - setName(String): void
  - getEmail(): String
  - setEmail(String): void
  - toString(): String ↑Object
  - id: Long
  - name: String
  - email: String

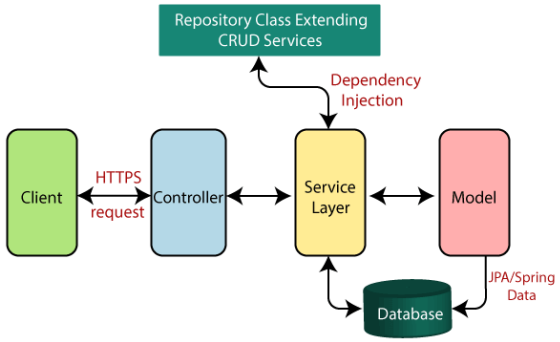
JavaBean: a POJO that conforms to certain conventions

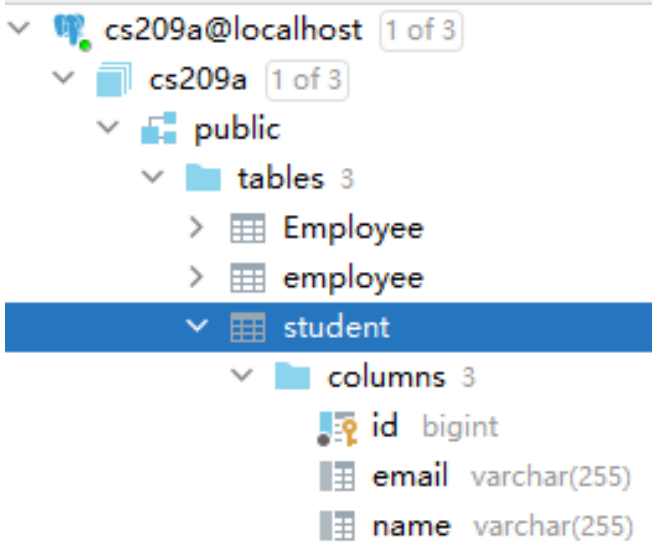
- All properties are private
- Public setters and getters
- A public no-argument constructor



Mapping Model Class to Database Table

- `@Entity`: specifies that the class is an entity and is mapped to a database table
- `@Table`: specifies the name of the database table to be used for mapping (default is the class name)
- `@Id`: specifies the primary key of an entity
- `@GeneratedValue`: specifies the generation strategies for the values of primary keys (default: auto).

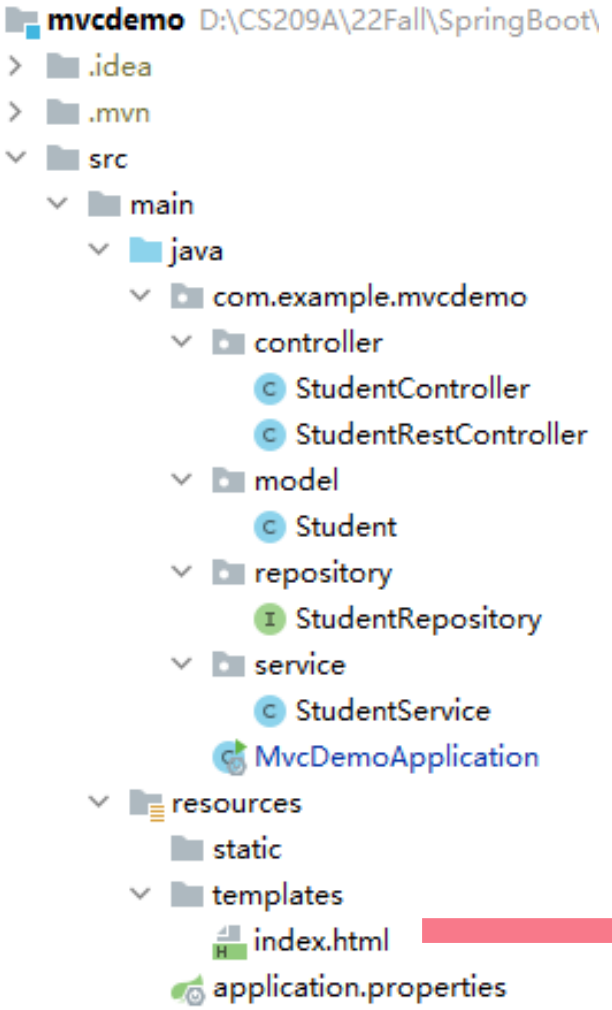





```
@Entity
@Table
public class Student {
    4 usages
    @Id
    @GeneratedValue
    private Long id;
    5 usages
    private String name;
    5 usages
    private String email;

    1 usage   yidatao
    public Student() {
    }
```

View

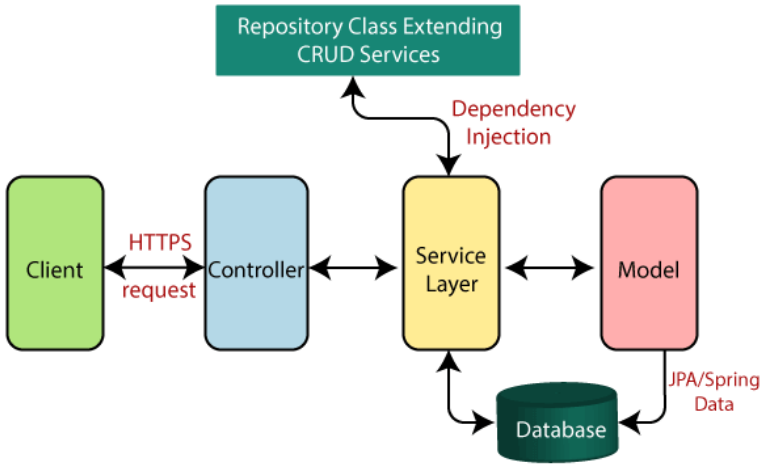
Thymeleaf is a modern server-side Java template engine for both web and standalone environments, allowing HTML to be correctly displayed in browsers and also work as static prototypes

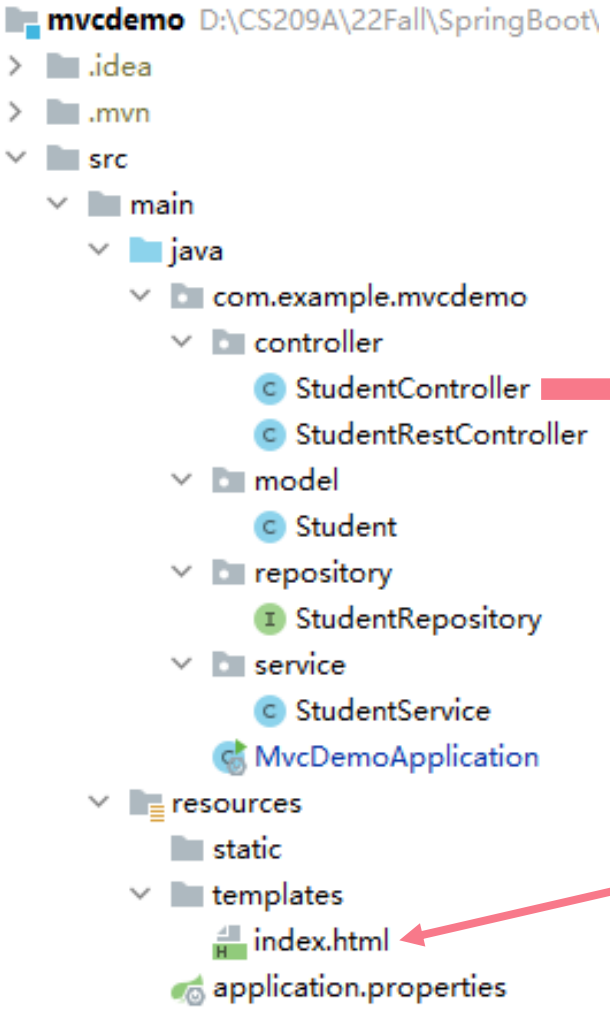



```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Spring Boot Demo</title>
</head>
<body>
 <h1>Student List</h1>
<table>
    <tr th:each="student: ${students}">
        <td th:text="${student.id}">ID</td>
        <td th:text="${student.name}">Name</td>
        <td th:text="${student.email}">Email</td>
    </tr>
</table>
</body>
</html>
```

Display model attributes in HTML.

Controller

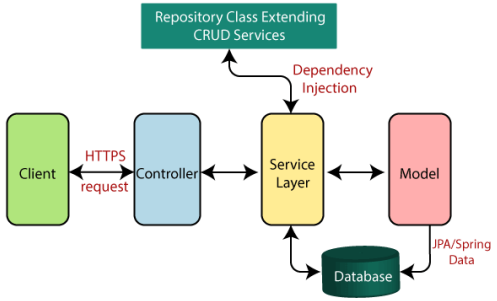


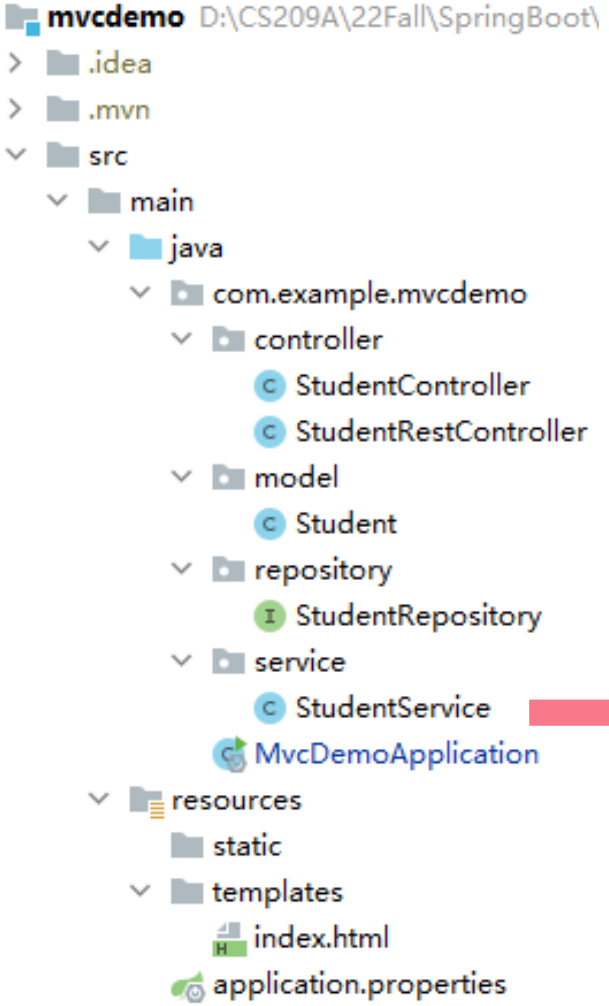



```
@Controller
public class StudentController {

    3 usages
    private final StudentService studentService;

    yidatao
    public StudentController(StudentService studentService) {
        this.studentService = studentService;
    }

    yidatao +1
    @RequestMapping("/list")
    public String getStudents(Model model){
        model.addAttribute( attributeName: "students", studentService.getStudents());
        return "index";
    }
```

`@Controller` is a class-level annotation that marks a class as a web request handler. It is often used to serve web pages. It is mostly used with `@RequestMapping` annotation.

Service



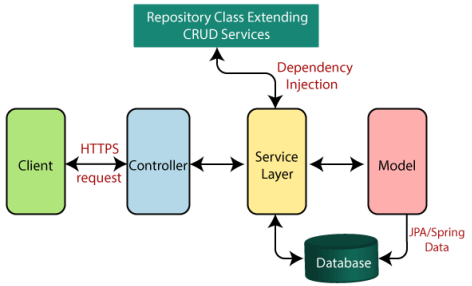


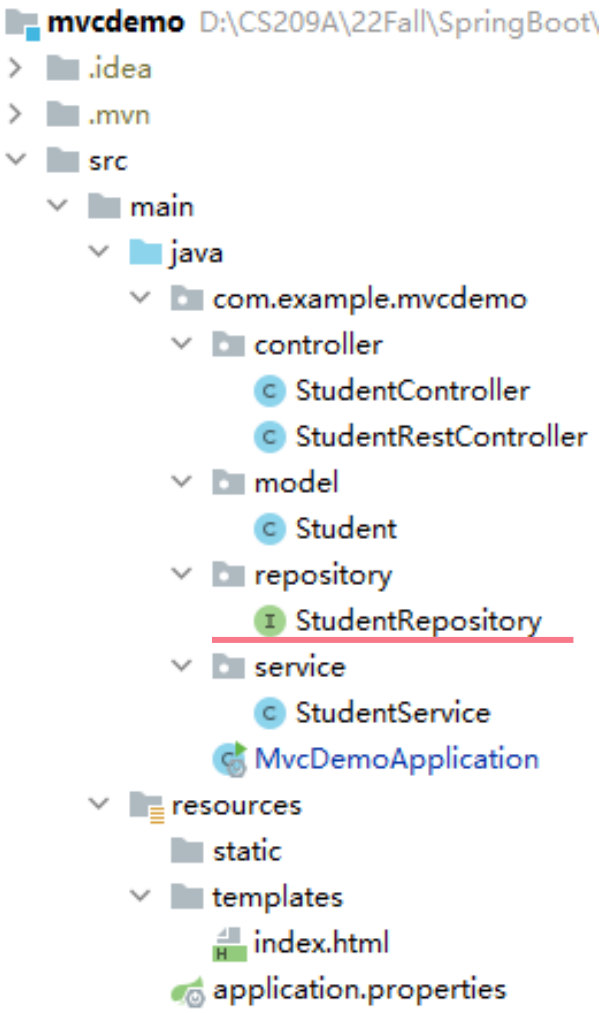

```
@Service
public class StudentService {
    7 usages
    private final StudentRepository studentRepository;

    yidatao
    @Autowired
    public StudentService(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    2 usages   new *
    public List<Student> getStudents(){
        return studentRepository.findAll();
    }

    yidatao *
    public void addStudents(){
        Student maria = new Student( name: "Mary",
                email: "mary@gmail.com");
        Student alex = new Student( name: "Alex",
                email: "alex@gmail.com");
        Student dean = new Student( name: "Dean",
                email: "dean@yahoo.com");
        studentRepository.saveAll(List.of(maria, alex, dean));
    }
```

`@Service`: used with classes that provide business functionalities.

`@Autowired`: injecting beans at runtime

Repository






```
import com.example.mvcdemo.model.Student;
import org.springframework.data.jpa.repository.JpaRepository;

4 usages   yidatao *
public interface StudentRepository extends JpaRepository<Student, Long> {

}
```

Entity/Model class name

Type of the id

A `JpaRepository` defines basic methods for performing CRUD operations, sorting and paginating data.

Repository

- A `JpaRepository` defines basic methods for performing CRUD operations, sorting and paginating data.
- To uses these methods, developers only need to extend specific `JpaRepository` for each domain/model entity (i.e., `Student`) in the application.
- Developers don't need to implement these methods. Spring Data JPA implements them automatically (by using Hibernate as the default implementation)

Interface CrudRepository<T,ID>

Method
count()
delete(T entity)
deleteAll()
deleteAll(Iterable <? extends T> entities)
deleteAllById(Iterable <? extends ID> ids)
deleteById(ID id)
existsById(ID id)
findAll()
findAllById(Iterable <ID> ids)
findById(ID id)
save(S entity)
saveAll(Iterable <S> entities)

Repository

- We could also define customized finder methods, following specific naming conventions, e.g.,
  - Method prefixes should be: findBy, readBy, queryBy, countBy, getBy…
  - Certain keywords are allowed
- Again, we don't need to actually implement them. Spring will generate the implementation automatically

Keyword	Sample
And	findByLastnameAndFirstname
Or	findByLastnameOrFirstname
Is, Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals
Between	findByStartDateBetween
LessThan	findByAgeLessThan
LessThanEqual	findByAgeLessThanEqual
GreaterThan	findByAgeGreaterThan
GreaterThanEqual	findByAgeGreaterThanEqual
After	findByStartDateAfter
Before	findByStartDateBefore
IsNull	findByAgeIsNull
IsNotNull, NotNull	findByAge(Is)NotNull
Like	findByFirstnameLike
NotLike	findByFirstnameNotLike
StartingWith	findByFirstnameStartingWith
EndingWith	findByFirstnameEndingWith
Containing	findByFirstnameContaining
OrderBy	findByAgeOrderByLastnameDesc
Not	findByLastnameNot

```
4 usages   yidatao *
public interface StudentRepository extends JpaRepository<Student, Long> {
    1 usage   new *
    List<Student> findByEmailLike(String email);
}
```

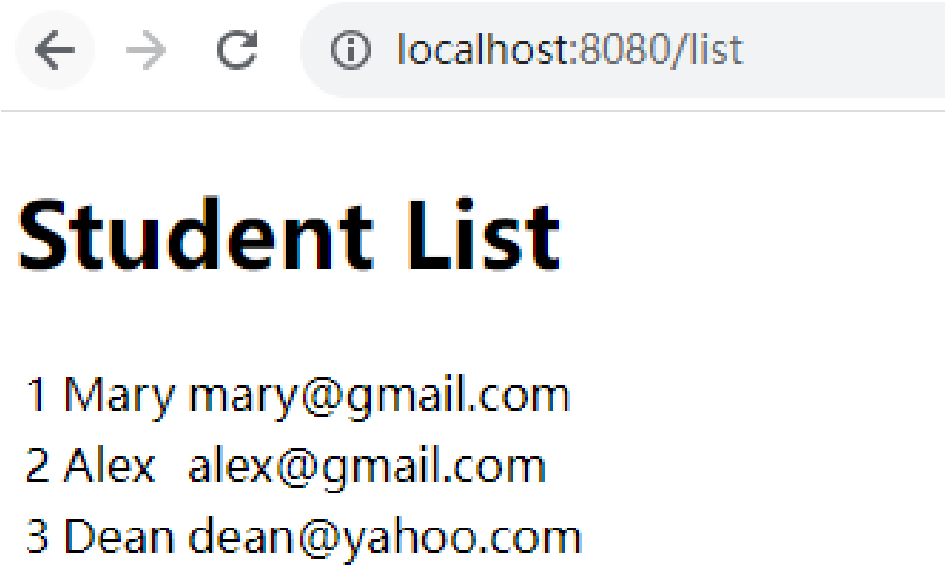
Repository

```
@Service
public class StudentService {
    7 usages
    private final StudentRepository studentRepository;
    yidatao
    @Autowired
    public StudentService(StudentRepository studentRepository) { this.studentRepositor

    2 usages   new *
    public List<Student> findByEmailLike(String email){
        return studentRepository.findByEmailLike("%" + email + "%");
    }

    1 usage   new *
    @Transactional
    public void updateStudent(Long studentId, String name, String email) {
        Student s = studentRepository.findById(studentId).
                orElseThrow(()-> new IllegalStateException("Student ID not exists"));
        if(name!=null && name.length()>0 && !name.equals(s.getName())){
            s.setName(name);
        }
        if(email!=null && email.length()>0 && !email.equals(s.getEmail())){
            s.setEmail(email);
        }
    }
```

Bootstrap

Spring boot's **`CommandLineRunner`** interface is used to run a code block only once in application's lifetime – after application is initialized.

```
@SpringBootApplication
public class MvcDemoApplication {
    yidatao
    public static void main(String[] args) {
        SpringApplication.run(MvcDemoApplication.class, args);
    }

    yidatao
    @Bean
    public CommandLineRunner commandLineRunner(StudentService service){
        return args -> {
            service.addStudents();
        };
    }

}
```



Building a RESTful Web Service

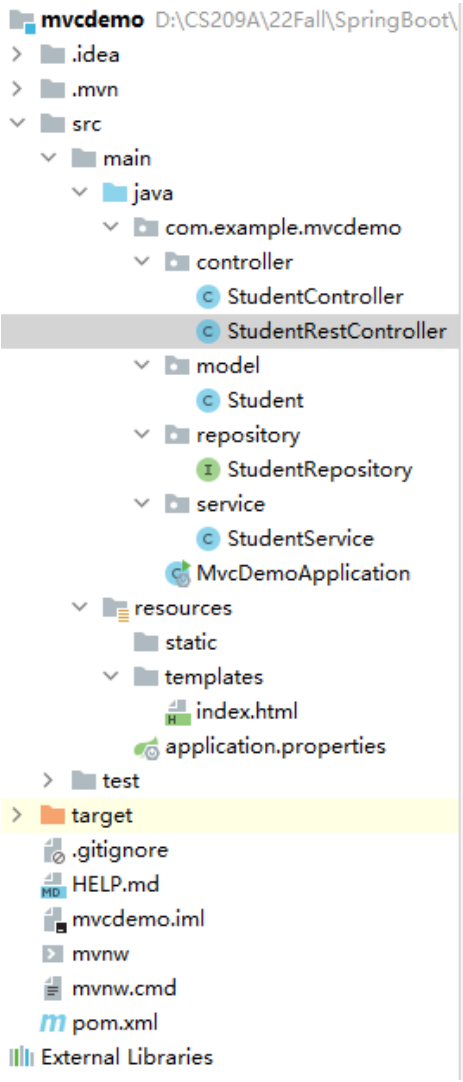
- Key difference between an MVC controller and RESTful controller: how HTTP response body is created
  - **MVC controller**: relies on a view technology to return data in HTML
  - **REST controller**: returns data as object, which is written directly to the HTTP response as JSON

- Spring Initializer: `Spring Web` is sufficient

RestController

`@RestController`: marks the class as a controller where every method returns a domain object instead of a view (shorthand for @Controller+@ResponseBody)

`@RequestMapping`: defines a base URL for all the REST APIs created in this controller

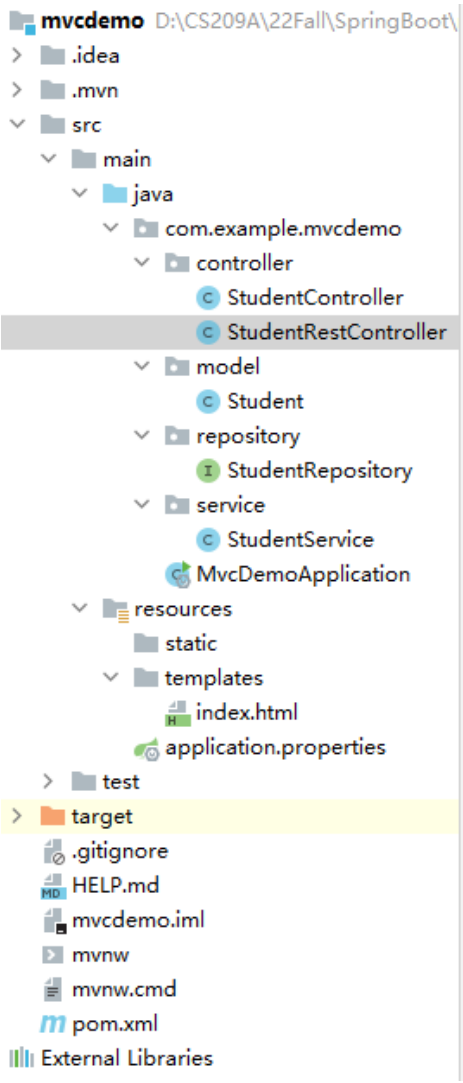



```java
7   import java.util.List;
8   import java.util.Optional;
9
10  @RestController
11  @RequestMapping("/api/students")
12  public class StudentRestController {
        4 usages
13      private final StudentService studentService;
14
15      public StudentRestController(StudentService studentService) {
16          this.studentService = studentService;
17      }
18
19      @GetMapping
20      public List<Student> getStudentsByEmail(@RequestParam(value = "email")
21                                              Optional<String> email) {
22          if (email.isPresent()){
23              return studentService.findByEmailLike(email.get());
24          }
25          return studentService.getStudents();
26      }
27
28      @PutMapping(path = "{studentId}")
29      public void updateStudent(@PathVariable("studentId") Long studentId,
30                                @RequestParam(required = false) String name,
31                                @RequestParam(required = false) String email) {
32          studentService.updateStudent(studentId, name, email);
33      }
```

RestController

`@GetMapping`: ensures that HTTP GET requests to `api/students` are mapped to the corresponding method.

`@RequestParam`: binds the value of the query string parameter `email` into the `email` parameter of this method




```java
import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/students")
public class StudentRestController {
    4 usages
    private final StudentService studentService;

    public StudentRestController(StudentService studentService) {
        this.studentService = studentService;
    }

    @GetMapping
    public List<Student> getStudentsByEmail(@RequestParam(value = "email")
                                            Optional<String> email) {
        if (email.isPresent()){
            return studentService.findByEmailLike(email.get());
        }
        return studentService.getStudents();
    }

    @PutMapping(path = "{studentId}")
    public void updateStudent(@PathVariable("studentId") Long studentId,
                              @RequestParam(required = false) String name,
                              @RequestParam(required = false) String email) {
        studentService.updateStudent(studentId, name, email);
    }
```

localhost:8080/api/students

localhost:8080/api/students


```
[
  {
    "id": 1,
    "name": "Mary",
    "email": "mary@gmail.com"
  },
  {
    "id": 2,
    "name": "Alex",
    "email": "alex@gmail.com"
  },
  {
    "id": 3,
    "name": "Dean",
    "email": "dean@yahoo.com"
  }
]
```


localhost:8080/api/students?e

localhost:8080/api/students?email=yahoo


```
[
  {
    "id": 3,
    "name": "Dean",
    "email": "dean@yahoo.com"
  }
]
```

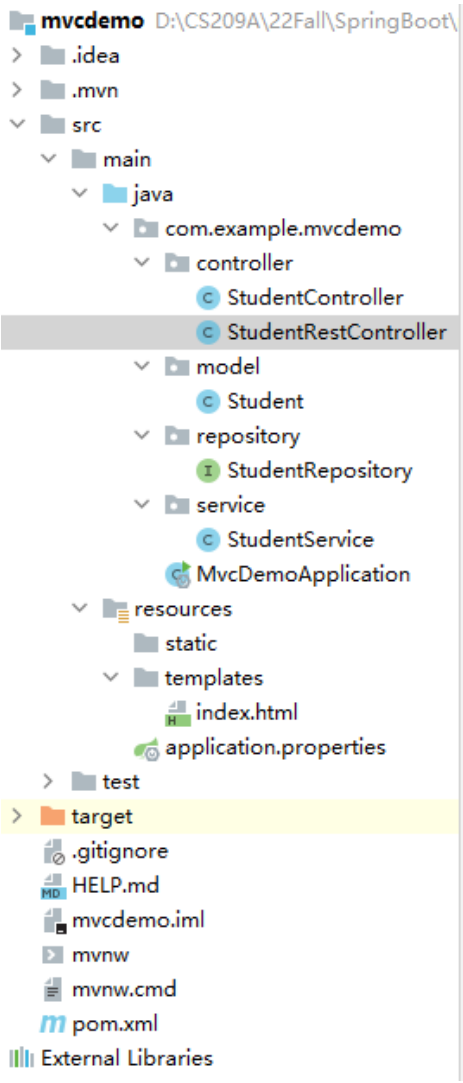

- mvcdemo D:\CS209A\22Fall\SpringBoot\
  - .idea
  - .mvn
  - src
    - main
      - java
        - com.example.mvcdemo
          - controller
            - StudentController
            - StudentRestController
          - model
            - Student
          - repository
            - StudentRepository
          - service
            - StudentService
          - MvcDemoApplication
      - resources
        - static
        - templates
          - index.html
        - application.properties
    - test
  - target
  - .gitignore
  - HELP.md
  - mvcdemo.iml
  - mvnw
  - mvnw.cmd
  - pom.xml
  - External Libraries

```java
import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/students")
public class StudentRestController {
    4 usages
    private final StudentService studentService;

    public StudentRestController(StudentService studentService) {
        this.studentService = studentService;
    }

    @GetMapping
    public List<Student> getStudentsByEmail(@RequestParam(value = "email")
                                            Optional<String> email) {
        if (email.isPresent()){
            return studentService.findByEmailLike(email.get());
        }
        return studentService.getStudents();
    }

    @PutMapping(path = "{studentId}")
    public void updateStudent(@PathVariable("studentId") Long studentId,
                              @RequestParam(required = false) String name,
                              @RequestParam(required = false) String email) {
        studentService.updateStudent(studentId, name, email);
    }
```

RestController

`@PutMapping`: maps HTTP PUT requests onto specific handler methods (shortcut for `@RequestMapping(method = RequestMethod.PUT)`)

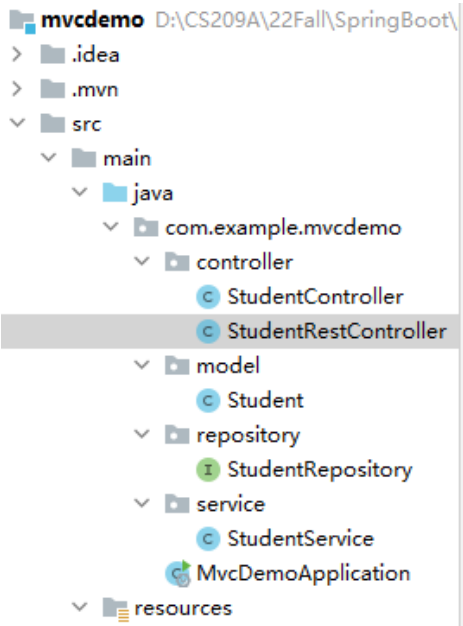
`@PathVariable`: extracts values from the URI path and binds to the `studentId` parameter




```
7   import java.util.List;
8   import java.util.Optional;
9
10  @RestController
11  @RequestMapping("/api/students")
12  public class StudentRestController {
        4 usages
13      private final StudentService studentService;
14
15      public StudentRestController(StudentService studentService) {
16          this.studentService = studentService;
17      }
18
19      @GetMapping
20      public List<Student> getStudentsByEmail(@RequestParam(value = "email")
21                                              Optional<String> email) {
22          if (email.isPresent()){
23              return studentService.findByEmailLike(email.get());
24          }
25          return studentService.getStudents();
26      }
27
28      @PutMapping(path = "{studentId}")
29      public void updateStudent(@PathVariable("studentId") Long studentId,
30                                @RequestParam(required = false) String name,
31                                @RequestParam(required = false) String email) {
32          studentService.updateStudent(studentId, name, email);
33      }
```

PUT http://localhost:8080/api/students/2?name=Alan&email=alan@gmail.com

RestController




```java
import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/students")
public class StudentRestController {
    4 usages
    private final StudentService studentService;

    public StudentRestController(StudentService studentService) {
        this.studentService = studentService;
    }

    @GetMapping
    public List<Student> getStudentsByEmail(@RequestParam(value = "email")
                                            Optional<String> email) {
        if (email.isPresent()){
            return studentService.findByEmailLike(email.get());
        }
        return studentService.getStudents();
    }

    @PutMapping(path = "{studentId}")
    public void updateStudent(@PathVariable("studentId") Long studentId,
                              @RequestParam(required = false) String name,
                              @RequestParam(required = false) String email) {
        studentService.updateStudent(studentId, name, email);
    }
```

```java
@Transactional
public void updateStudent(Long studentId, String name, String email) {
    Student s = studentRepository.findById(studentId).
            orElseThrow(()-> new IllegalStateException("Student ID not exists"));
    if(name!=null && name.length()>0 && !name.equals(s.getName())){
        s.setName(name);
    }
    if(email!=null && email.length()>0 && !email.equals(s.getEmail())){
        s.setEmail(email);
    }
}
```

PUT http://localhost:8080/api/students/2?name=Alan&email=alan@gmail.com

RestController

localhost:8080/api/students

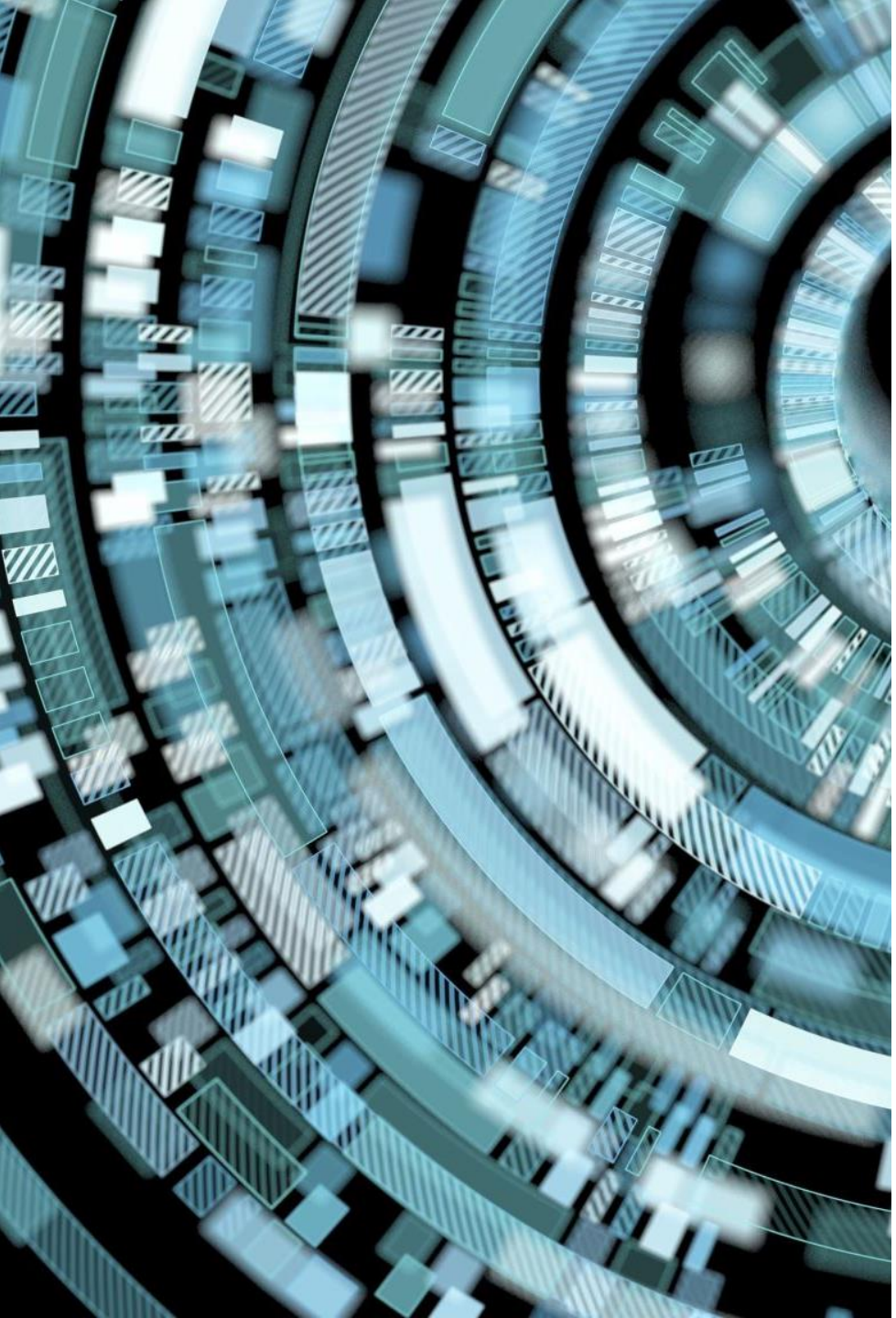
localhost:8080/api/students


```
[
  {
    "id": 1,
    "name": "Mary",
    "email": "mary@gmail.com"
  },
  {
    "id": 3,
    "name": "Dean",
    "email": "dean@yahoo.com"
  },
  {
    "id": 2,
    "name": "Alan",
    "email": "alan@gmail.com"
  }
]
```


Updated

- mvcdemo D:\CS209A\22Fall\SpringBoot\
  - .idea
  - .mvn
  - src
    - main
      - java
        - com.example.mvcdemo
          - controller
            - StudentController
            - StudentRestController
          - model
            - Student
          - repository
            - StudentRepository
          - service
            - StudentService
          - MvcDemoApplication
      - resources
        - static
        - templates
          - index.html
        - application.properties
    - test
  - target
  - .gitignore
  - HELP.md
  - mvcdemo.iml
  - mvnw
  - mvnw.cmd
  - pom.xml
  - External Libraries

```
7   import java.util.List;
8   import java.util.Optional;
9
10  @RestController
11  @RequestMapping("/api/students")
12  public class StudentRestController {
        4 usages
13      private final StudentService studentService;
14
15      public StudentRestController(StudentService studentService) {
16          this.studentService = studentService;
17      }
18
19      @GetMapping
20      public List<Student> getStudentsByEmail(@RequestParam(value = "email")
21                                              Optional<String> email) {
22          if (email.isPresent()){
23              return studentService.findByEmailLike(email.get());
24          }
25          return studentService.getStudents();
26      }
27
28      @PutMapping(path = "{studentId}")
29      public void updateStudent(@PathVariable("studentId") Long studentId,
30                                @RequestParam(required = false) String name,
31                                @RequestParam(required = false) String email) {
32          studentService.updateStudent(studentId, name, email);
33      }
```



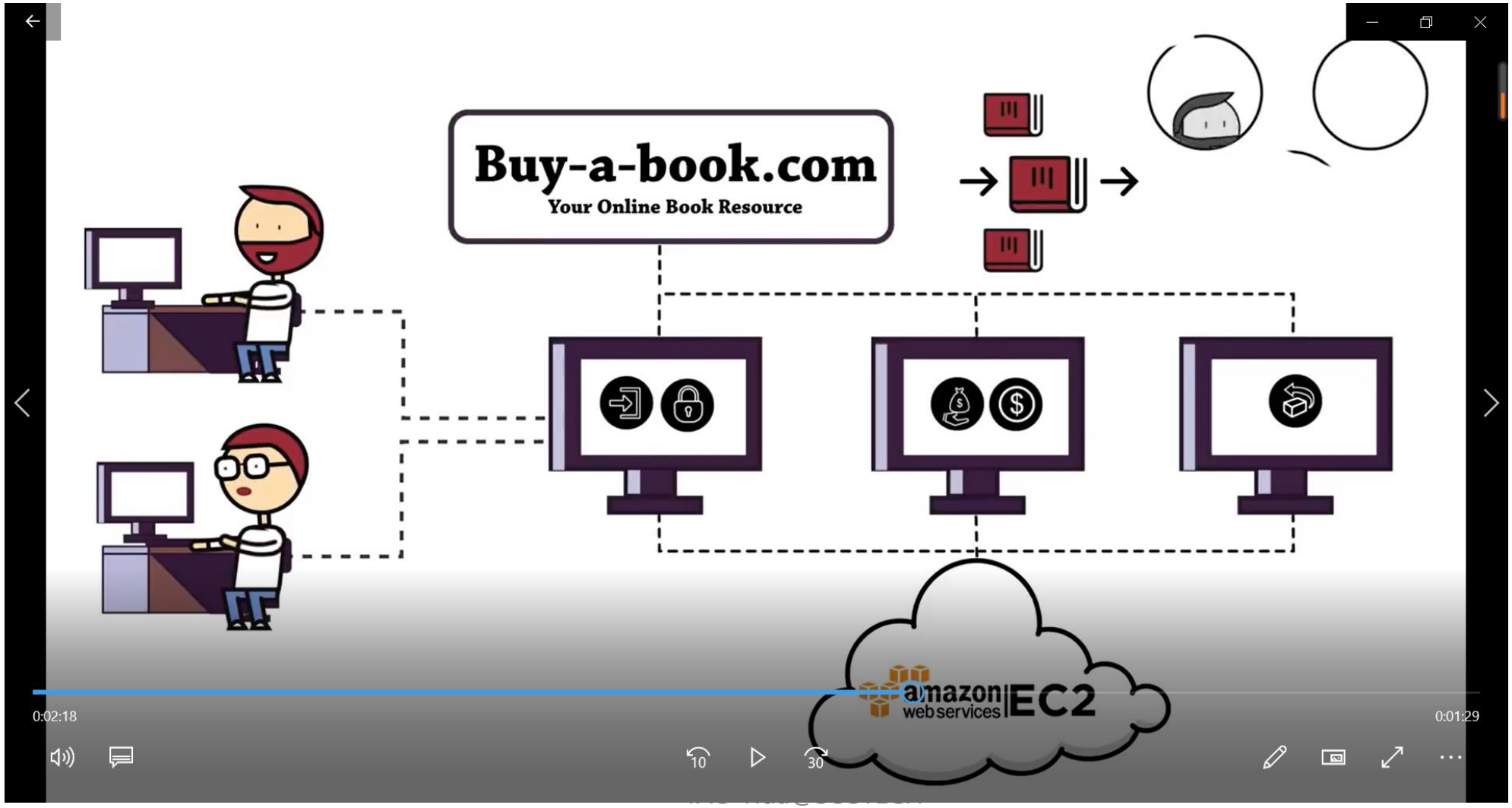
Lecture 12

- The Spring Framework
  - IoC & Dependency Injection
  - Spring AOP
  - Spring MVC
- Spring Boot
  - Overview
  - Building a MVC web application
  - Building a RESTful web service
  - Microservices

Java Microservices

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms (e.g., RESTful API)
- Spring Boot has become the de facto standard for Java™ microservices

Microservices



The Spring Ecosystem

Next Lecture

- Testing
- Logging