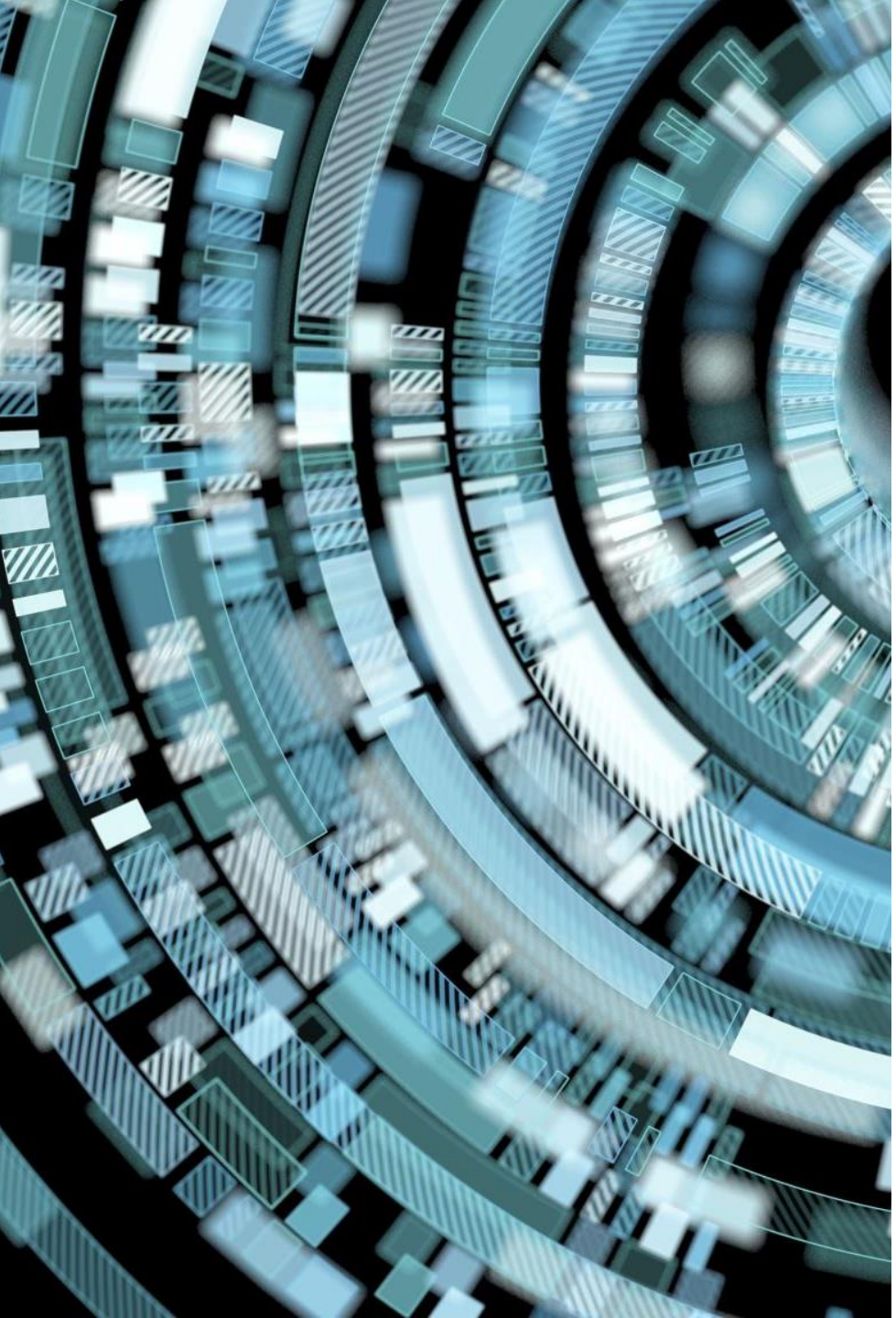
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn

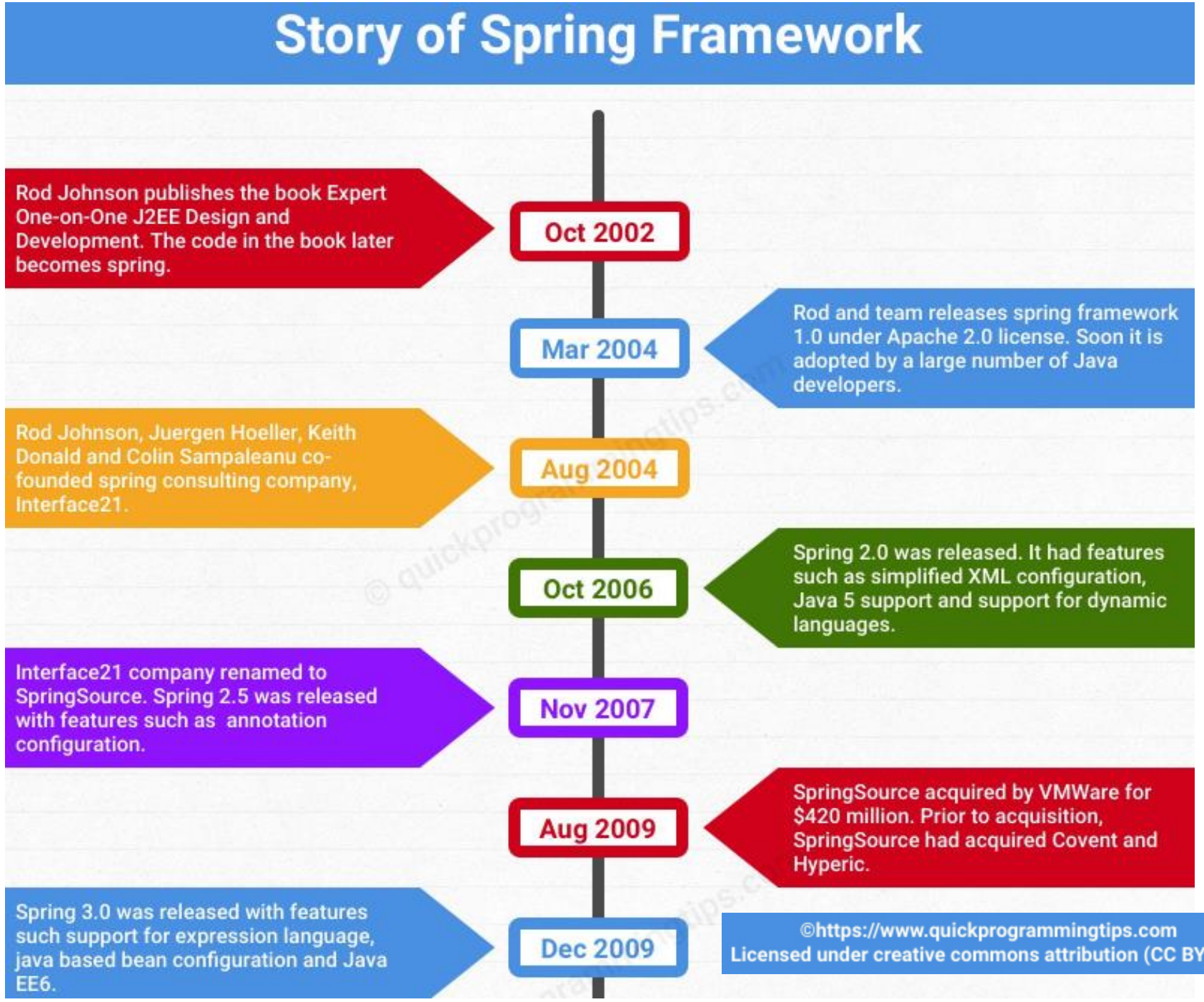


Lecture 13

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

- This book covered the state of Java enterprise application development at the time and pointed out a number of major deficiencies with Java EE and EJB component framework.
- The book proposed a simpler solution based on POJO and **dependency injection**
- The book shows a high quality, scalable online seat reservation application can be built without using EJB. For building the application, Rod wrote over 30,000 lines of infrastructure code! It included a number of reusable java interfaces and classes such as ApplicationContext and BeanFactory
- The book is an instant hit. Much of the infrastructure code freely provided as part of the book was highly reusable and soon a number of developers started using it in their projects

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

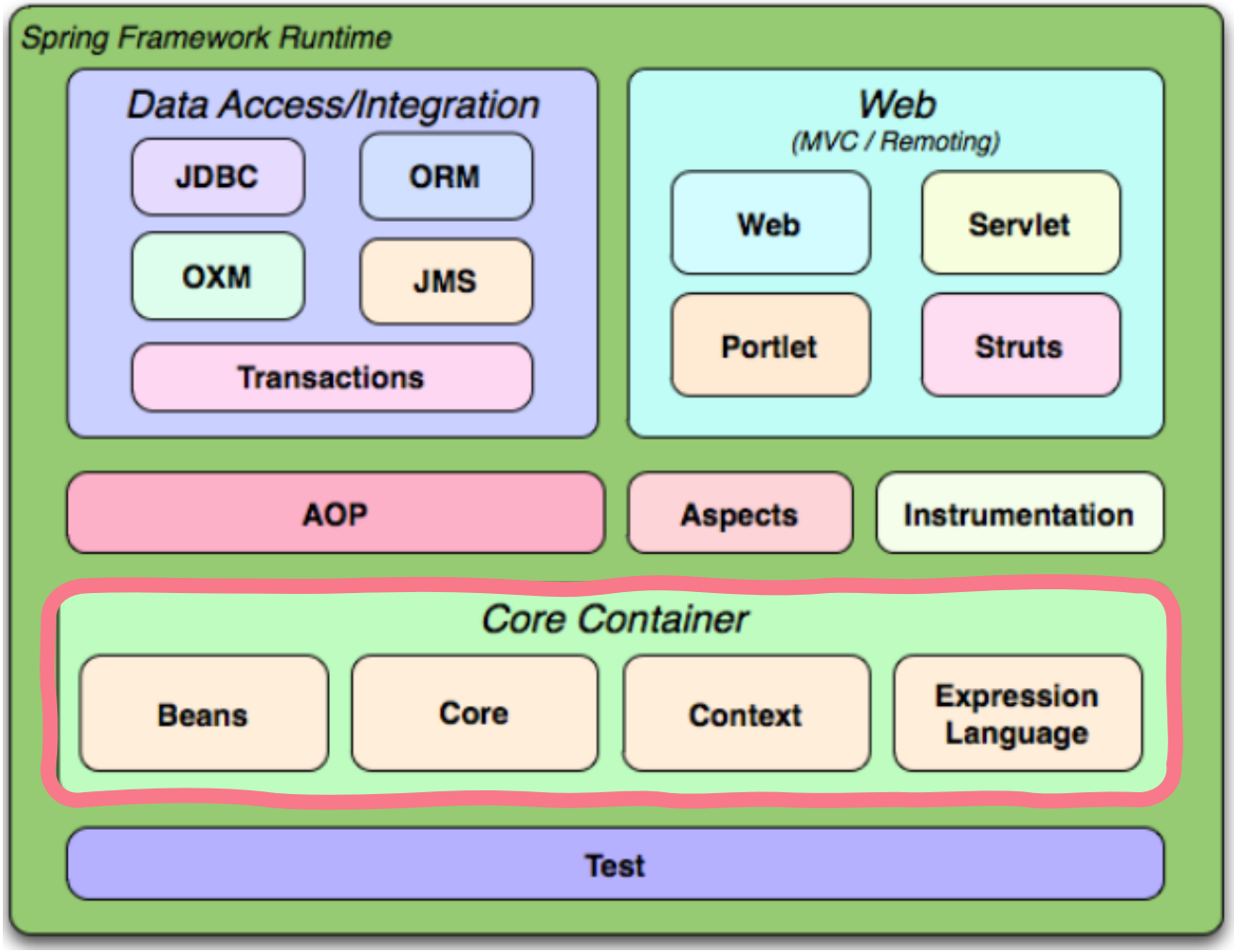


The Spring Framework

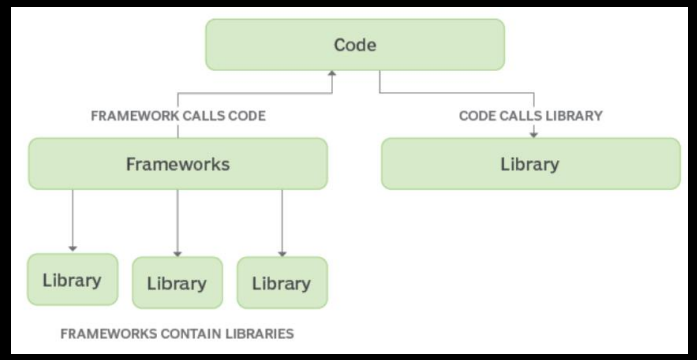
- The Spring Framework is an open-source, lightweight framework that enables developers to develop enterprise-class applications using Plain Old Java Object, POJO, instead of EJB
- It also offers tons of extensions that are used for building all sorts of large-scale applications on top of the Java EE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

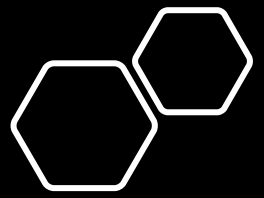


Core Concepts in Spring



- 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): a design pattern used to implement IoC.
- Dependency injection makes a class independent of its dependencies. It achieves that by decoupling (解耦) the usage of an object from its creation (e.g., by the Spring IoC container).
- DI aims to separate the concerns of constructing objects and using them, 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, and hard to test.

https://stackoverflow.com/a/6085922

```
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:
We are injecting the dependencies (Wheel and Battery) at runtime.
DI can be done by setter injection or constructor injection.
DI is configured in Spring's config file

Configurations

- @Configuration
  - A Java class annotated with @Configuration is a configuration by itself
  - Classes with @Configuration define and instantiate beans
- @Bean
  - @Bean annotation works with @Configuration to create Spring beans.
  - Methods annotated with @Bean create and return the actual bean

```
@Configuration
@ComponentScan("com.baeldung.constructordi")
public class Config {

    @Bean
    public Engine engine() {
        return new Engine("v8", 5);
    }

    @Bean
    public Transmission transmission() {
        return new Transmission("sliding");
    }
}
```

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

Beans

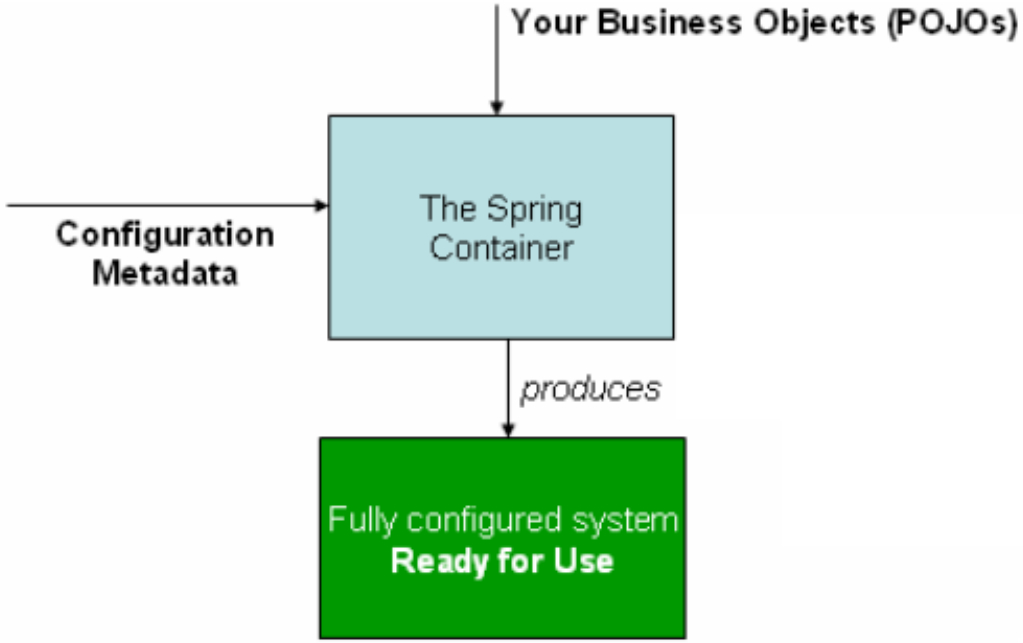
- A bean is simply one of many objects in your Spring application
- A bean is an object that is instantiated, assembled, and managed by a Spring IoC container
- Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

```java
@Configuration
@ComponentScan("com.baeldung.constructordi")
public class Config {

    @Bean
    public Engine engine() {
        return new Engine("v8", 5);
    }

    @Bean
    public Transmission transmission() {
        return new Transmission("sliding");
    }
}
```

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

Spring Core Container (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.



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

Spring Core Container (IoC Container)

- `@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 {

    @Autowired
    public Car(Engine engine, Transmission transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }
}
```

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

Spring Core Container (IoC Container)

- `@Autowired` can be applied on fields (bad practice), 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 {

    @Autowired
    public Car(Engine engine, Transmission transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }
}
```

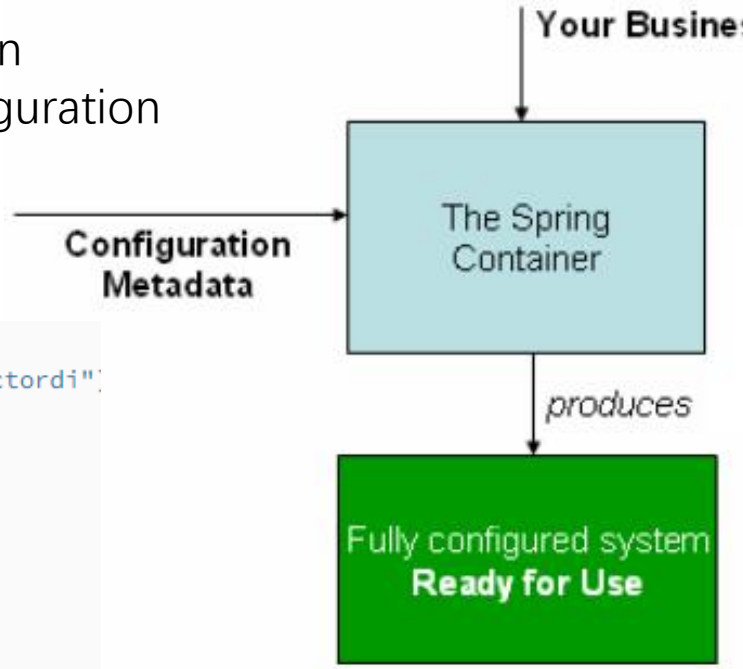
https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html

To Put it Together

```
ApplicationContext context = new AnnotationConfigApplicationContext(Config.class);
Car car = context.getBean(Car.class);
```

Configuration could be:

- XML-based configuration
- Annotation-based configuration




```
@Configuration
@ComponentScan("com.baeldung.constructordi")
public class Config {

    @Bean
    public Engine engine() {
        return new Engine("v8", 5);
    }

    @Bean
    public Transmission transmission() {
        return new Transmission("sliding");
    }
}
```

```
@Component
public class Car {

    @Autowired
    public Car(Engine engine, Transmission transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }
}
```

Spring container will:

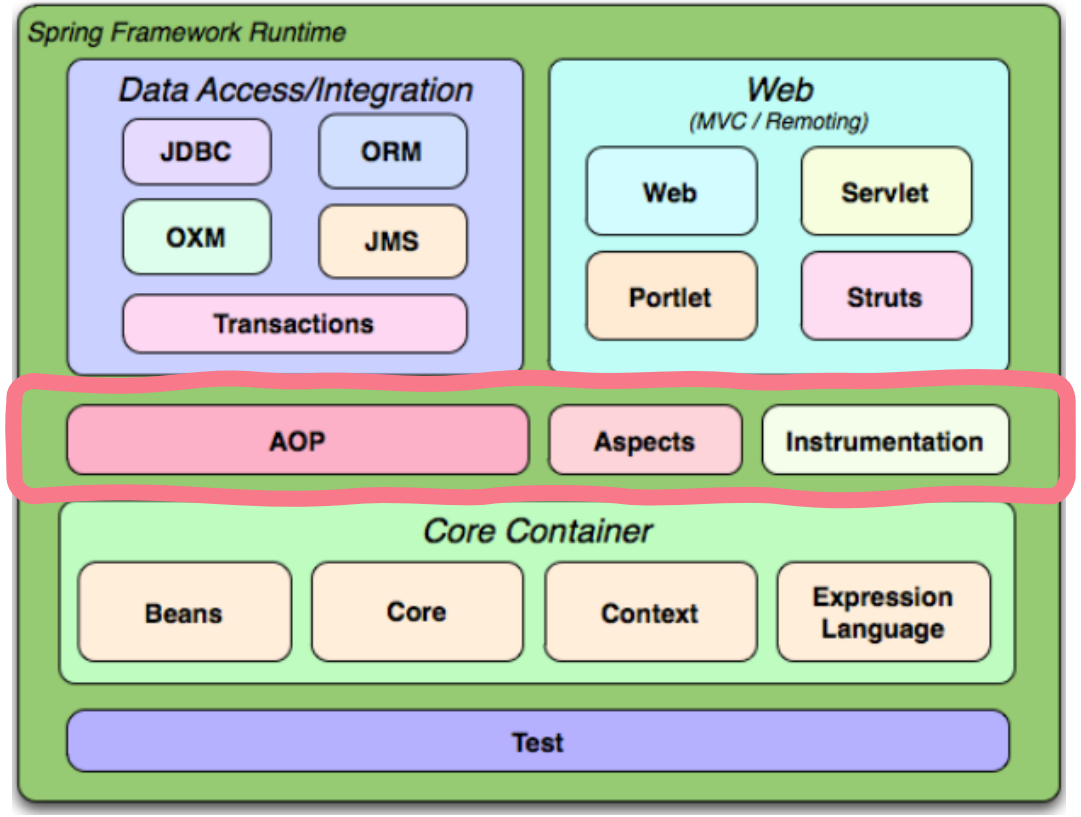
- Scan our application for classes annotated with @Component
- Instantiate them and inject any specified dependencies into them
- Inject them wherever needed

https://www.baeldung.com/constructor-injection-in-spring

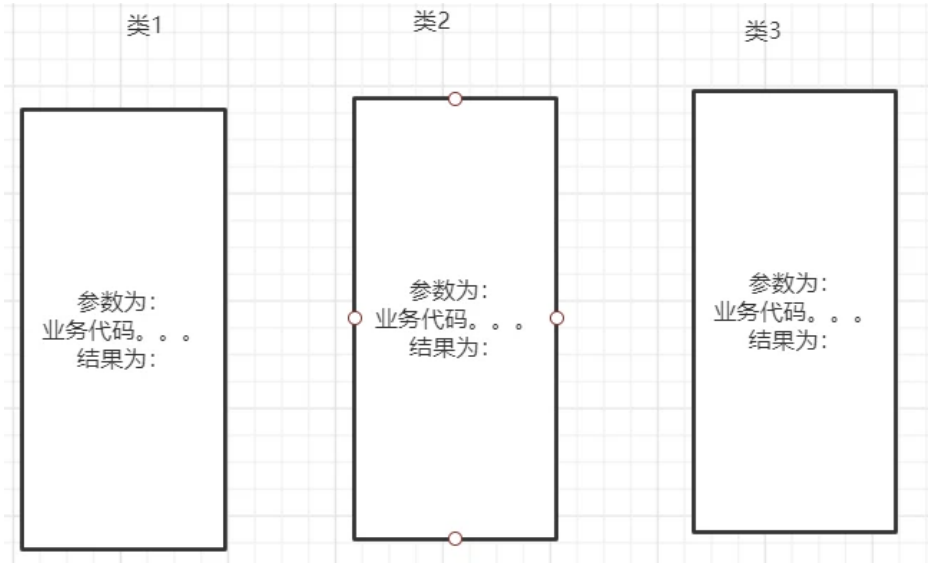
Spring AOP

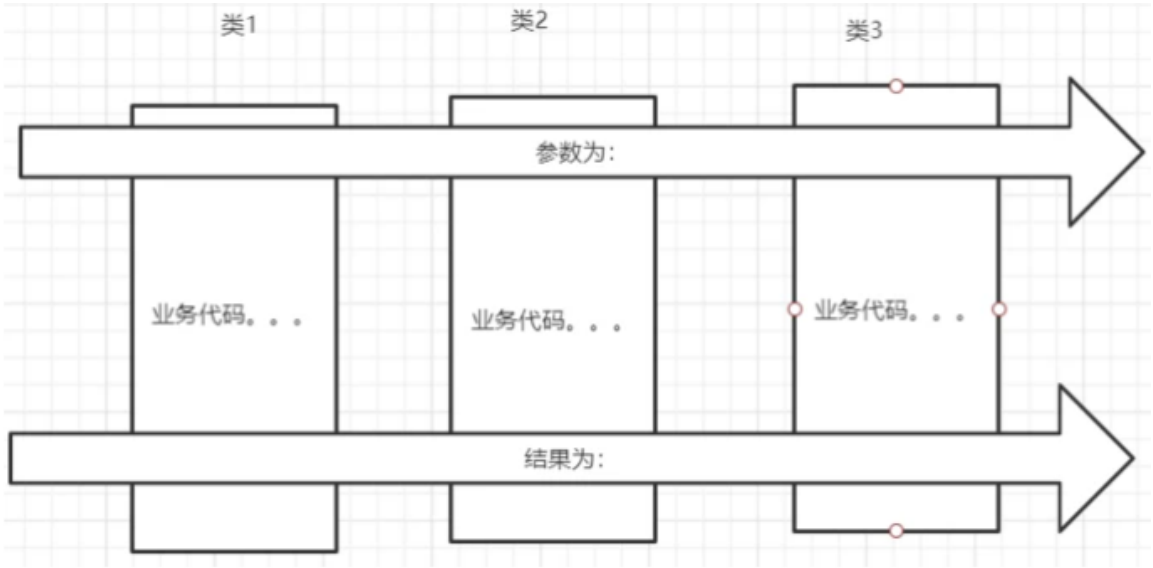
- AOP (Aspect-Oriented Programming, 面向切面编程): a programming paradigm that complements OOP by allowing the separation of cross-cutting concerns (i.e., we could add additional cross-cutting behavior to existing code without modifying the code itself)

- Cross-cutting concerns: a piece of logic or code that is going to be written in multiple classes/layers but is not business logic
  - Logging
  - Exception handling
  - Security aspects
  - Transaction management
  - …



Spring AOP



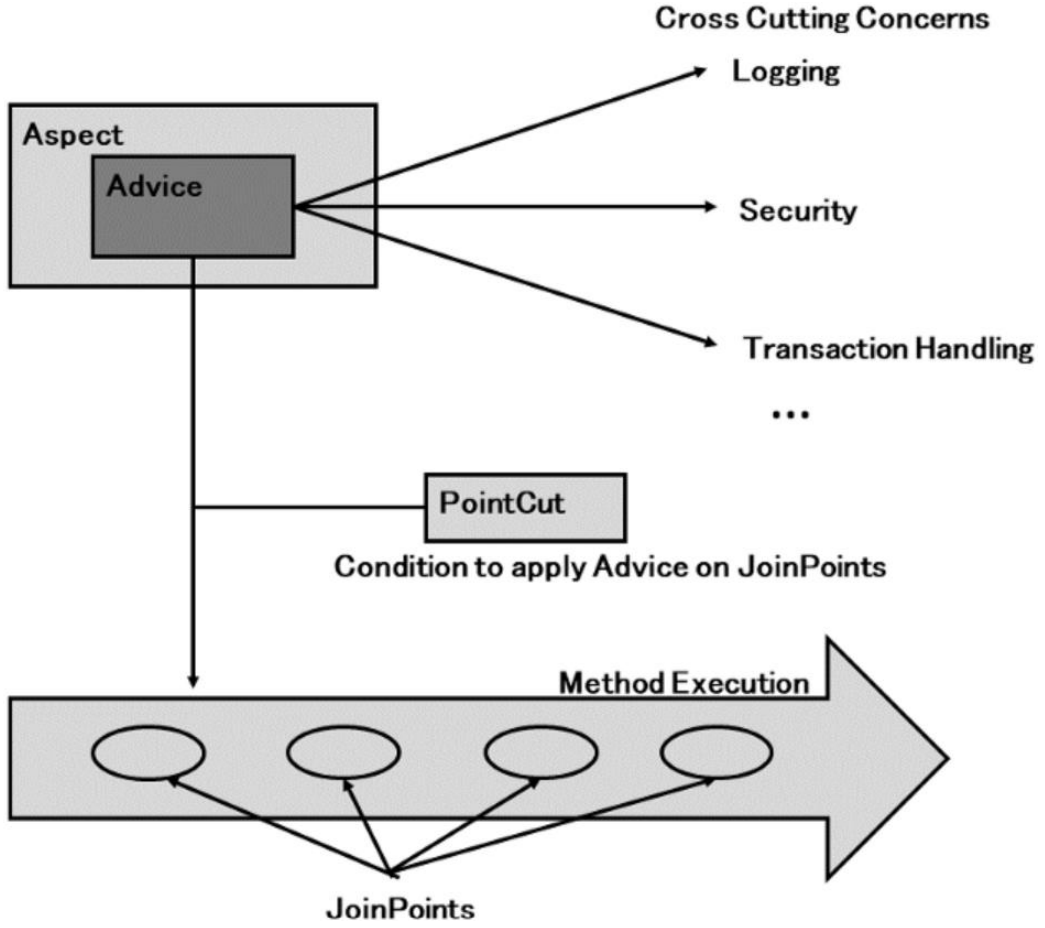


Without AOP: business code and non-business code are tangled together

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)

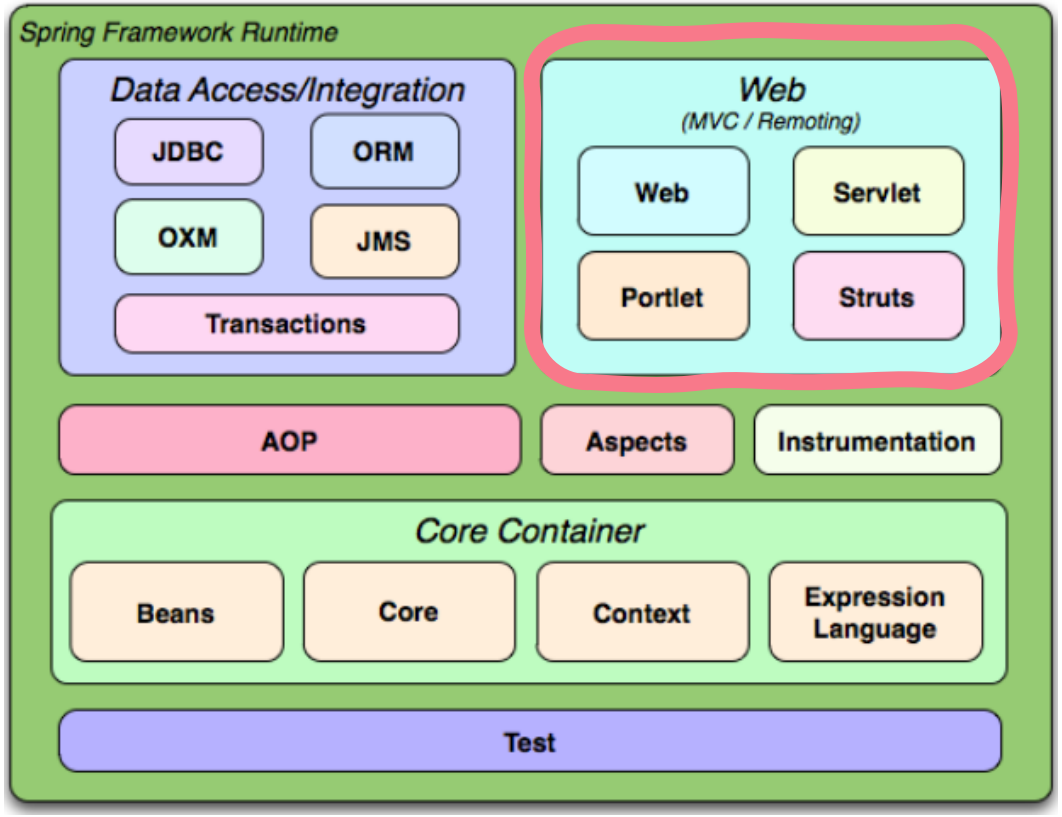
Pointcut: expressions to match joint-point methods

Before Advice: action taken at a join point

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

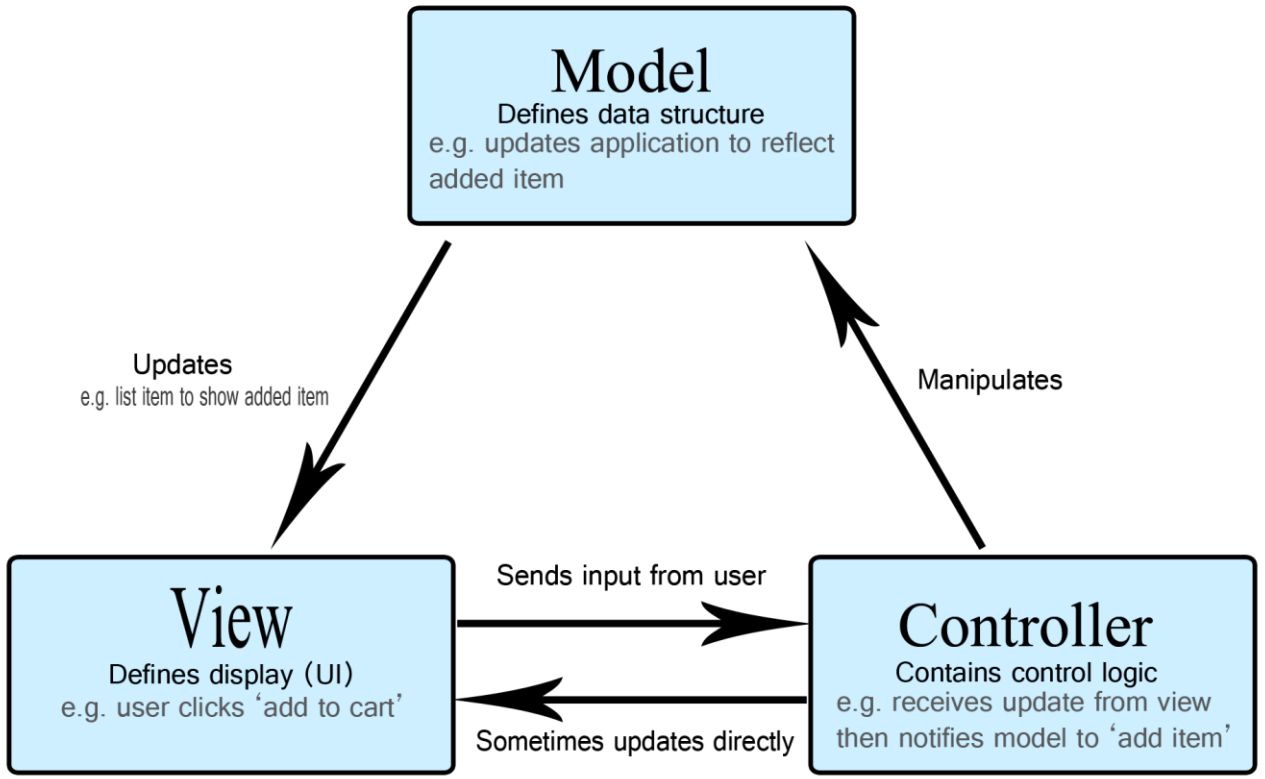
Spring MVC

- 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)
- The Web layer consists of the `spring-web`, `spring-webmvc`, `spring-websocket`, and `spring-webmvc-portlet` modules.
- Spring MVC (`spring-webmvc`) contains Spring's model-view-controller (MVC) and REST Web Services implementation for web applications.



MVC Design Pattern

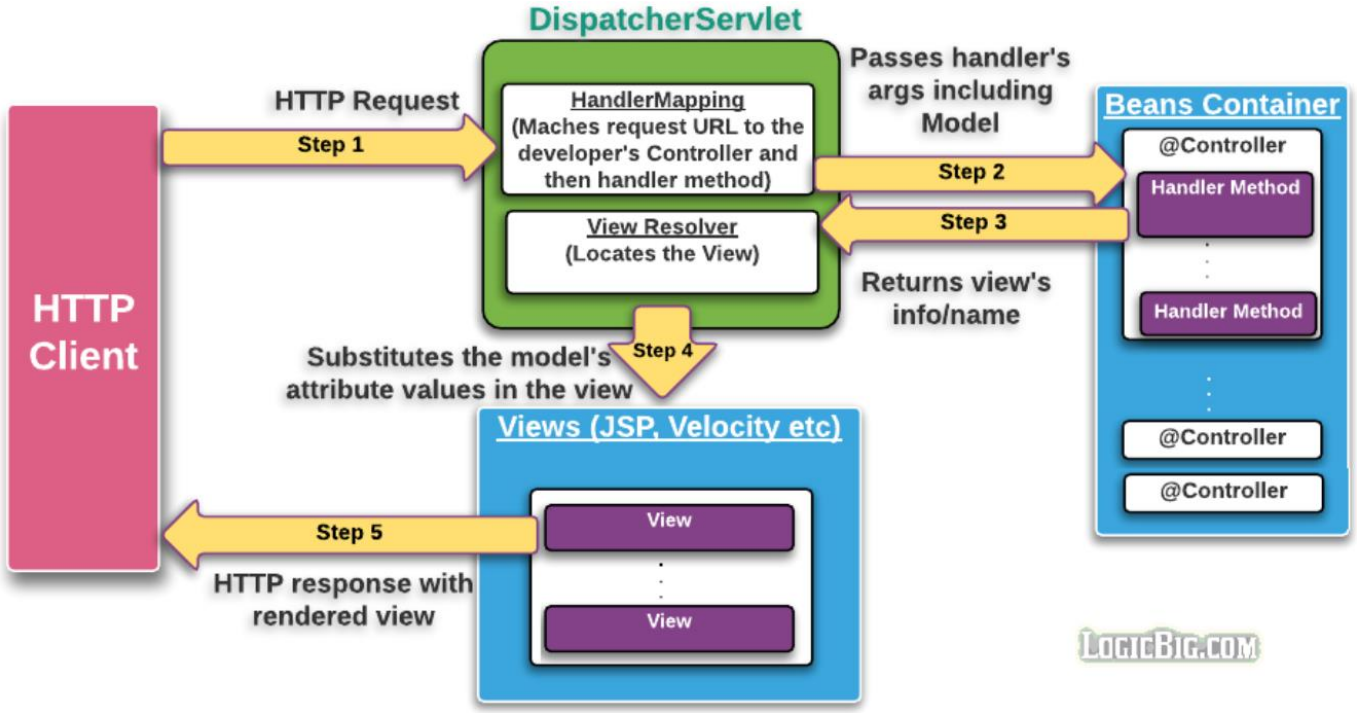
- Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements.
  - **Model** directly manages the data, logic and rules of the application
  - **View** represents the visualization of the 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`.

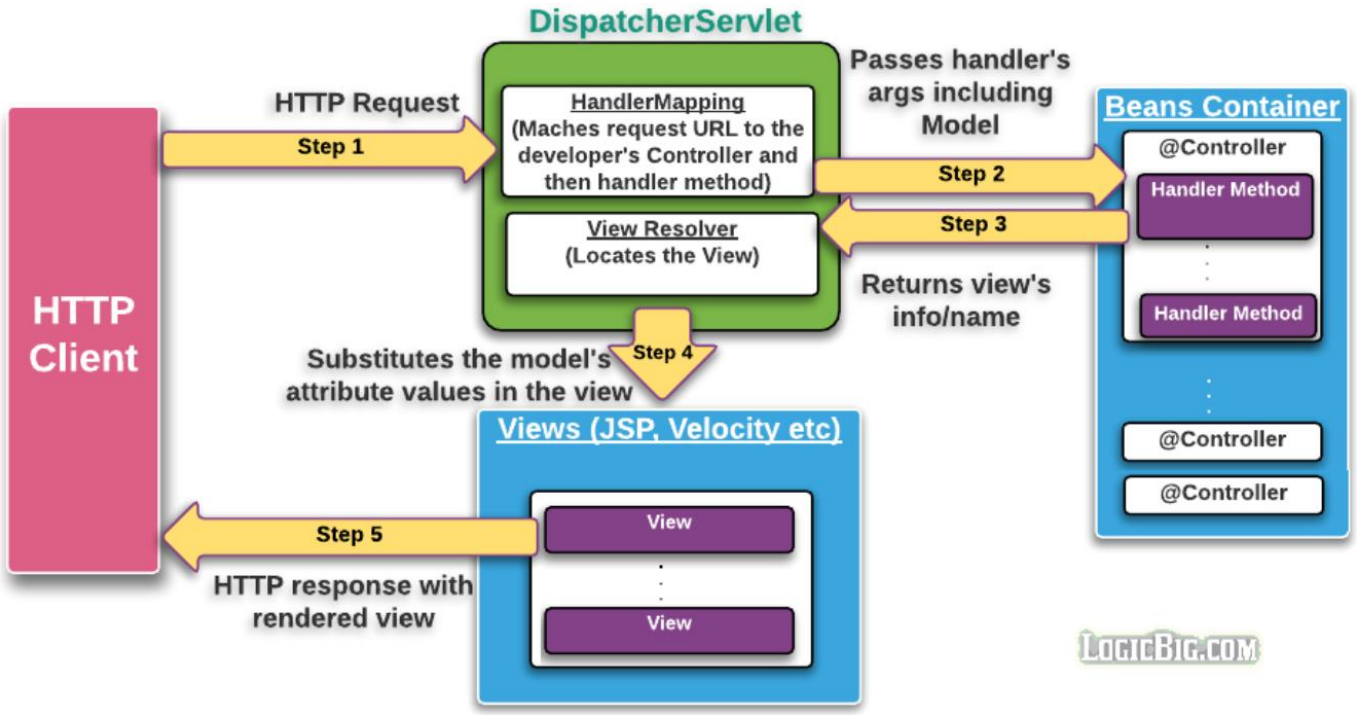


https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/spring-mvc-intro.html

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";
    }
}
```

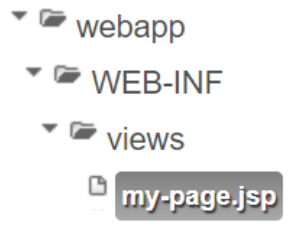


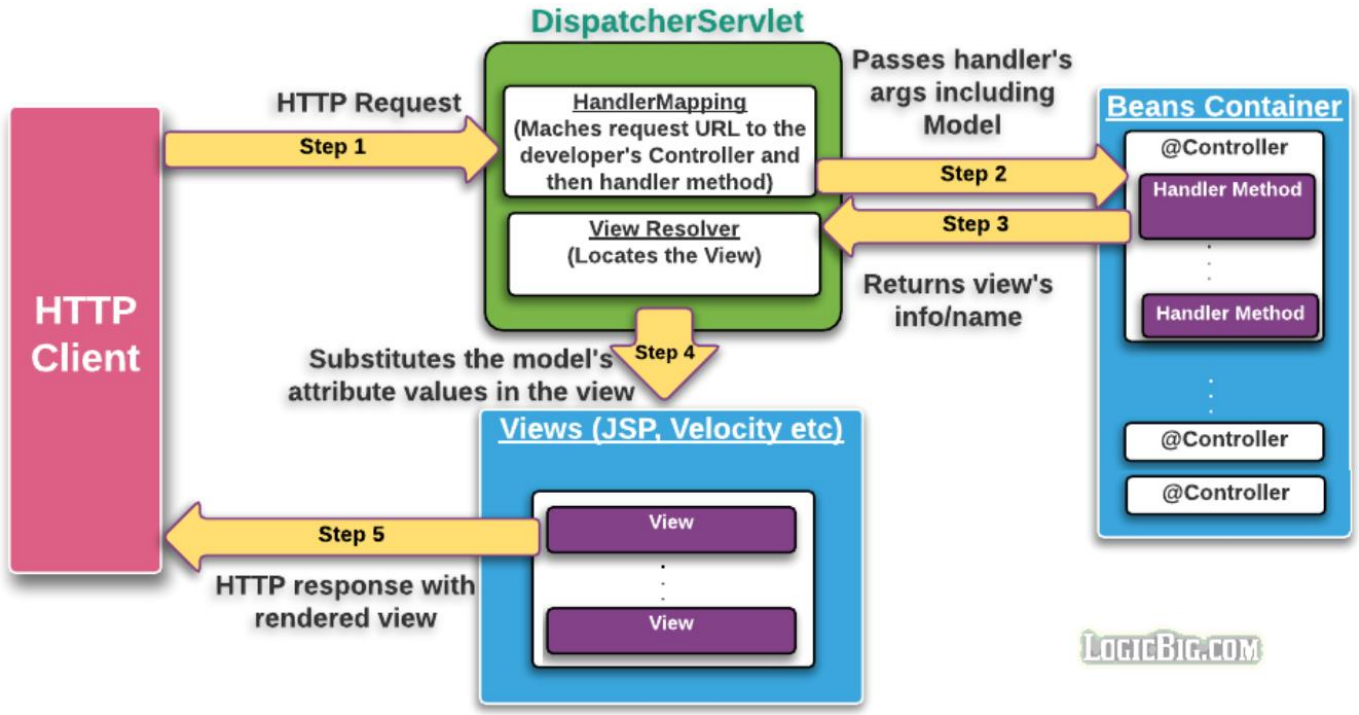
https://www.logicbig.com/tutorials/spring-framework/spring-web-mvc/spring-mvc-intro.html

Spring MVC Workflow – The View

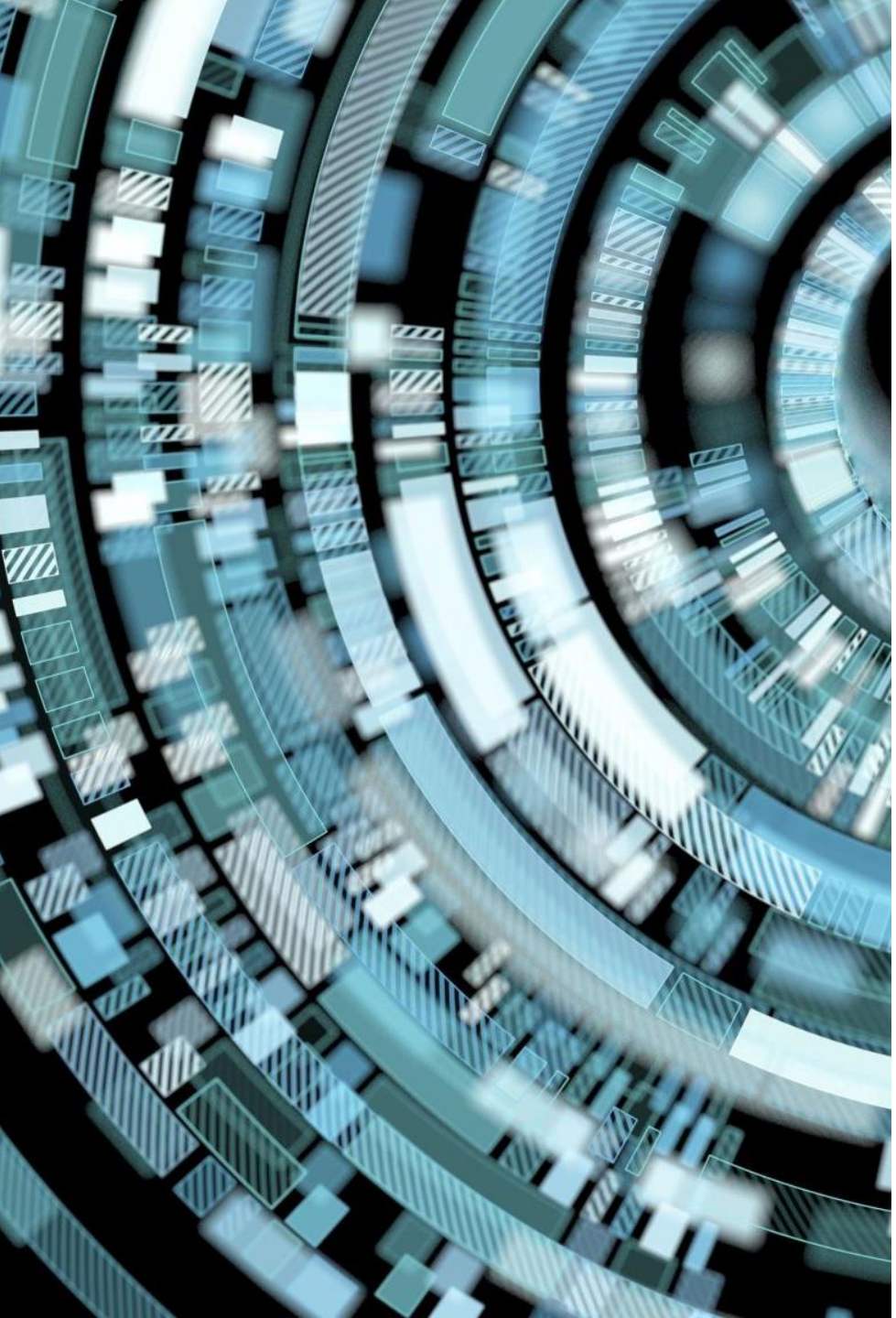
```java
@Configuration
public class MyWebConfig {

    @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>
```





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



Lecture 13

- 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,

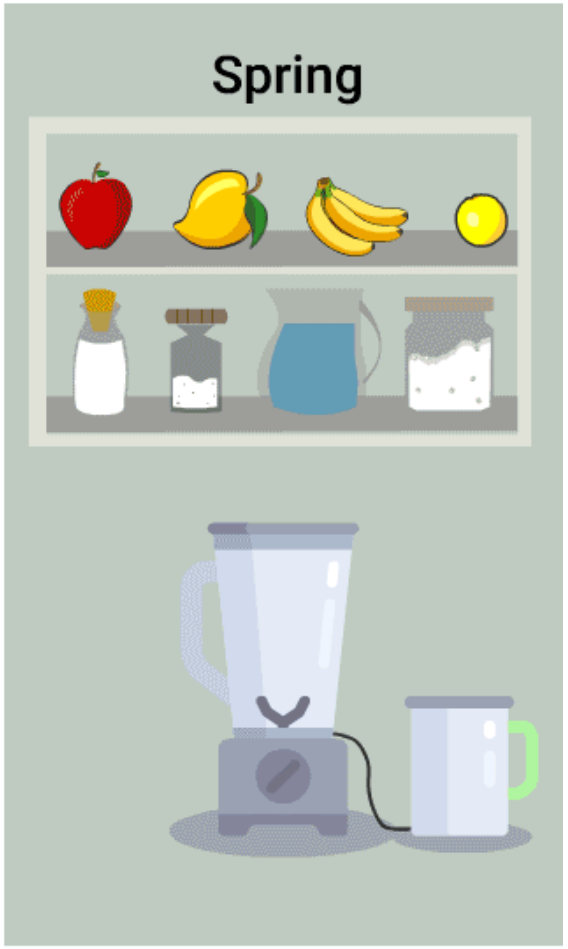
> *I think that Spring's web application architecture can be significantly simplified if it were to provided tools and a reference architecture that leveraged the Spring component and configuration model from top to bottom. Embedding and unifying the configuration of those common web container services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

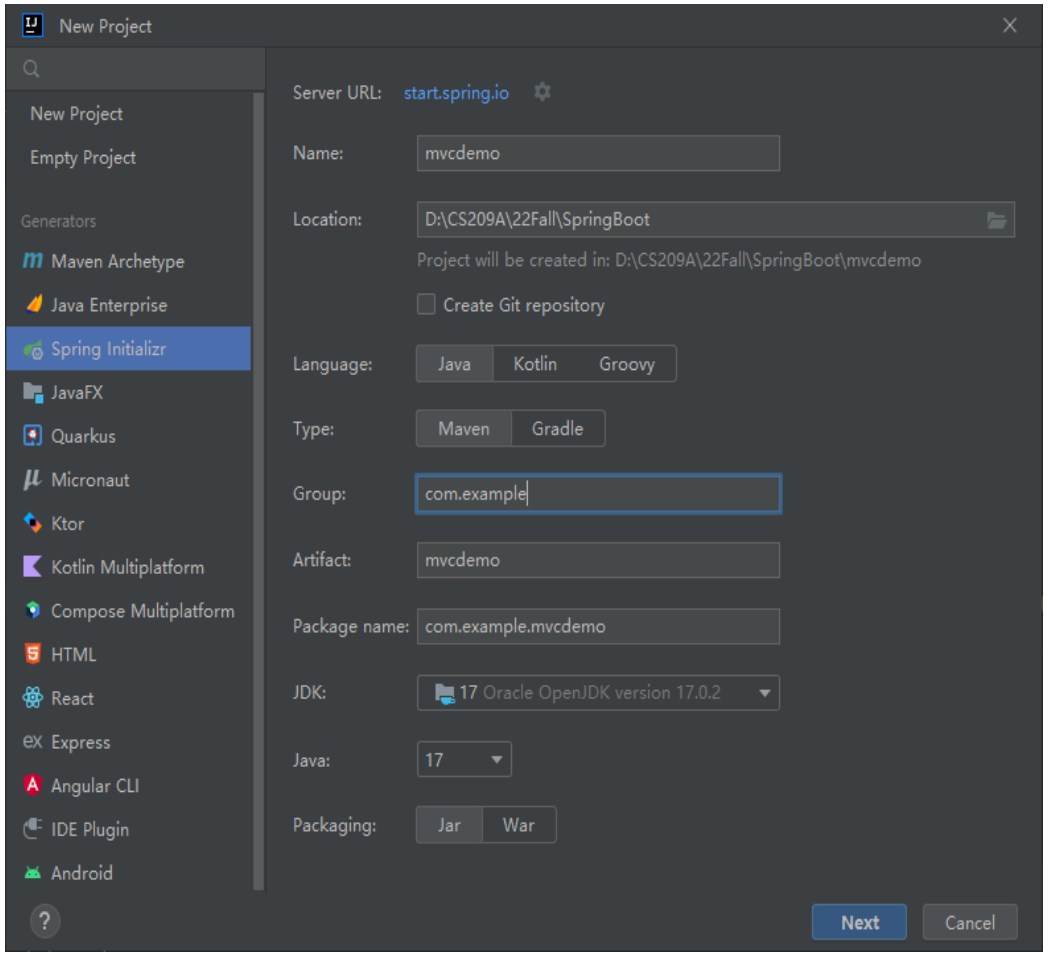
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using.auto-configuration

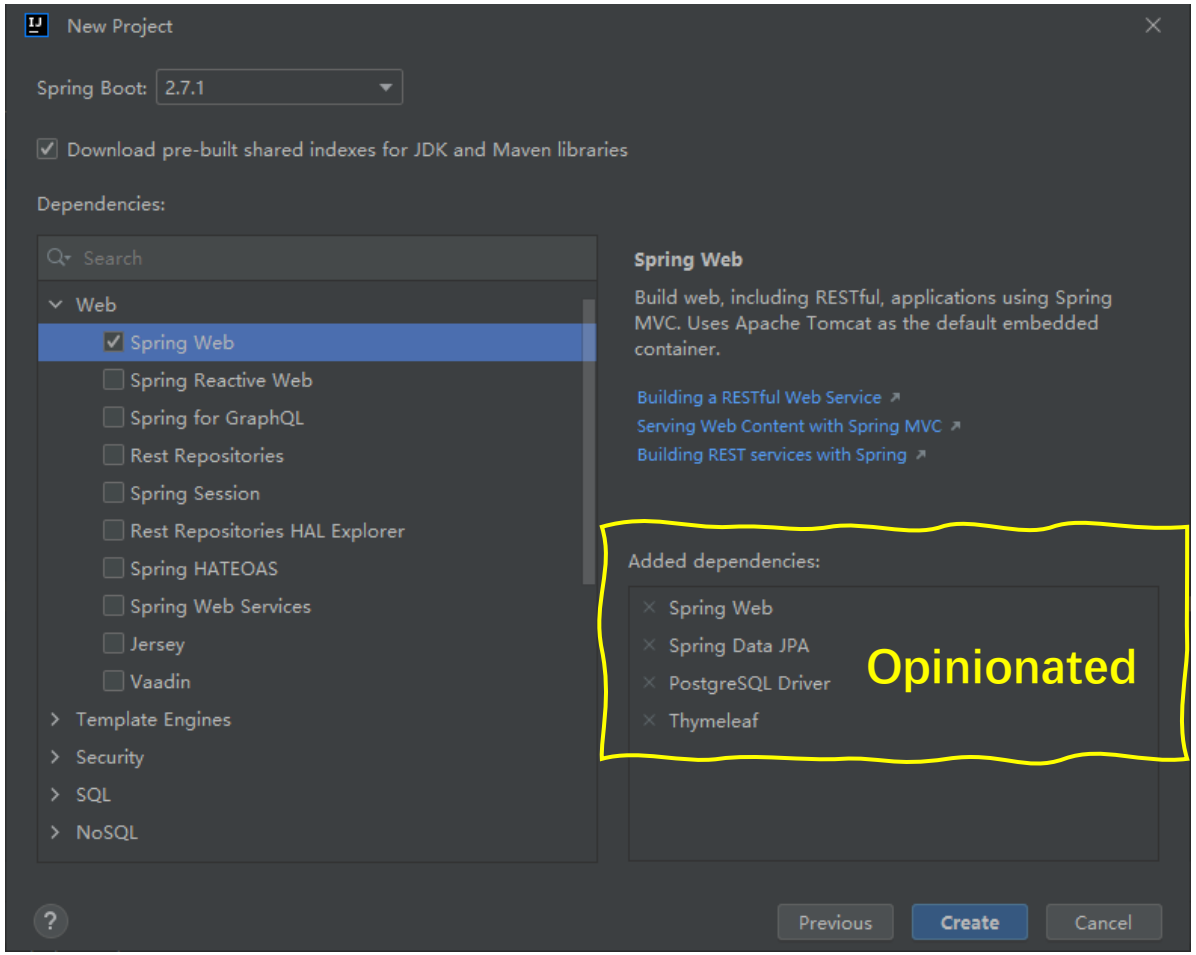
Creating a web application

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

- Using `Spring MVC`
  - Download and configure Tomcat
  - Manually add maven dependencies
    - spring-core
    - spring-context
    - spring-aop
    - spring-webmvc
    - spring-web
    - …
  - Configurations
  - More configurations
  - …

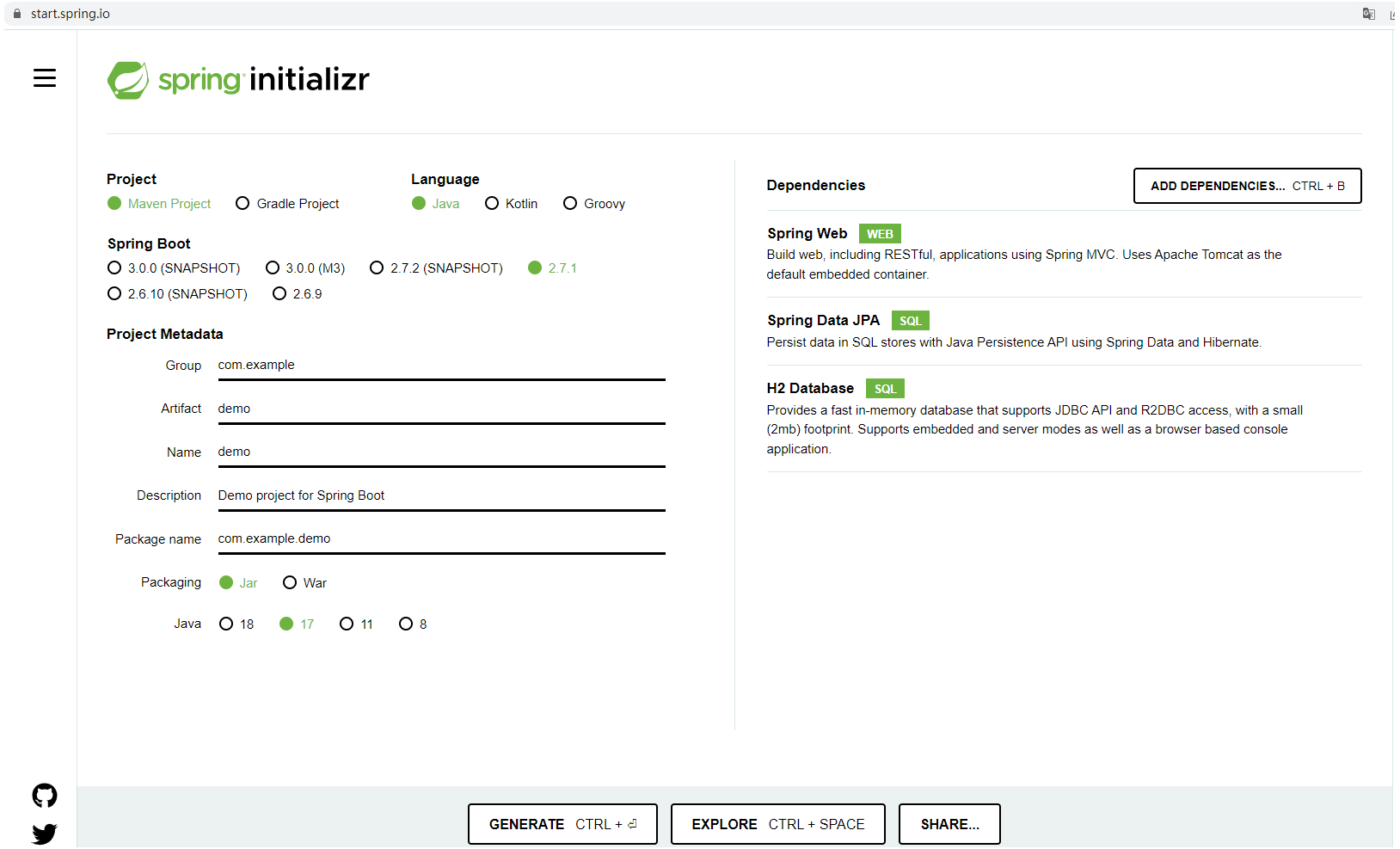
Creating a web app with Spring Initializer





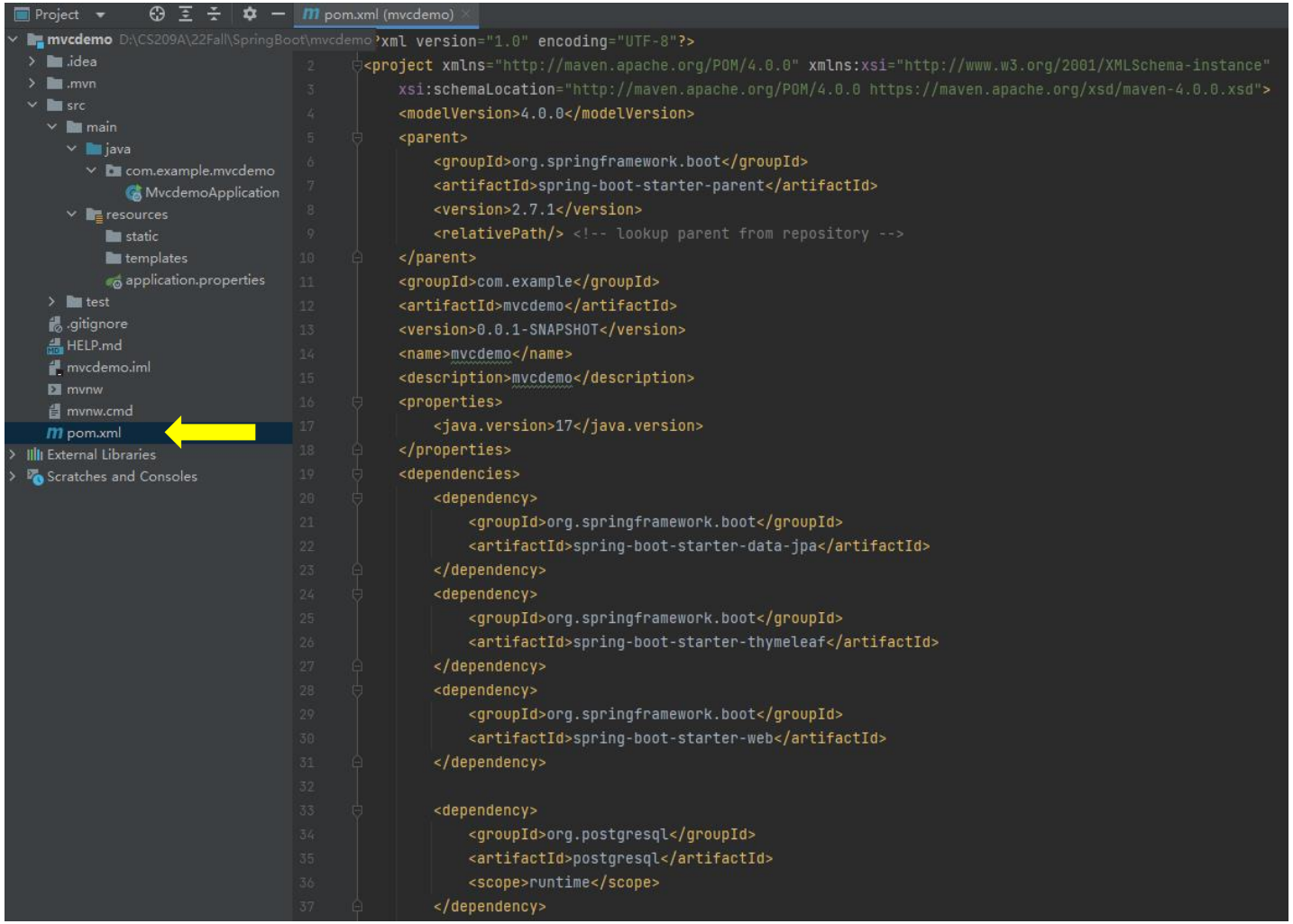
Supported by IntelliJ Ultimate

Creating a web app with Spring Initializer



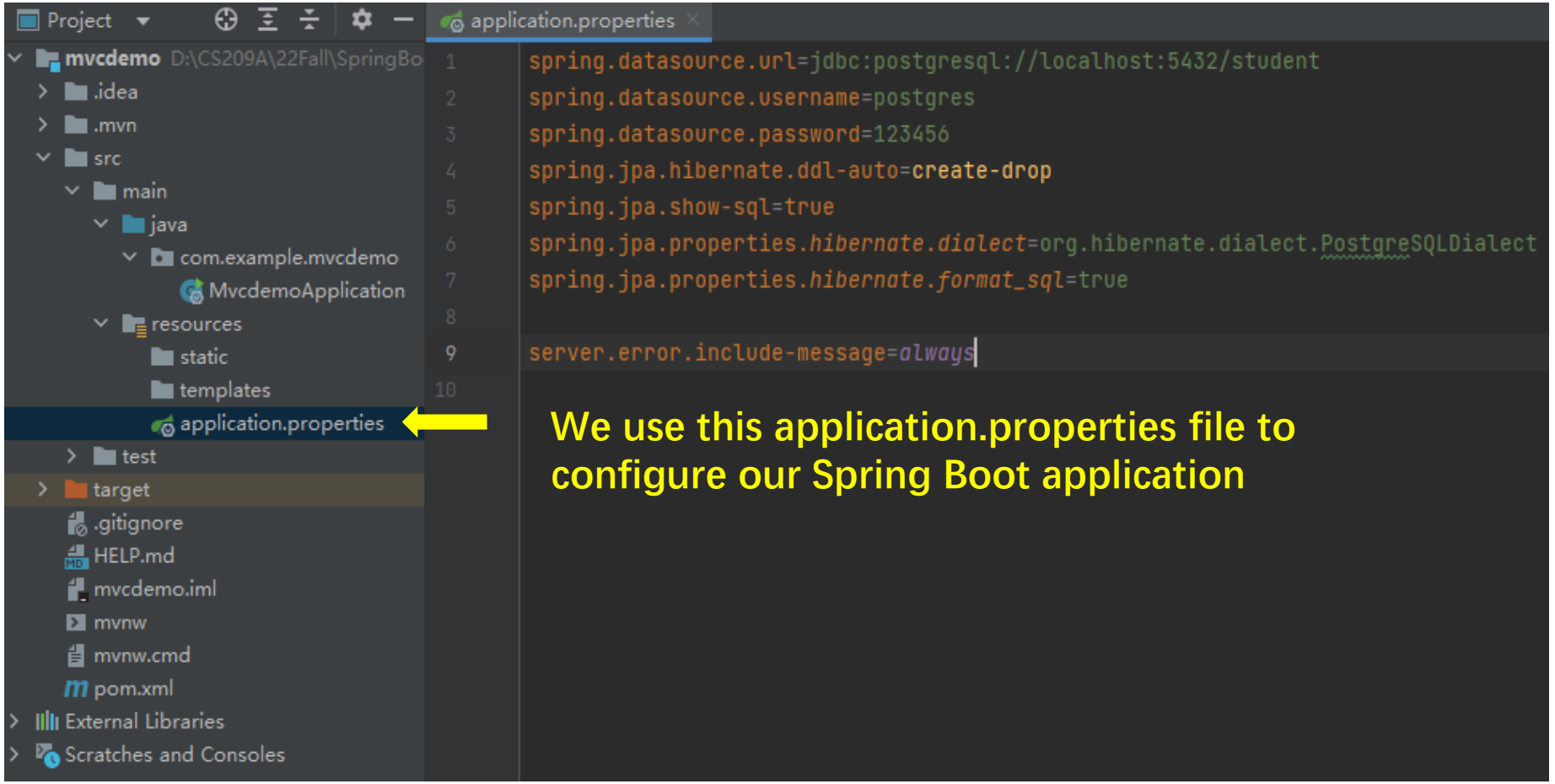
Generate & download the project, then open in IntelliJ

Maven Dependencies

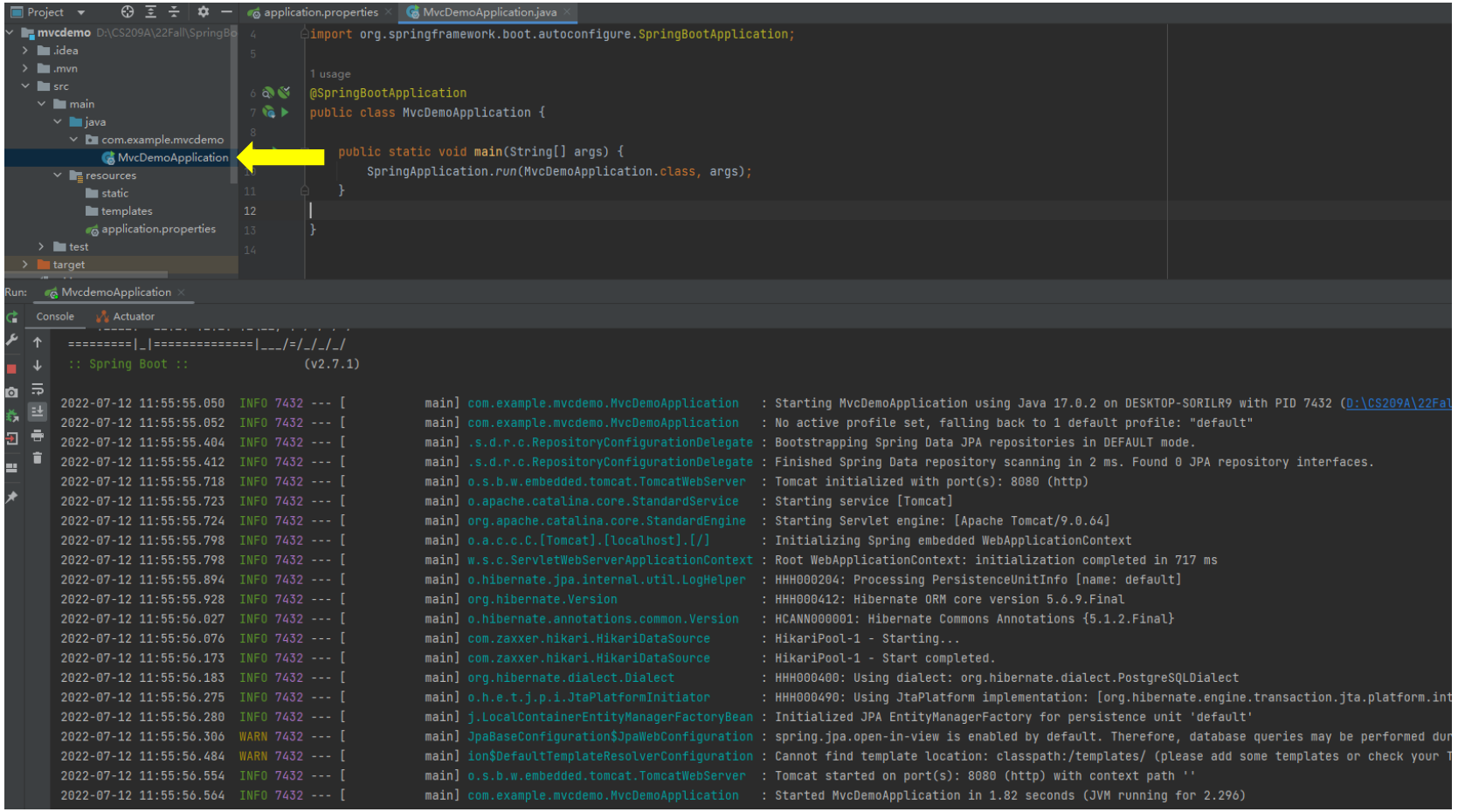


Application Properties

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



Application Class



`@SpringBootApplication` annotation enables 3 features:

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

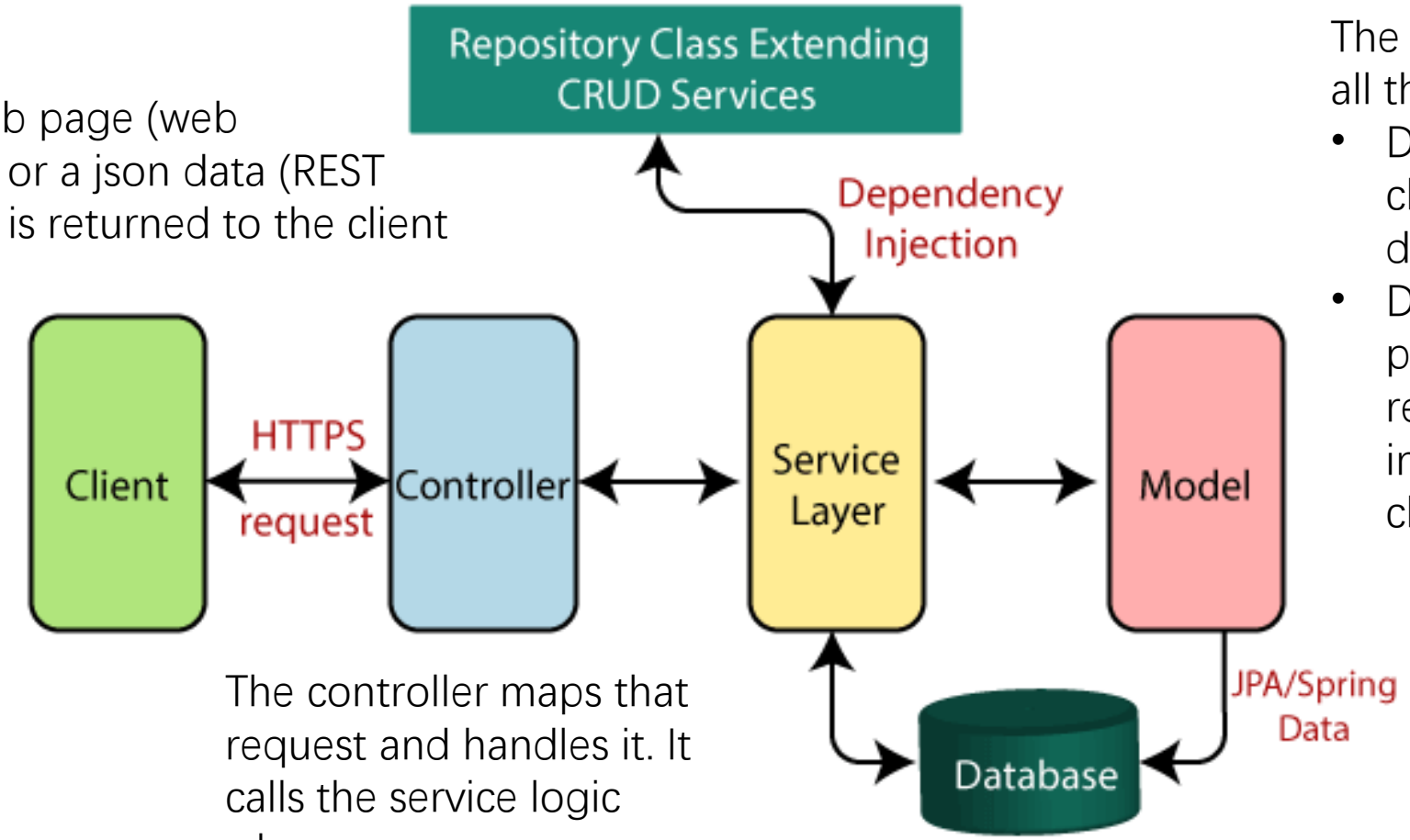
Convention over Configuration

- 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

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

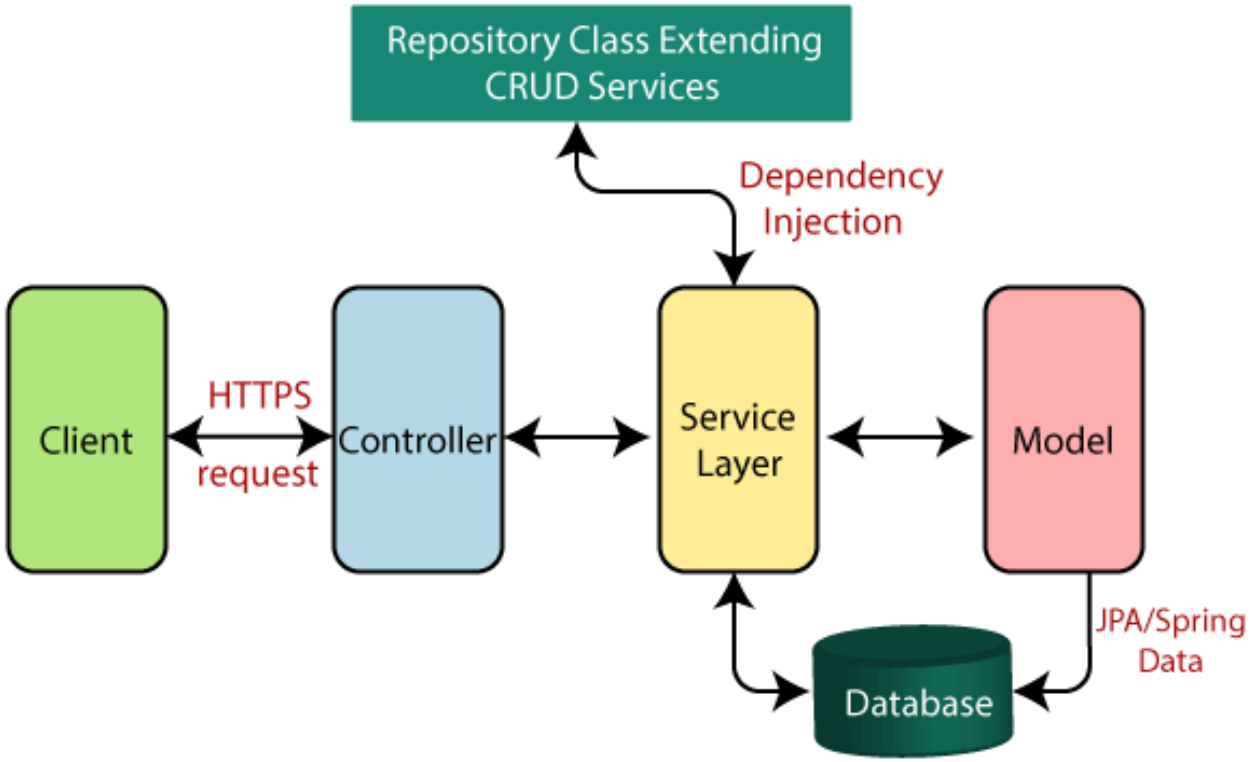


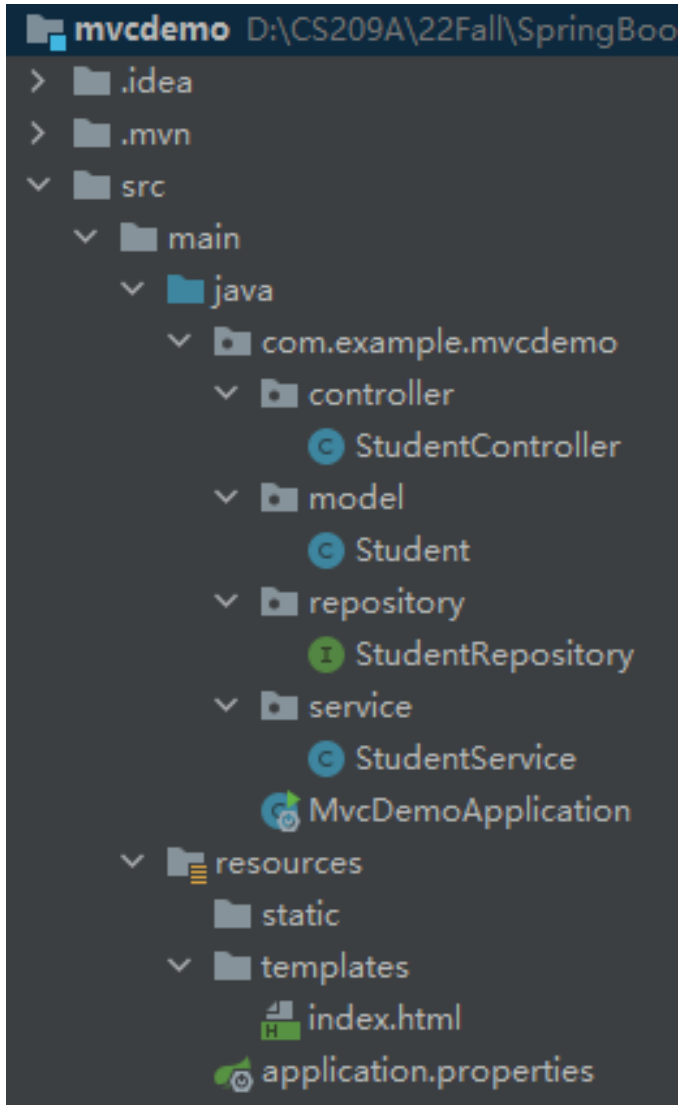
The controller maps that request and handles it. It calls the service logic when necessary.

The service layer performs all the business logic

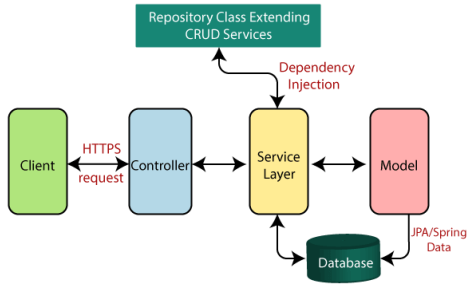
- Data is the model classes mapped to database by JPA
- Data operations are performed through a repository class, which is injected to the service class

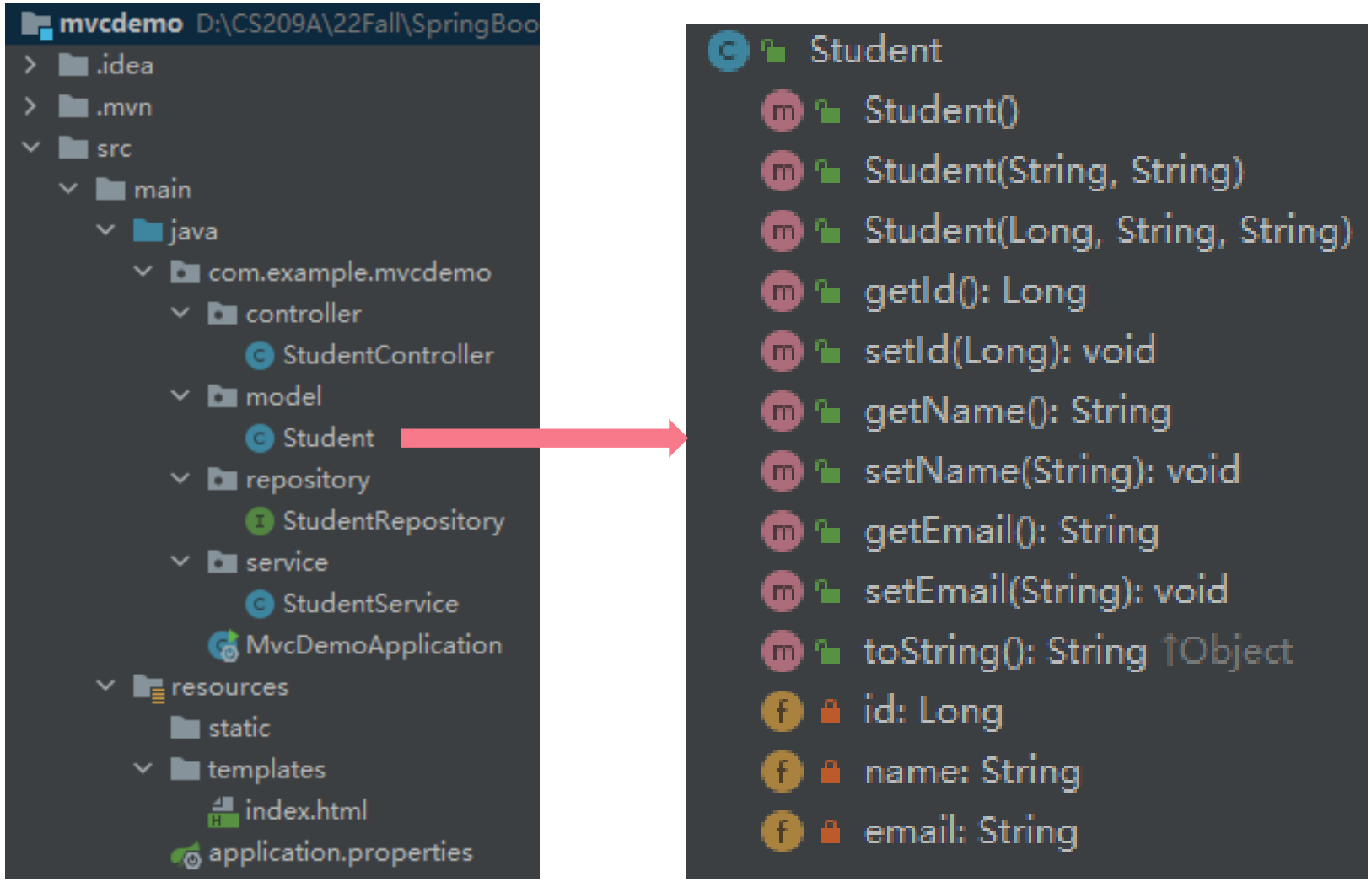
Spring Boot Flow Architecture





Model





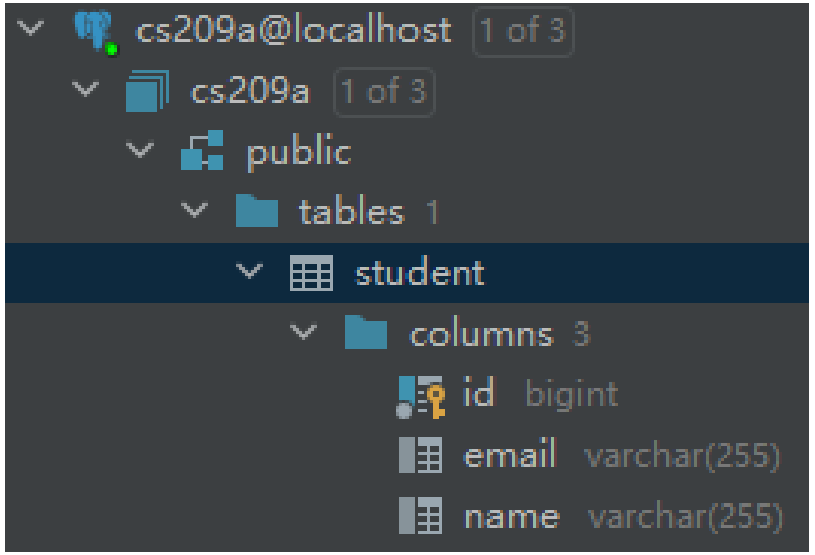
JavaBean: a POJO that conforms to certain conventions

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

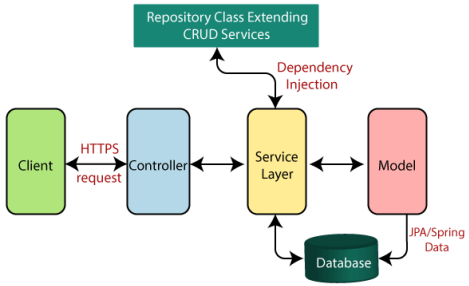
Mapping Model Class to Database Table

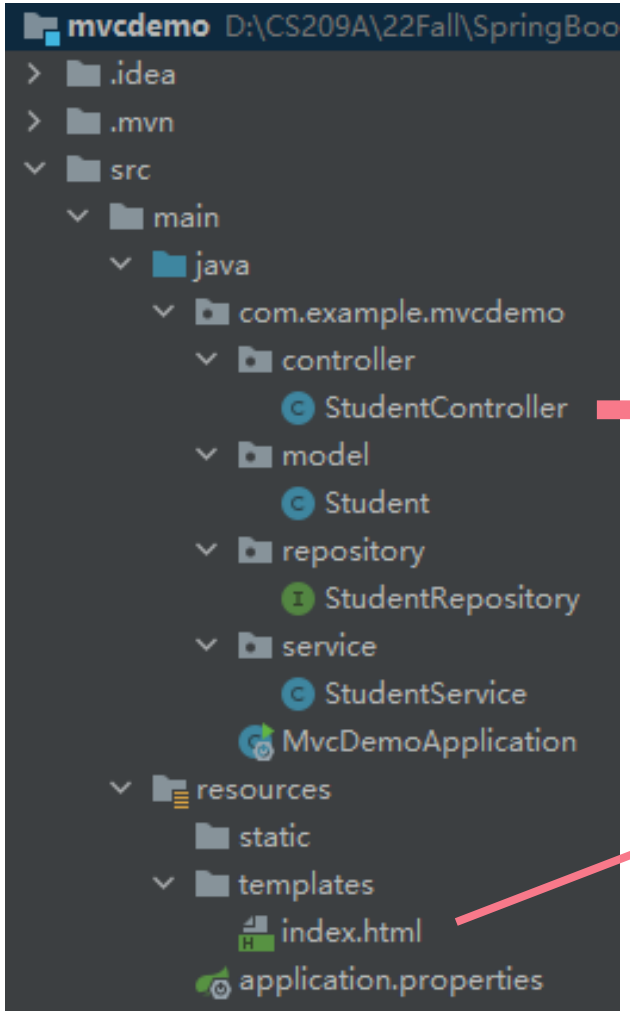
```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

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

    1 usage
    public Student() {
    }
```

`@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).



Controller

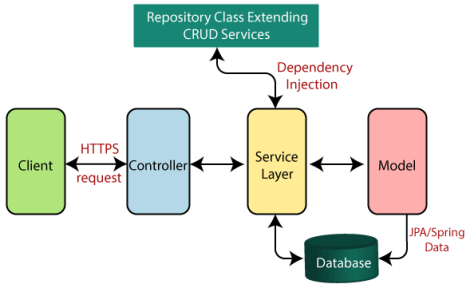


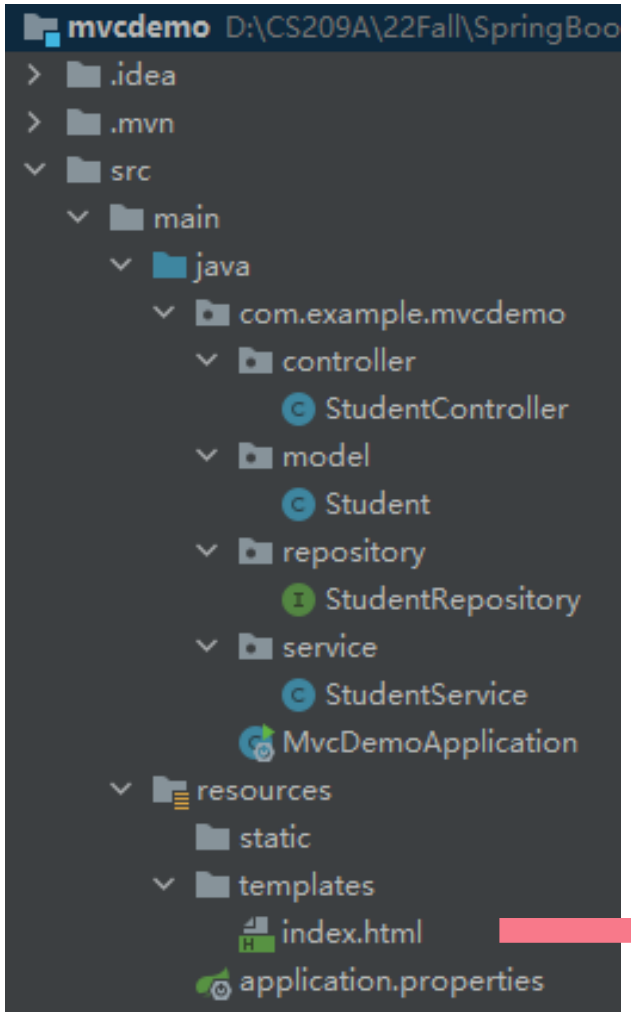



```java
@Controller
public class StudentController {

    2 usages
    private final StudentService studentService;

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

    @RequestMapping("/students")
    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.

View

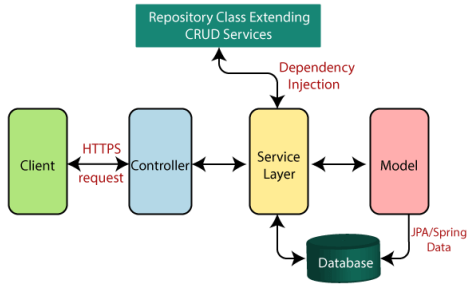


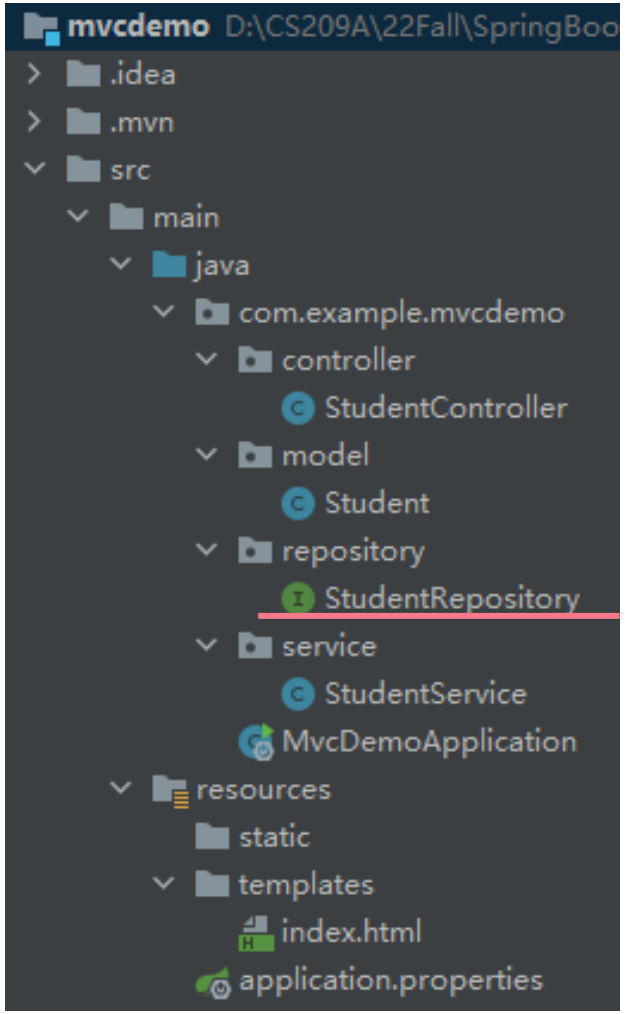



```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Spring Boot Demo</title>
</head>
<body>
<h1>Student List</h1>
<table>
    <tr th:each="student: ${students}">
        <td th:text="${student.id}">ID</td>
        <td th:text="${student.name}">Name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

Display model attributes in HTML.

Repository





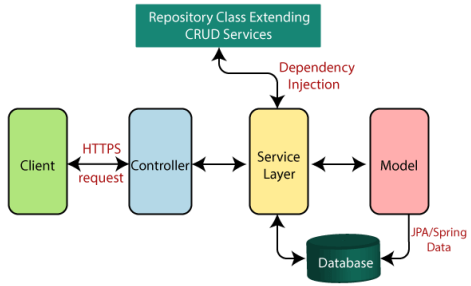

```java
import com.example.mvcdemo.model.Student;
import org.springframework.data.jpa.repository.JpaRepository;

5 usages
public interface StudentRepository extends JpaRepository<Student, Long> {
}
```

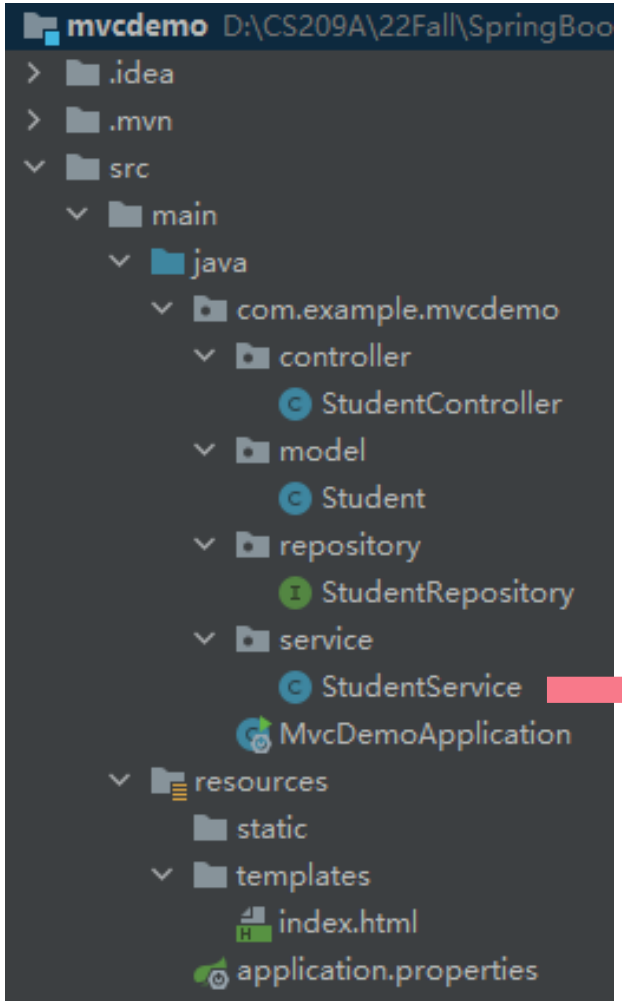
Entity/Model class name

Type of the id

- Create a repository interface (by extending specific repository interfaces such as **`JpaRepository`**) for each domain/model entity in the application.
- A repository contains methods for performing CRUD operations, sorting and paginating data.
- We could either use the provided methods (e.g., **`findAll()`**), or define customized finder (e.g., **`findByAgeGreaterThan`**). Spring will provide the implementation automatically.

Service






```java
@Service
public class StudentService {

    3 usages
    private final StudentRepository studentRepository;

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

    1 usage
    public List<Student> getStudents(){
        return studentRepository.findAll();
    }

    1 usage
    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@gmail.com");
        studentRepository.saveAll(List.of(maria, alex, dean));
    }
}
```

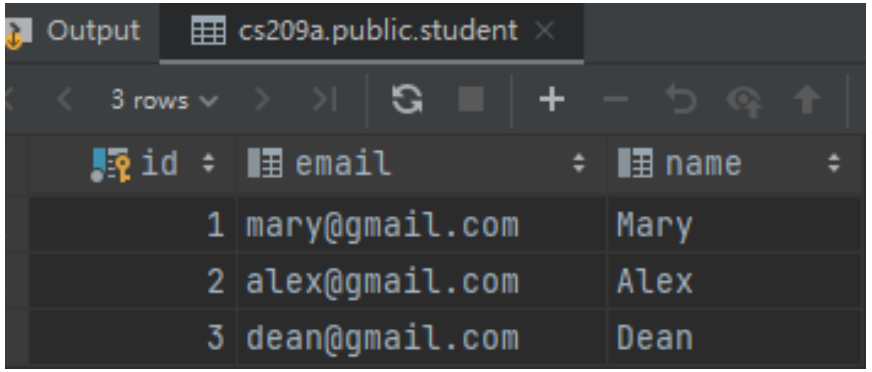
`@Service`: used with classes that provide business functionalities.

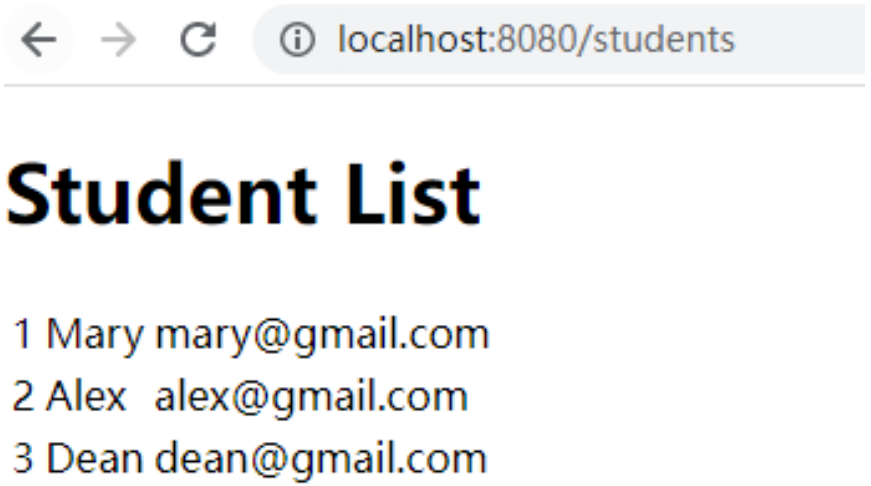
`@Autowired`: injecting beans at runtime

Bootstrap

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

```java
@SpringBootApplication
public class MvcDemoApplication {

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

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

}
```

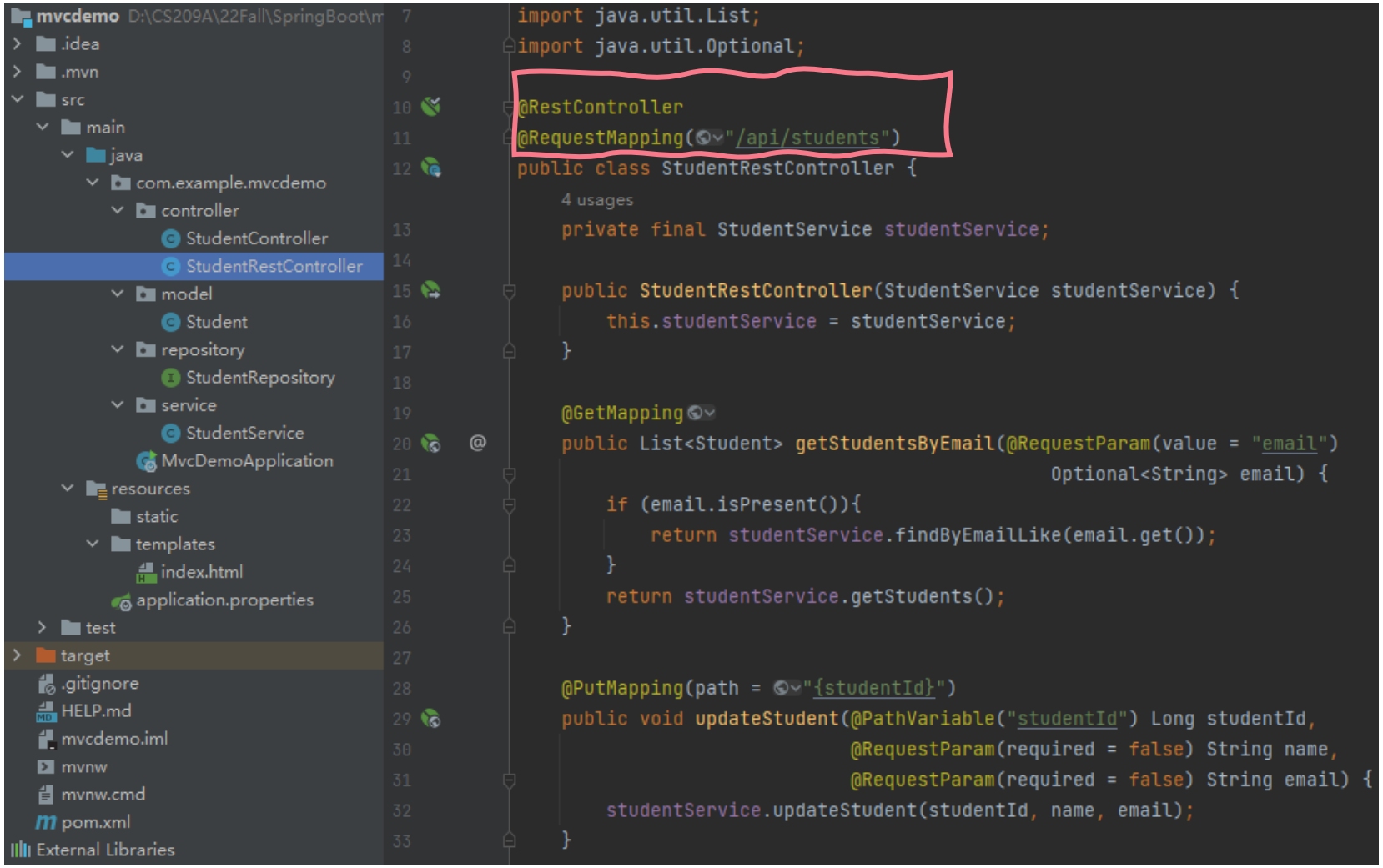




Building a RESTful Web Service

- 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

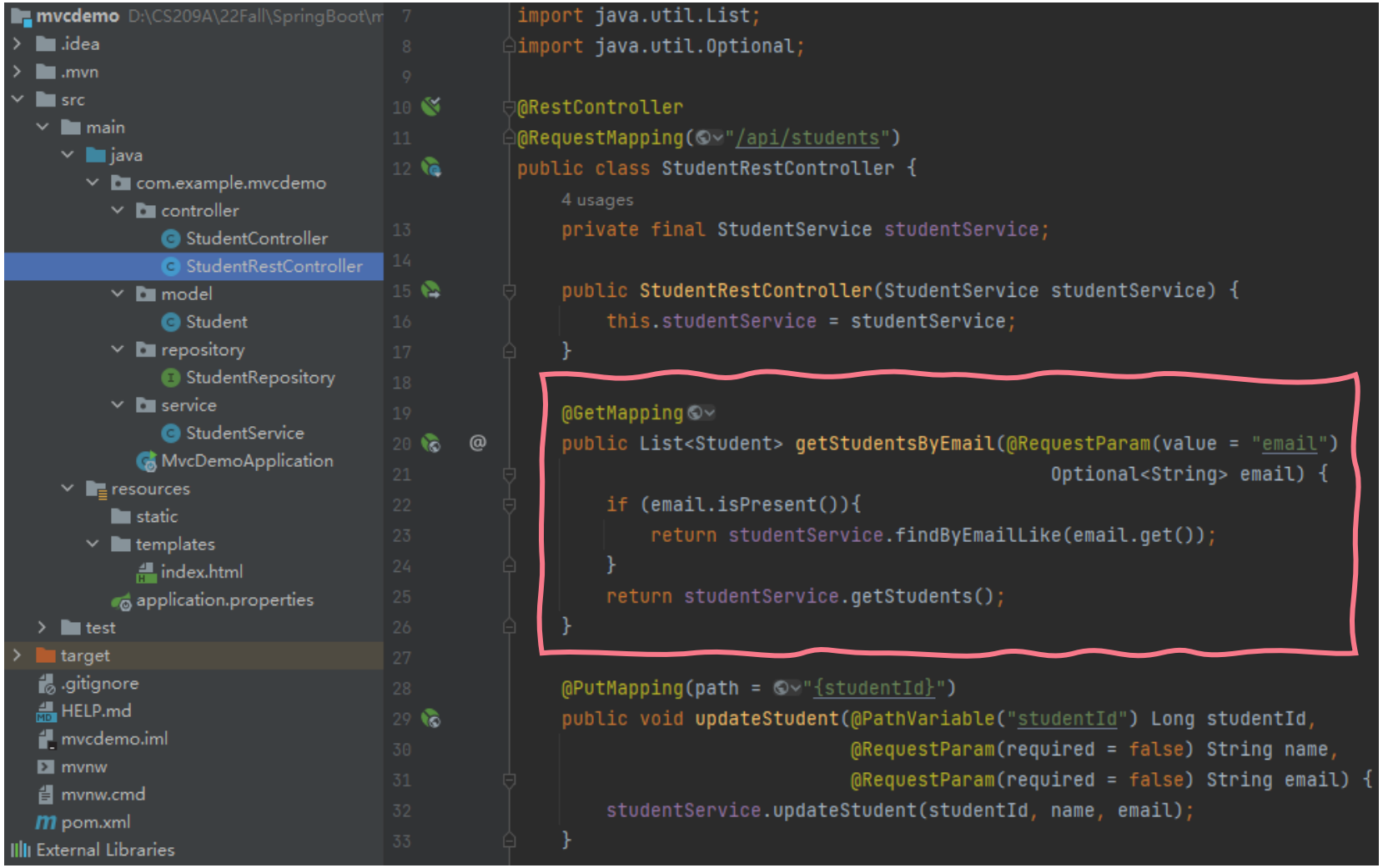
RestController



`@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

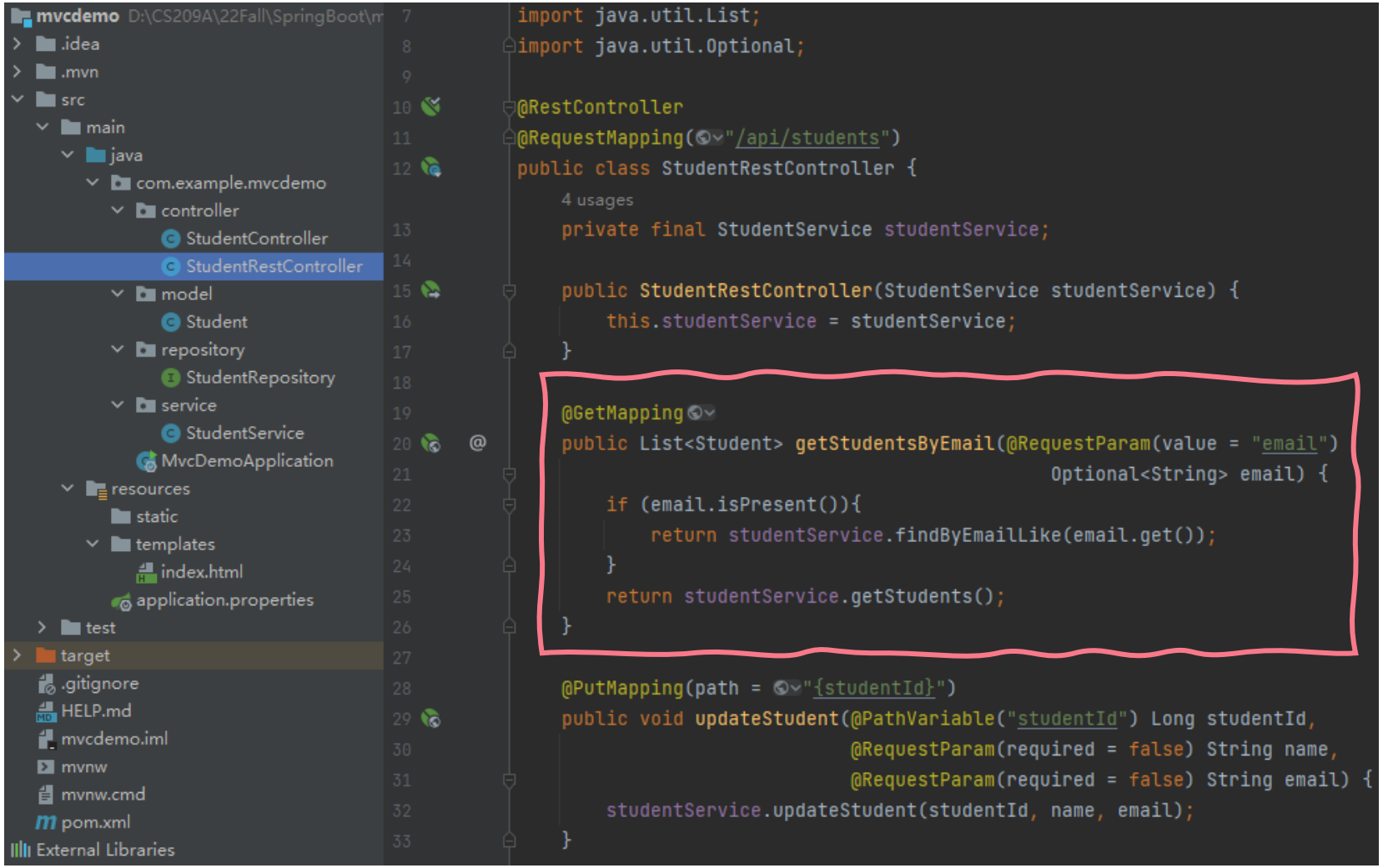
RestController

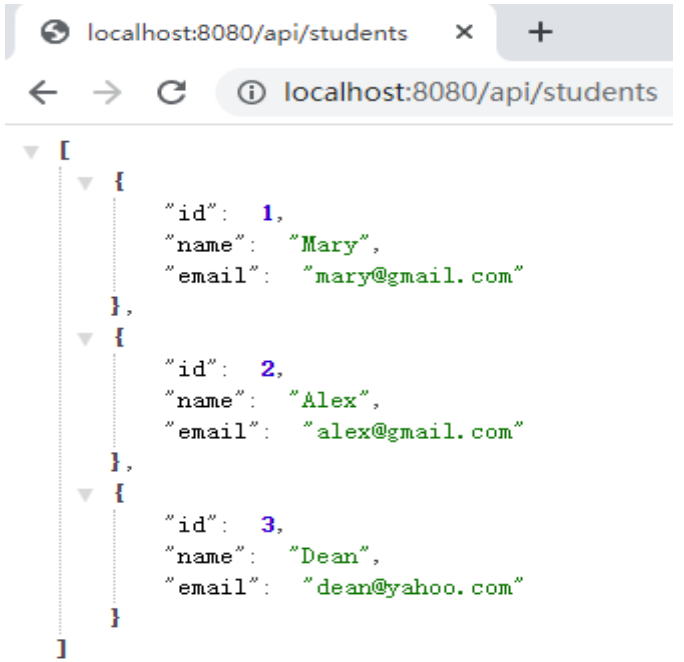


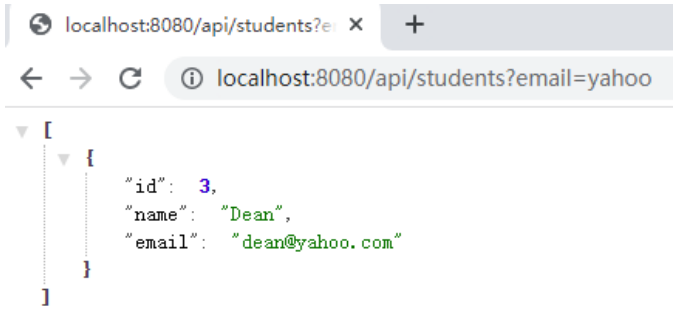
`@GetMapping`: ensures that HTTP GET requests to api/students are mapped to the corresponding method.

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

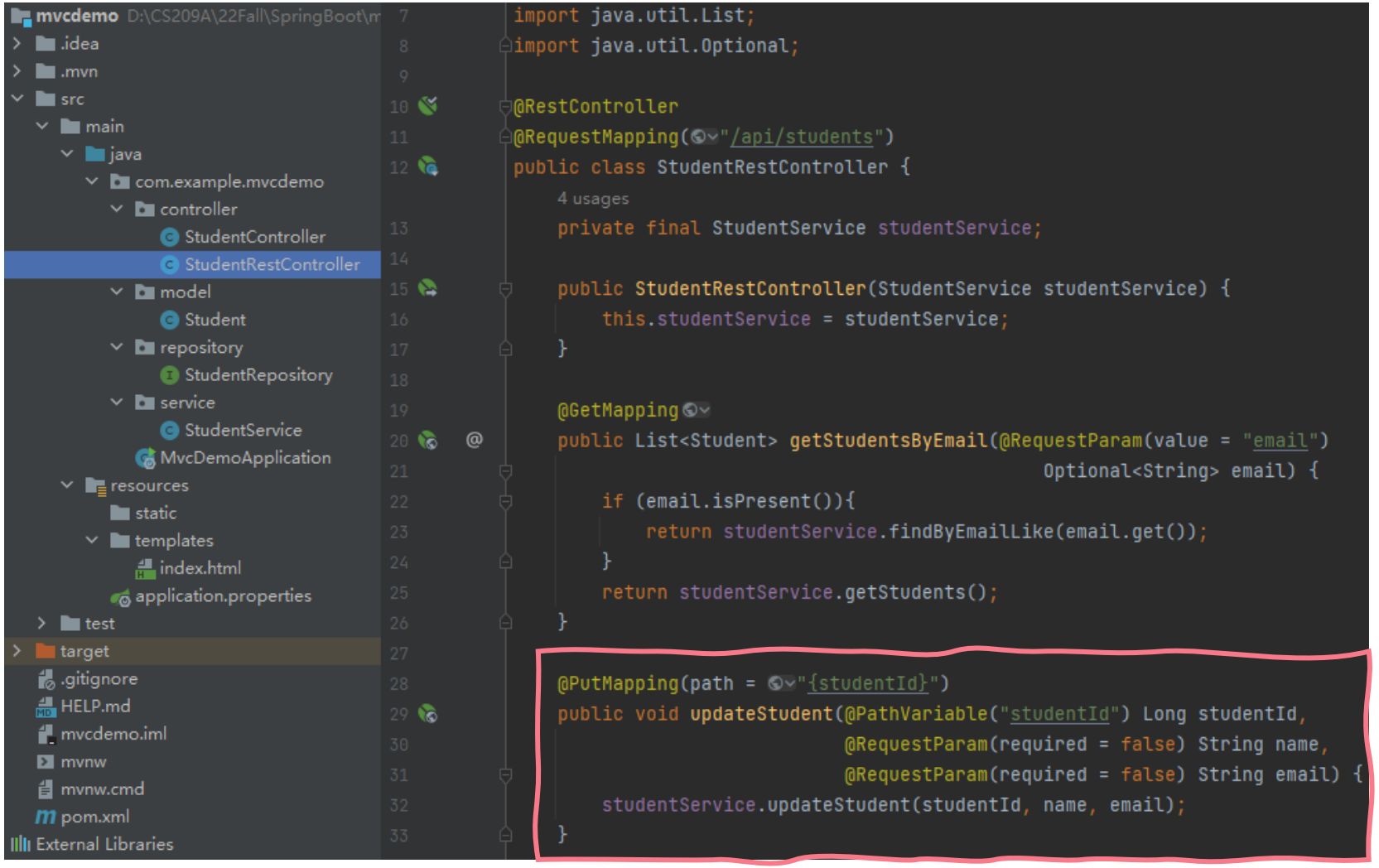
RestController







RestController

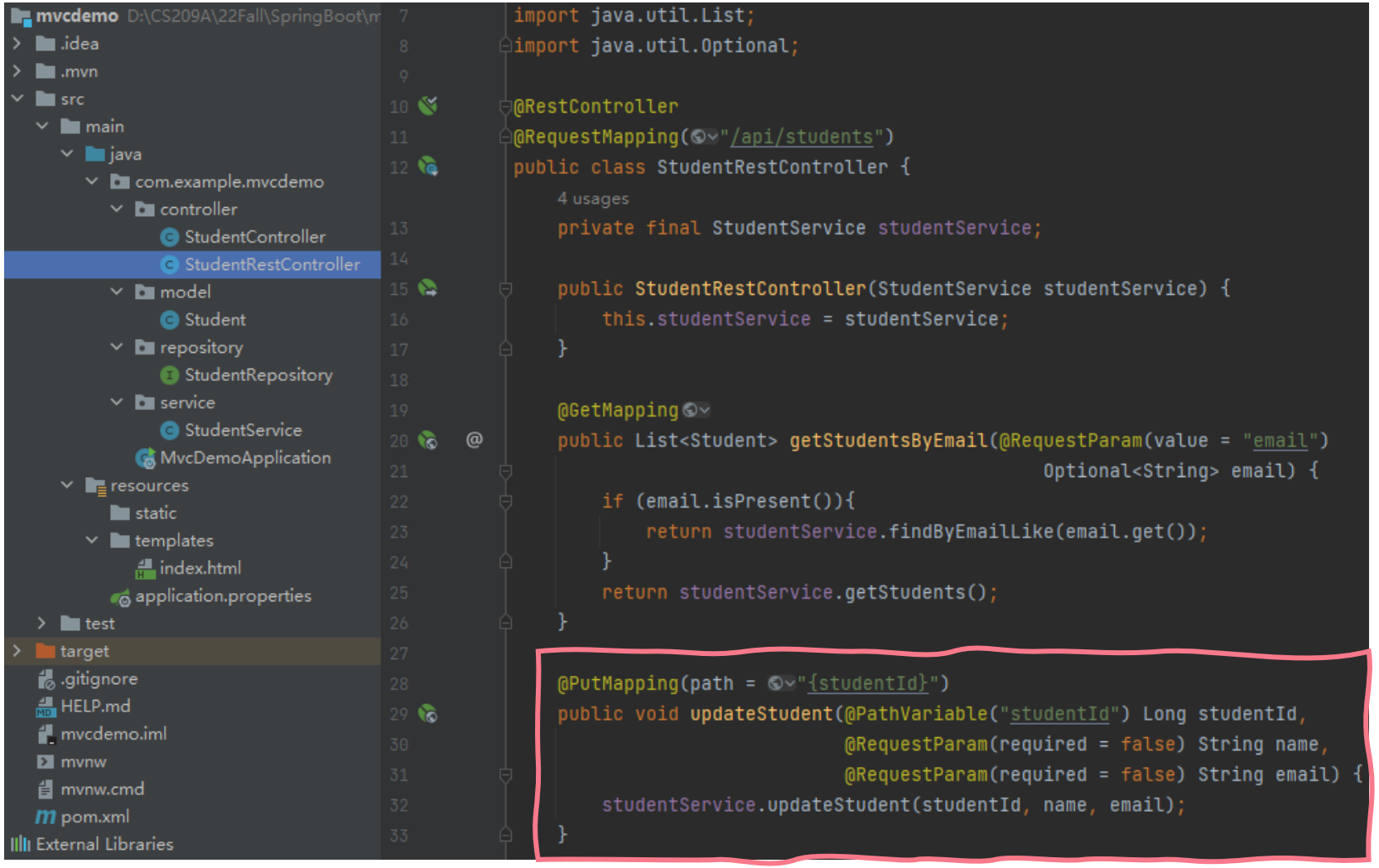


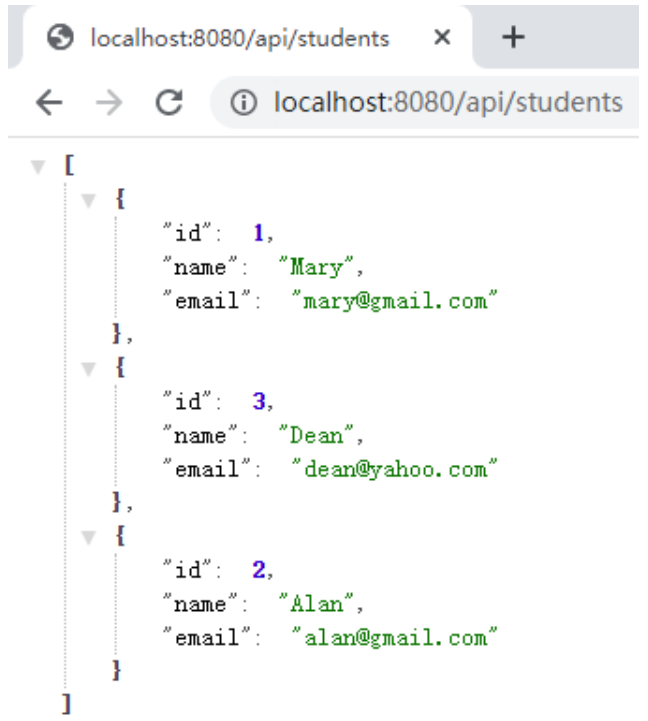
`@PutMapping`: maps HTTP PUT requests onto specific handler methods (shortcut for `@RequestMapping(method = RequestMethod.PUT)`)

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

RestController

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





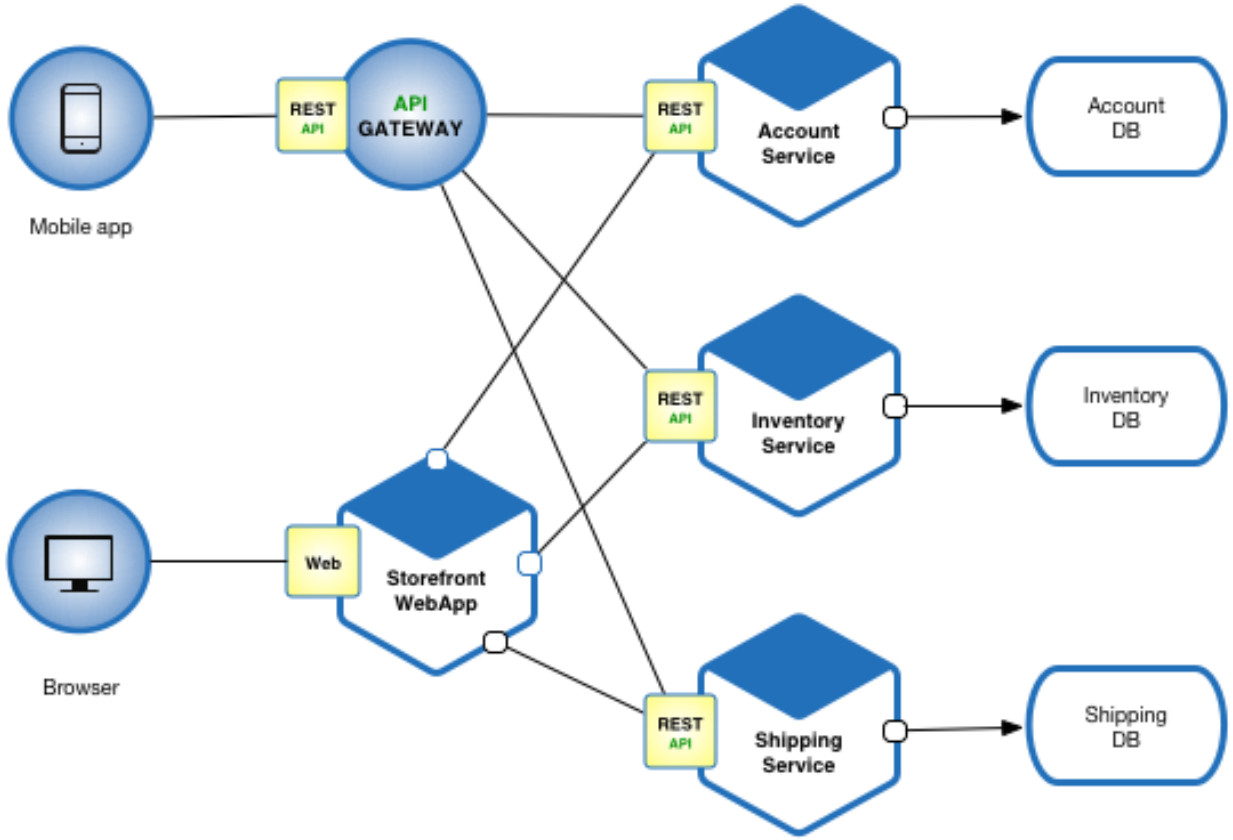
Java Microservices

- 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
- Microservice style (微服务架构) vs Monolithic style (整体式架构)

Monolithic Architecture

- The whole application is packaged into a single jar/war
- Less flexible for large team/code base
- Difficult to scale, wasting deployment resources

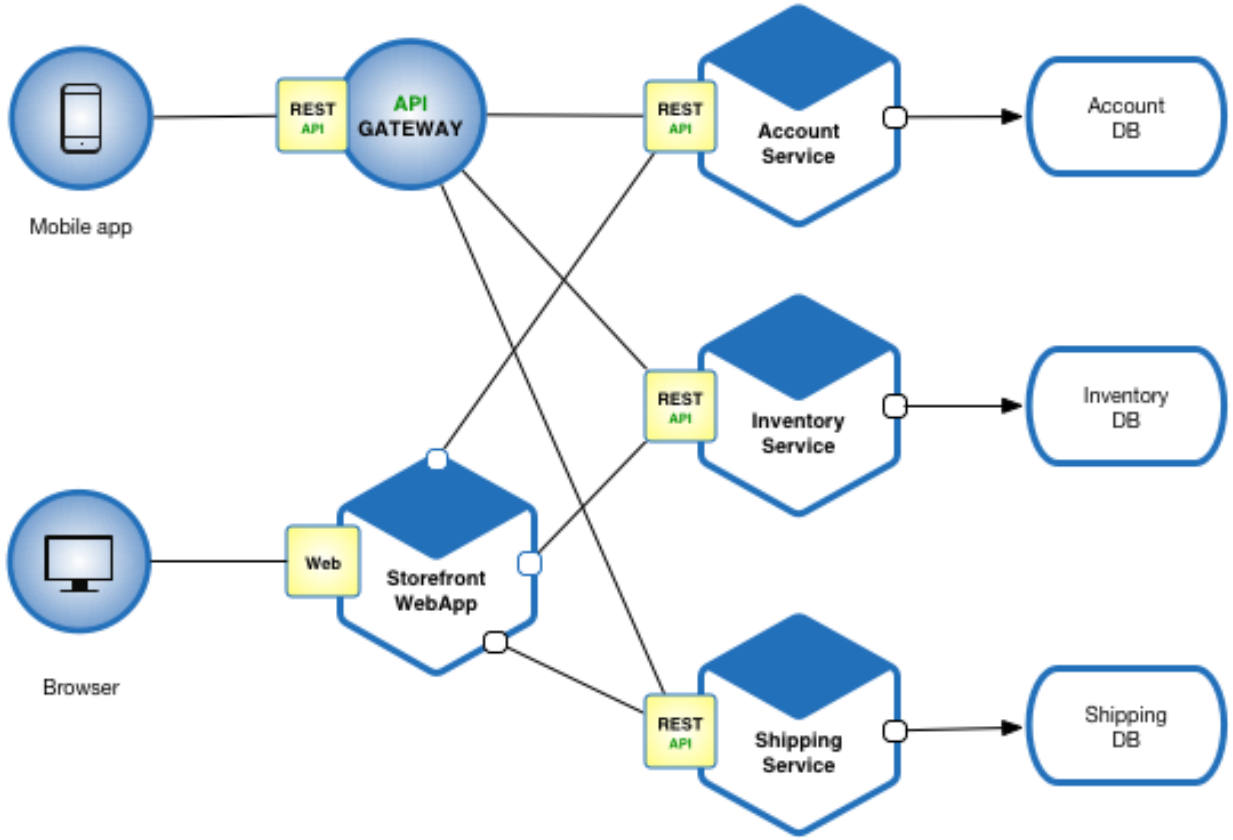
Microservice Architecture



https://microservices.io/patterns/microservices.html

- Microservices allow a large application to be separated into smaller independent parts, with each part having its own responsibility
- Each service can be developed, managed, and deployed independently.
- Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.

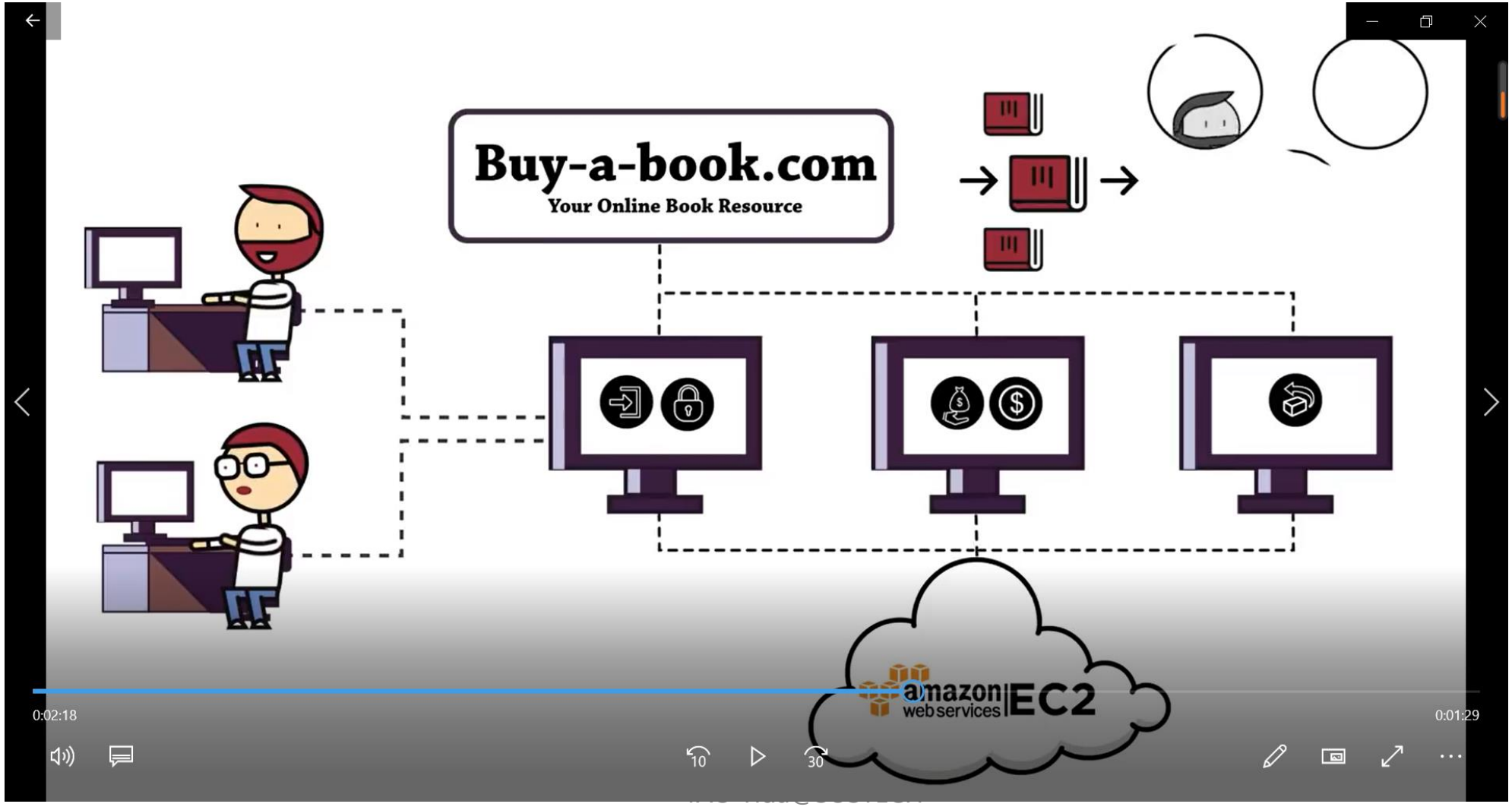
Microservice Architecture



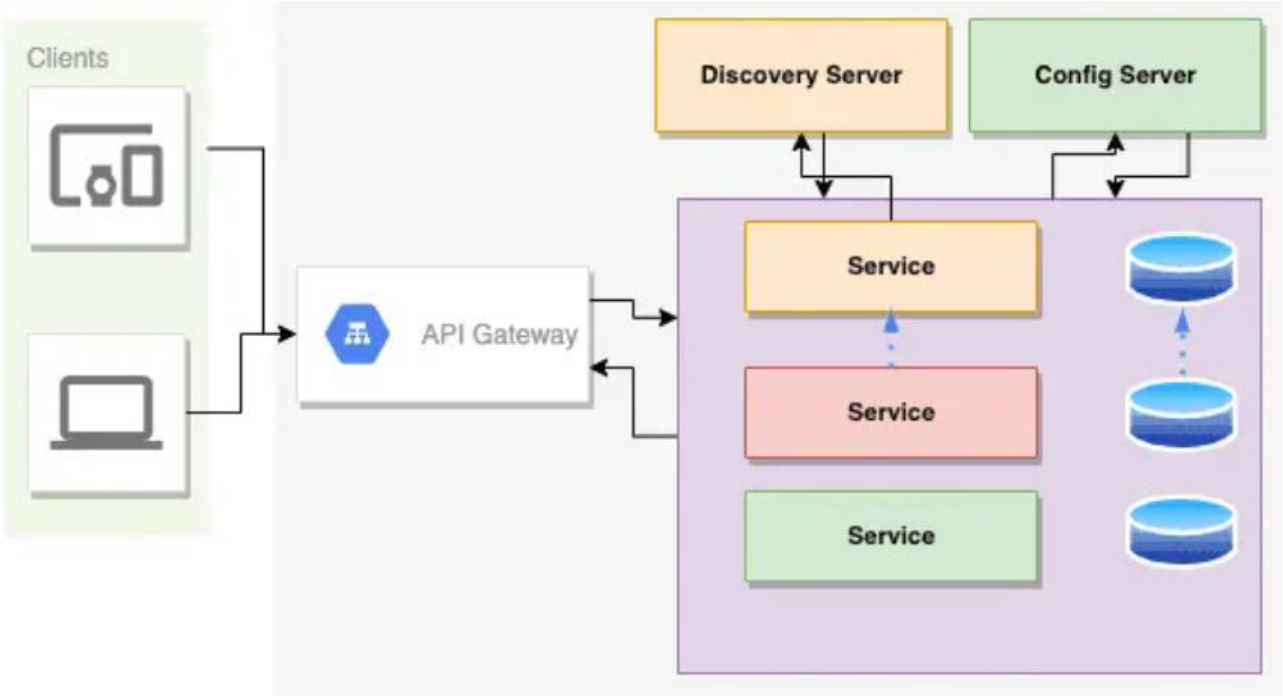
https://microservices.io/patterns/microservices.html

- Each microservice could be accessed via REST API or web interface
- We could scale only the required microservice (e.g., we could have a lot more instances of the account microservice than the shipping microservice)

Microservices



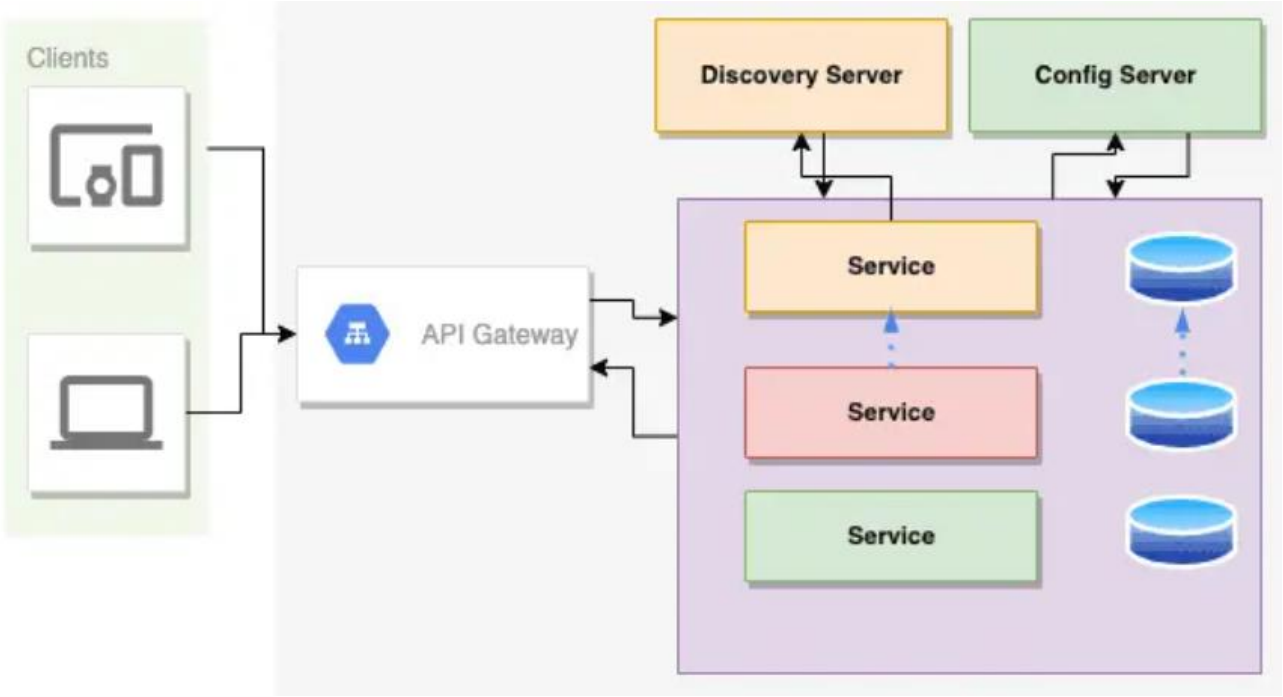
Microservice Architecture



- Each microservice has its own database.
- Clients typically do not have direct access to the services. It typically interacts through API gateway.
- API Gateway: When a client makes a request, the API gateway breaks it into multiple requests, routes them to the right places, produces a response, and keeps track of everything.

https://www.javadevjournal.com/spring-boot/microservices-with-spring-boot/

Microservice Architecture



- We will register each service with the discovery server, which has information of all the microservices available in the system.
- Configuration server contains all the configurations for our microservices and we will use this server to get configuration information like hostname, url etc. for our microservices.

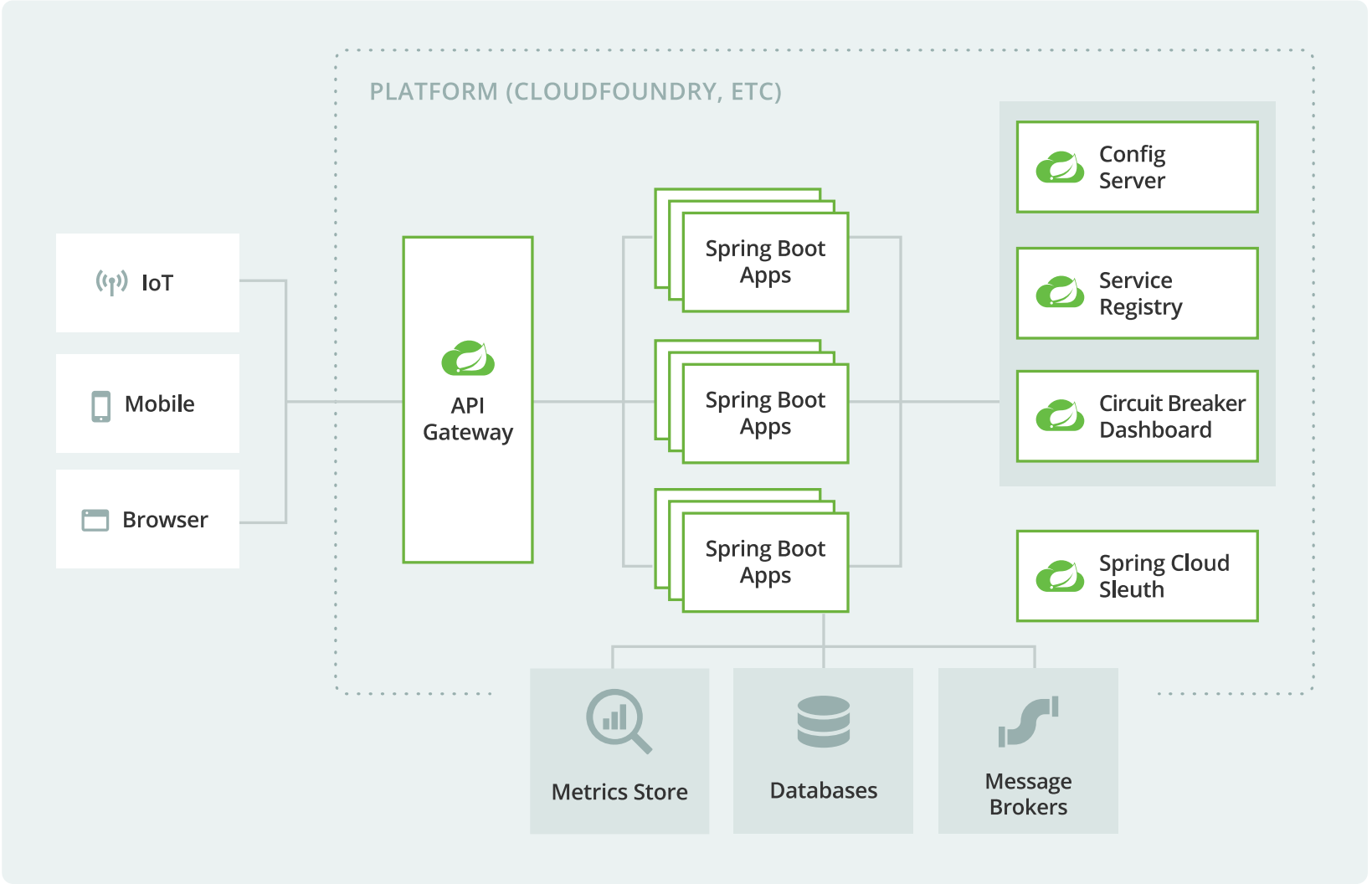
https://www.javadevjournal.com/spring-boot/microservices-with-spring-boot/

Microservices Resilience with Spring Cloud

- One of the key challenges in deploying a microservice is handling smooth communication between different microservices.
- One might require load balancers, some sort of central registry that keeps track of which microservices are up or down, and error handling in case of broken connections, etc.
- Spring Cloud provides such tool support
  - Functional services: statistics service, account service and notification service
  - Supporting infrastructure services: log analysis, configuration server, service discovery, authentication service, etc.

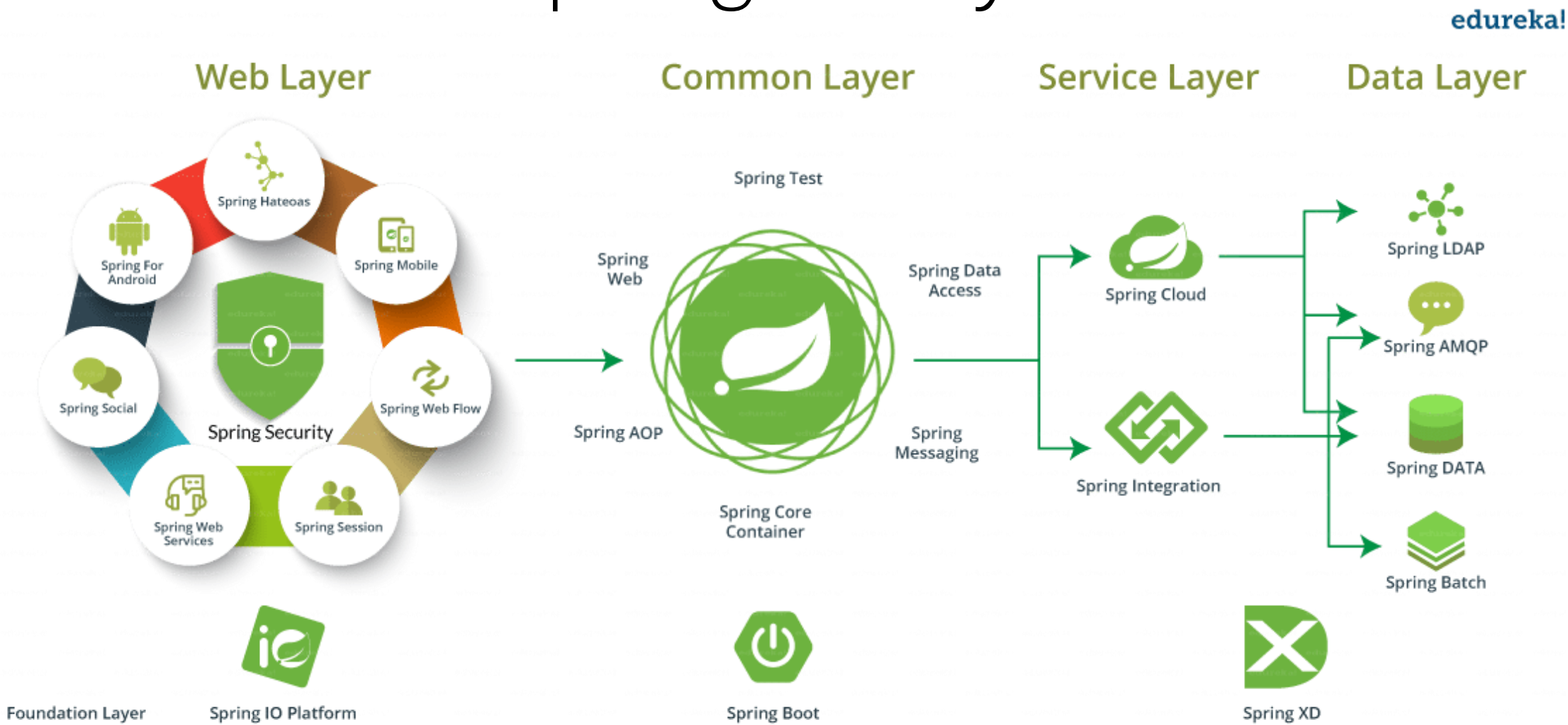
https://medium.com/clover-platform-blog/building-a-microservice-with-spring-boot-and-spring-cloud-1c8275d7d229

Microservices Resilience with Spring Cloud



https://spring.io/microservices

The Spring Ecosystem



Next Lecture

- Design Patterns