Spring Boot Master Course

Rajeev Gupta

rgupta.mtech@gmail.com

https://www.linkedin.com/in/rajeevguptajavatrainer

rgupta.mtech@gmail.com

Trainer's Profile

- Expert Java trainer MTech (Computer Science) with 18+ years experience in expertise in Java , OOAD, Design Patterns core, Spring Framework, Spring Boot, Spring Boot Microservice, Spring REST, Spring Data, Spring Security, Spring WS, Spring Mongo DB, DevOps with Java, Jenkin, Docker, Kubernetes, Messaging with Rabbit MQ, Kafka, Cloud AWS, GCP, Hibernate 5, EJB 3, Struts 1/2
- Helping technology organizations by training their fresh and senior engineers in key technologies and processes.
- Taught graduate and post-graduate academic courses to students with professional degrees.
- Conducted about 100 batches including fresher and lateral engineers.

Corporate Client

- Bank Of America
- MakeMyTrip
- GreatLearning
- Deloitte
- Kronos
- Yamaha Moters
- IBM
- Sapient
- Accenture
- Airtel
- Gemalto
- Cyient Ltd
- Fidelity Investment Ltd
- Blackrock
- Mahindra Comviva
- Iris Software
- harman
- Infosys
- Espire
- Steria
- Incedo
- Capgemini
- HCL
- CenturyLink
- Nucleus
- Ericsson
- Ivy Global
- Avaya
- NEC Technologies
- A.T. Kearney
- UST Global
- TCS
- North Shore Technologies
- Incedo
- Genpact
- Torry Harris
- Indian Air force
- Indian railways

rgupta.mtech@gmail.com

Training Etiquette

Punctuality

Join the session 5 minutes prior to the session start time. We start on time and conclude on time!

Feedback

Make sure to submit a constructive feedback for all sessions as it is very helpful for the presenter.

Silent Mode

Keep your mobile devices in silent mode, feel free to move out of session in case you need to attend an urgent call.

Avoid Disturbance

Avoid unwanted chit chat during the session.

Spring Boot Master Course

Module 1: Introduction
Module 2: Spring Core
Module 3: Spring AOP
Module 4: Spring JDBC
Module 5: Spring MVC
Module 6: Spring Boot REST
Module 7: Spring Boot REST
Module 8: Spring Data Joins
Module 9: Spring Security
Module 10: RestTemplate Webclient ,HttpClient OpenFeign

rgupta.mtech@gmail.com

Spring REST Projects

- BlogPost Application
- Bank Application
- Event Management Application
- Employee Management Application
- ProductStore Application

rgupta.mtech@gmail.com

Module 1: Introduction to Spring Framework

rgupta.mtech@gmail.com

What is Spring framework?

rgupta.mtech@gmail.com

What Is Spring Framework?

Spring framework helps develop various types of applications using the Java platforms.
It provides an extensive level of infrastructure support.
Spring also provides the “Plain Old Java Objects” (POJOs) mechanisms using which developers can easily create the Java SE programming model with the full and partial JAVA EE(Enterprise Edition).
Spring strives to facilitate the complex and unmanageable enterprise Java application development revolution by offering a framework that incorporates technologies, such as:

Dependency Injection (DI)
Aspect-oriented Programming (AOP)
Reduction of boilerplate code using Template design pattern (As in Spring JDBC)
Plain Old Java Object (POJO)

rgupta.mtech@gmail.com

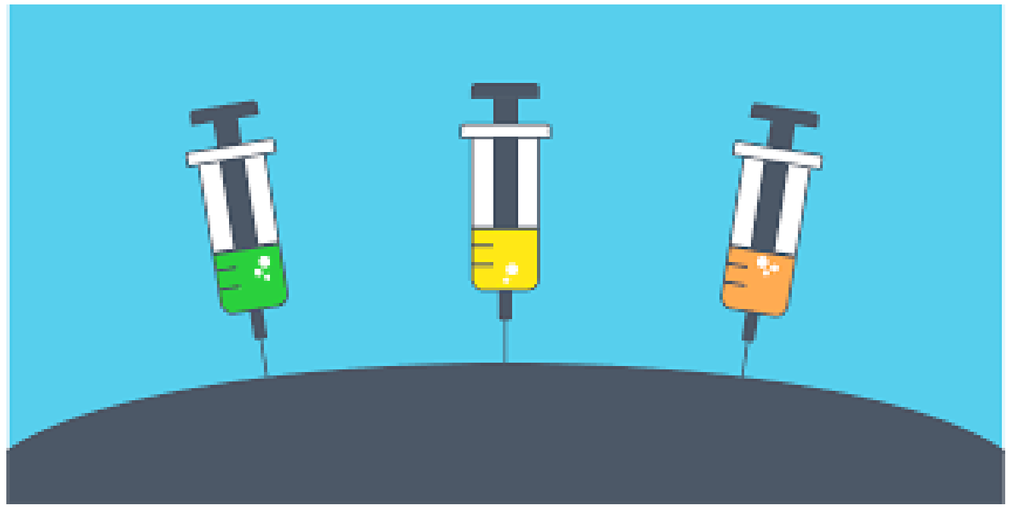
Why Spring framewrok

rgupta.mtech@gmail.com

The main goal of Spring framework is to simplify the development of enterprise software so it can be done quicker, because most of boilerplate configuration code are handled by the framework.

rgupta.mtech@gmail.com

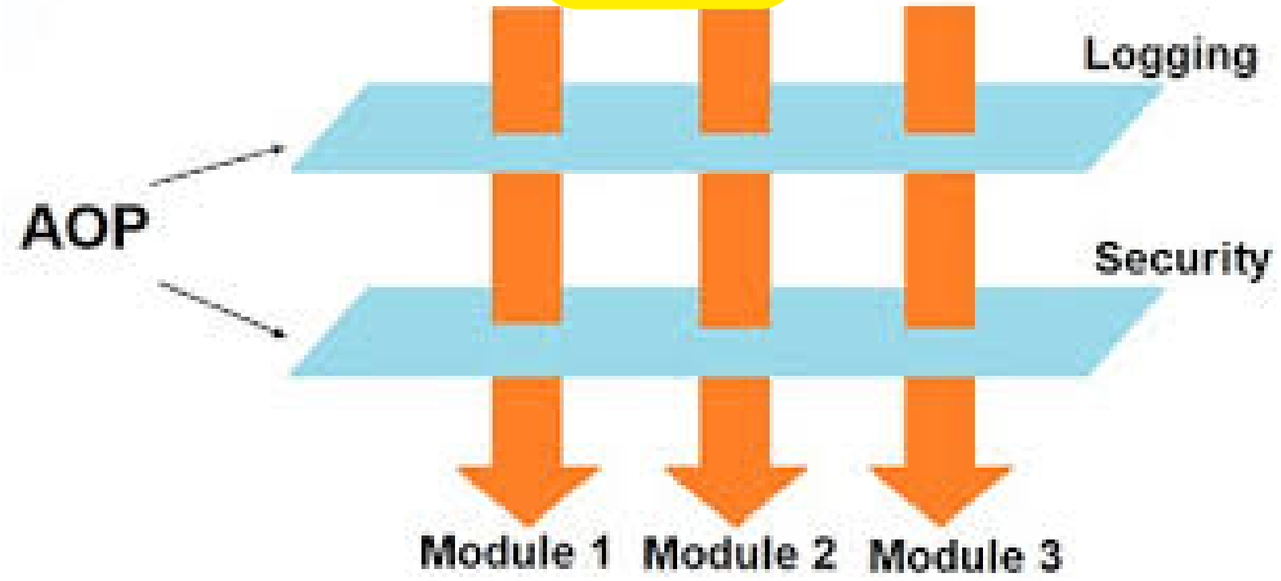
Spring
support Dependency Injection



Components are not responsible to create object on that it depends, spring container do that job!

rgupta.mtech@gmail.com

Spring
support Aspect Oriented Programming
AOP



Spring with AOP help us to reduce code clutter by improving the readability and maintainability of your code

rgupta.mtech@gmail.com

Spring
avoids writing lots of boilerplate Code

Spring help us to readuce
boilerplate code
spring as glue framework

rgupta.mtech@gmail.com


splunk>
EJB
terprise Java Bean
neo4j
kafka
aws
cassandra
JDBC
JavaServer Faces
JSF
PostgreSQL
mongoDB.
mon
STRUTS
WS
FEVICOL
MR
JPA
Java Persistence API
HIBERNATE
ZIPKIN
SMS
MySQL
JWT
Stack
docker
okta
Spring Integrates Anything
चुटकी में चिपकाये
rgupta.mtech@gmail.com

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

rgupta.mtech@gmail.com

Module 2: Spring Core

rgupta.mtech@gmail.com

Module 2: Spring Core Outline

- Introduction to Spring framework
- Spring Vs Spring Boot
- Dependency Injection using xml
  - Constructor, setter injection
  - C and p namespace
  - Scopes
  - Autowire
  - Collection mappings
  - Bean factory vs application context
  - Splitting configuration in multiple files
  - Bean life cycle
- Dependency Injection using annotation
  - @Autowired , @Qualifier, @Resource, @PostConstruct , @Predestroy, @Service, @Repository
- Dependency Injection using java configuration
  - AnnotationConfigApplicationContext
  - @Configuration, @Bean, @Import, @Scope
  - @PropertySources
  - Using Environment to retrieve properties
- Using Java configuration
  - What are Profiles?
  - Activating profiles

rgupta.mtech@gmail.com

What is Spring?

rgupta.mtech@gmail.com

Introduction to Spring Framework

The Spring Framework is an application framework and inversion of control container for the Java platform.

The framework's core features can be used by any Java application, but there are extensions for building web applications

release date: June 2003

Java based Application Framework that used to simplify Java EE application development

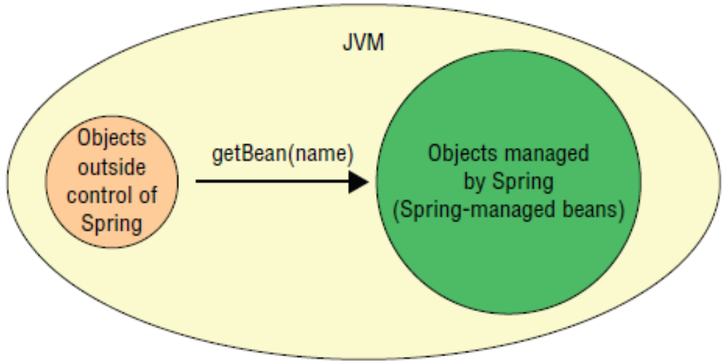
- Platform: Java EE
- Stable release: 6.0.7 / 20 March 2023
- Developer: VMware, Pivotal Software
- Programming language: Java on top of the Java EE platform.

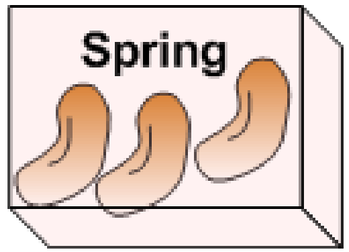
rgupta.mtech@gmail.com

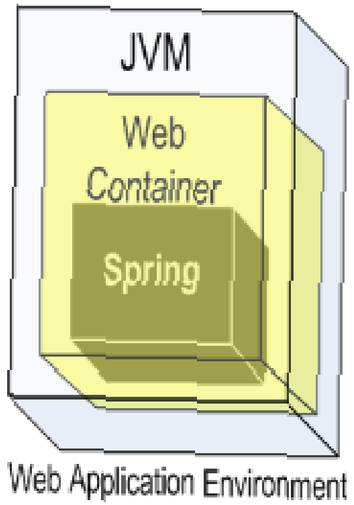
What is Spring framework?

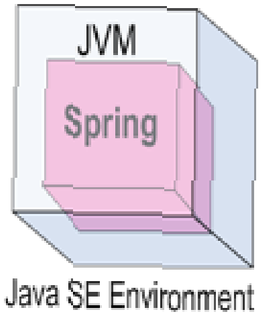
Spring framework is a container that manage life cycle of an bean it. It does three primary jobs:

- **Bean Wiring (Dependency Injection)**
- **Bean weaving (Aspect Oriented Programming)**
- **Reduction of boilerplate code using Template design pattern**









rgupta.mtech@gmail.com

Product management Three Tier Application

```java
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao=new ProductDaoImplMap();

    @Override
    public List<String> getAll() {
        return productDao.getAll();
    }

}
```

```java
import java.util.*;
public interface ProductDao {
    public List<String> getAll();
}
```

```java
public class ProductDaoImplMap implements ProductDao {

    @Override
    public List<String> getAll() {
        System.out.println("map implementation");
        return List.of("tv","laptop","keyboard");
    }

}
```

```java
public class ProductDaoImplJdbc implements ProductDao {

    @Override
    public List<String> getAll() {
        System.out.println("jdbc implementation");
        return List.of("tv","laptop","keyboard");
    }

}
```

rgupta.mtech@gmail.com

Spring Dependency Injection

Now we have new requirment from the management to use JDBC in DAO layer

We need to change code of service layer

Spring framework allow to swap the implementation without changing code in another layer

rgupta.mtech@gmail.com

Spring Dependency Injection

Now Spring inject Suitable implementation of dao layer and we need not to change code of service layer , wow!

```
0  public class ProductServiceImpl implements ProductService {
1
2      private ProductDao productDao;
3
4
5      public ProductServiceImpl() {}
6
7      public ProductServiceImpl(ProductDao productDao) {
8          this.productDao = productDao;
9      }
0
1      public void setProductDao(ProductDao productDao) {
2          this.productDao = productDao;
3      }
```

rgupta.mtech@gmail.com

Why Spring Boot?

```java
@Configuration
@ComponentScan(basePackages={"com.bookapp"})
@EnableAspectJAutoProxy
@PropertySource(value="db.properties")
@EnableTransactionManagement
public class ModelConfig {

    @Autowired
    private Environment environment;

    @Bean(name="dataSource")
    public DataSource getDataSource(){
        DriverManagerDataSource ds=new DriverManagerDataSource();
        ds.setDriverClassName(environment.getProperty("driver"));
        ds.setUrl(environment.getProperty("url"));
        ds.setUsername(environment.getProperty("username"));
        ds.setPassword(environment.getProperty("password"));
        return ds;
    }

    @Bean
    public LocalSessionFactoryBean getSessionFactory(){
        LocalSessionFactoryBean sf=new LocalSessionFactoryBean();
        sf.setDataSource(getDataSource());
        sf.setPackagesToScan("com.bookapp.model.persistance");
        sf.setHibernateProperties(getHibernateProperties());
        return  sf;
    }

    public Properties getHibernateProperties() {
        Properties properties=new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "validate");
        properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
        properties.setProperty("hibernate.show_sql", "true");
        properties.setProperty("hibernate.formate_sql", "true");
        return properties;
    }

    @Bean
    public PersistenceExceptionTranslationPostProcessor
    getPersistenceExceptionTranslationPostProcessor(){
        PersistenceExceptionTranslationPostProcessor ps=
                new PersistenceExceptionTranslationPostProcessor();
        return ps;
    }

    @Bean(name="transactionManager")
    //@Autowired
    public HibernateTransactionManager getHibernateTransactionManager(SessionFactory factory){
        HibernateTransactionManager tm=new HibernateTransactionManager();
        tm.setSessionFactory(factory);
        return tm;
    }
```

rgupta.mtech@gmail.com

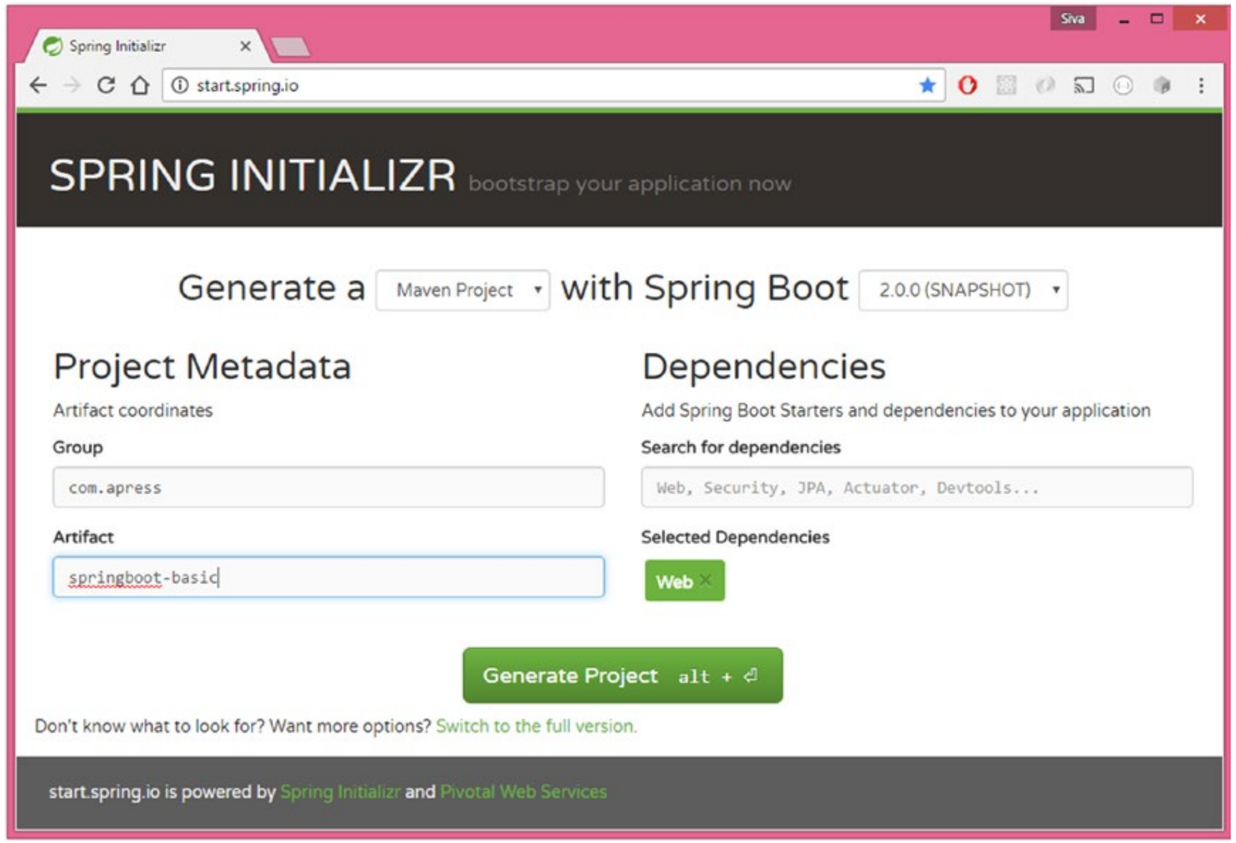
Spring vs Spring Boot

- In Spring we need to write lots of boilerplate code while Spring Boot reduces boilerplate code thus reducing LOC.
- In the case of Spring, we need to set up the server explicitly while Spring Boot offers an embedded server.
- The primary feature of Spring is dependency injection while the primary feature of Spring Boot is Autoconfiguration.
- We need to manually define dependencies for the Spring project while Spring Boot comes with the concept of starter that internally takes care of downloading the required dependencies i.e. JARs.

rgupta.mtech@gmail.com

Getting started with Spring

- We will understand How to create a simple Spring Boot web application serving a simple HTML page and explore various aspect of a typical Spring Boot application



rgupta.mtech@gmail.com

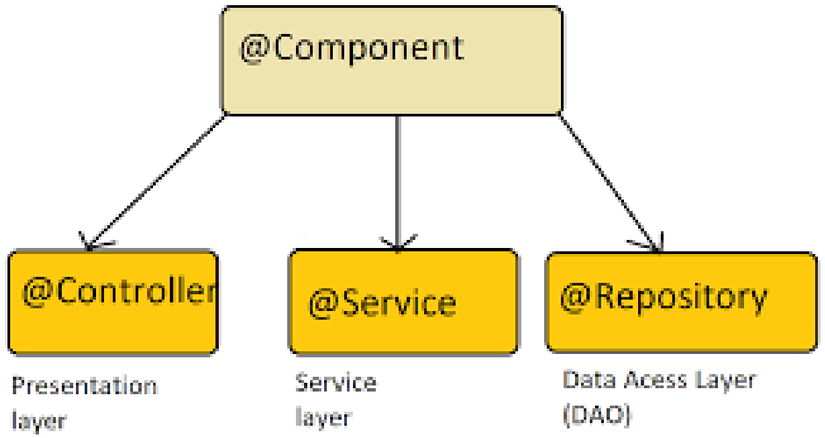
Productstore Application with Spring Framework

```java
@Service(value = "productService")
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;
    @Autowired
    public ProductServiceImpl(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }
```

```java
public interface ProductDao {
    public List<Product> getAll();
    public Product getById(Integer id );
    public Product addProduct(Product product);
    public Product deleteProduct(Integer id );
    public Product updateProduct(Integer id, Product product );
}
```

```java
@Repository
@Primary
public class ProductDaoMapImpl implements ProductDao{

    private HashMap<Integer, Product> productsMap;
    private int counter=0;
    public ProductDaoMapImpl() {
        productsMap=new HashMap<Integer, Product>();
        productsMap.put(1, new Product(1, "desktop", new BigDecimal(55000), true));
        productsMap.put(2, new Product(1, "Mouse pad", new BigDecimal(550), true));
        productsMap.put(3, new Product(1, "Torch", new BigDecimal(520), false));
        counter=3;
```

```java
@Repository
public class ProductDaoJdbcImpl implements ProductDao{

    private HashMap<Integer, Product> productsMap;
    private int counter=0;
    public ProductDaoJdbcImpl() {
        productsMap=new HashMap<Integer, Product>();
        productsMap.put(1, new Product(1, "laptop", new BigDecimal(55000), true));
        productsMap.put(2, new Product(1, "Mouse pad", new BigDecimal(550), true));
        productsMap.put(3, new Product(1, "Torch", new BigDecimal(520), false));
        counter=3;
```

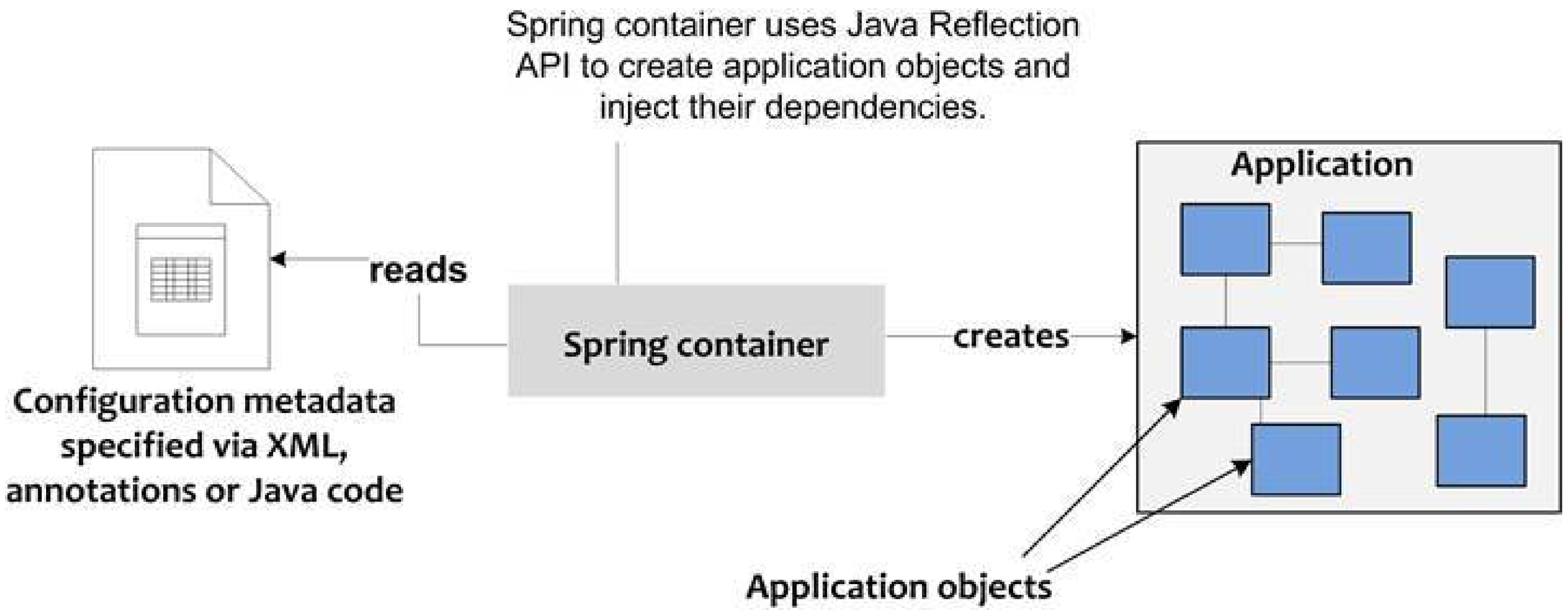


rgupta.mtech@gmail.com

Some of Spring modules

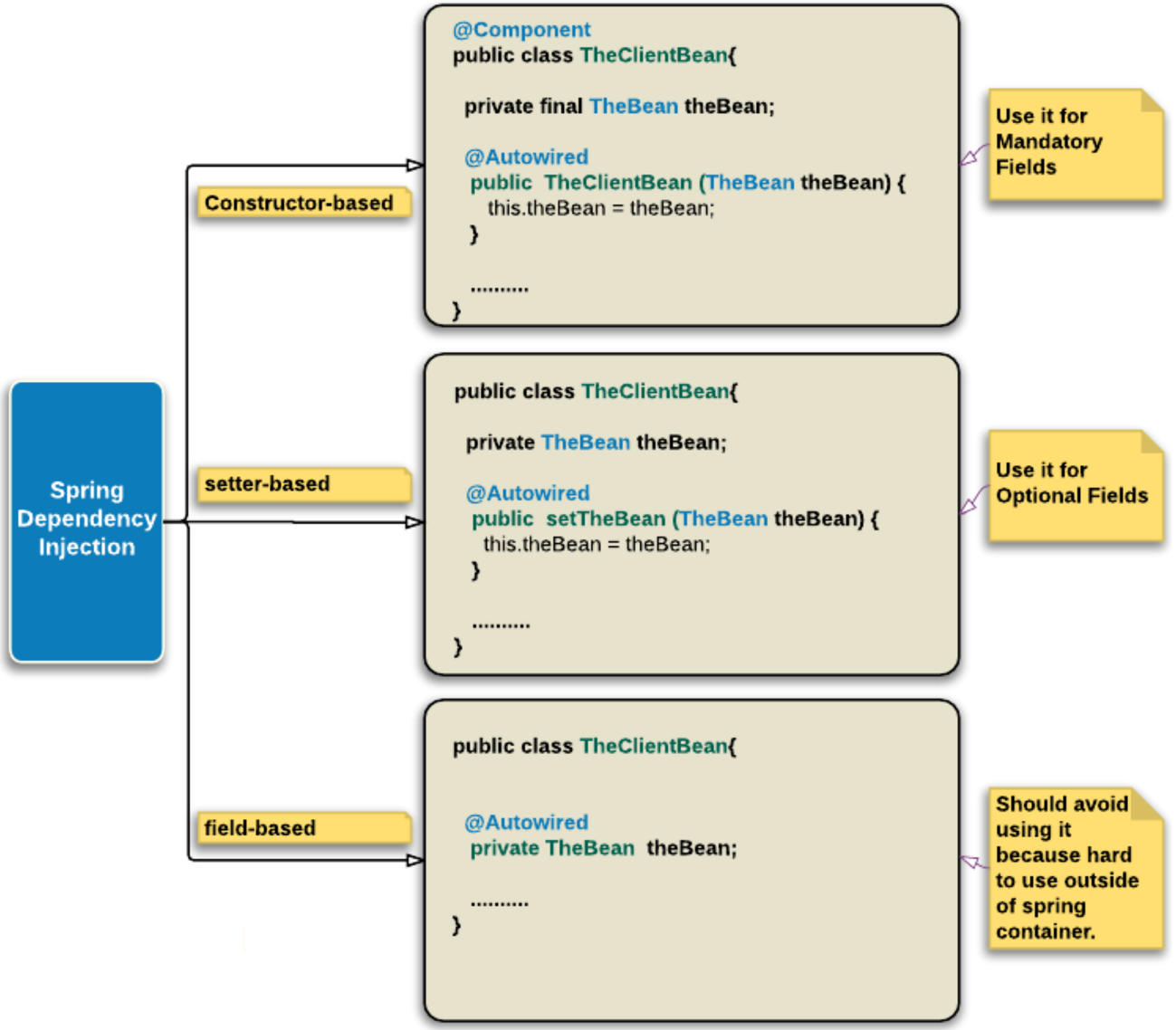
rgupta.mtech@gmail.com

How Spring Container works?



rgupta.mtech@gmail.com

Different ways of DI in Spring



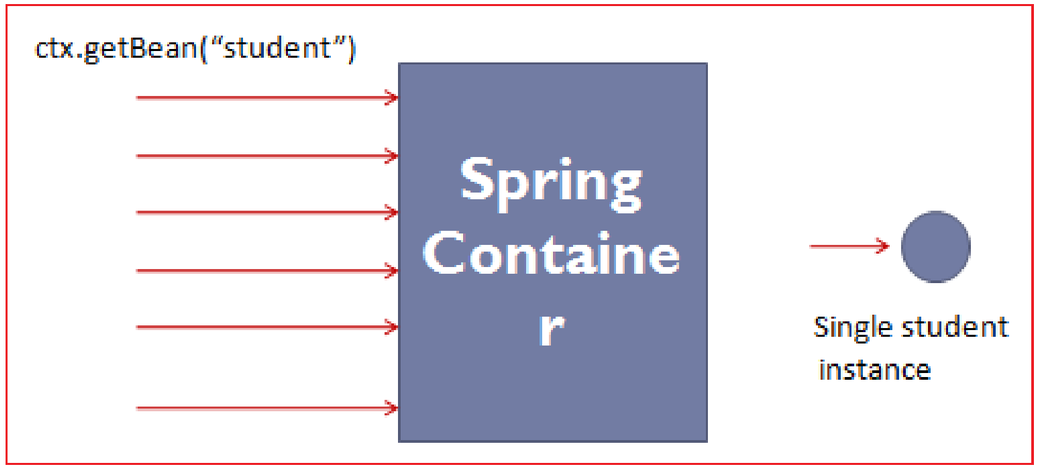
rgupta.mtech@gmail.com

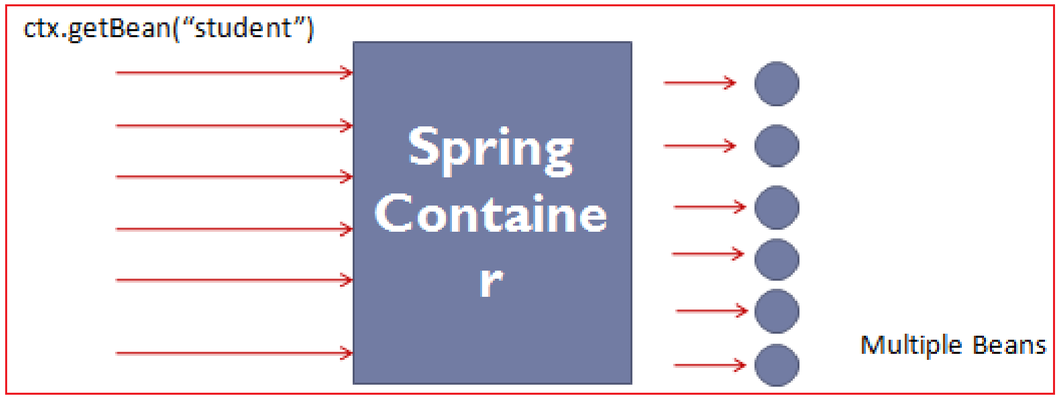
Bean Scopes

SCOPE NAME	SCOPE DEFINITION
`singleton`	Only one instance from a bean definition is created. It is the default scope for bean definitions.
`prototype`	Every access to the bean definition, either through other bean definitions or via the `getBean(..)` method, causes a new bean instance to be created. It is similar to the `new` operator in Java.
`request`	Same bean instance throughout the web request is used. Each web request causes a new bean instance to be created. It is only valid for web-aware `ApplicationContexts`.
`session`	Same bean instance will be used for a specific HTTP session. Different HTTP session creations cause new bean instances to be created. It is only valid for web-aware `ApplicationContexts`.
`globalSession`	It is similar to the standard HTTP Session scope (described earlier) and applies only in the context of portlet-based web applications.

rgupta.mtech@gmail.com

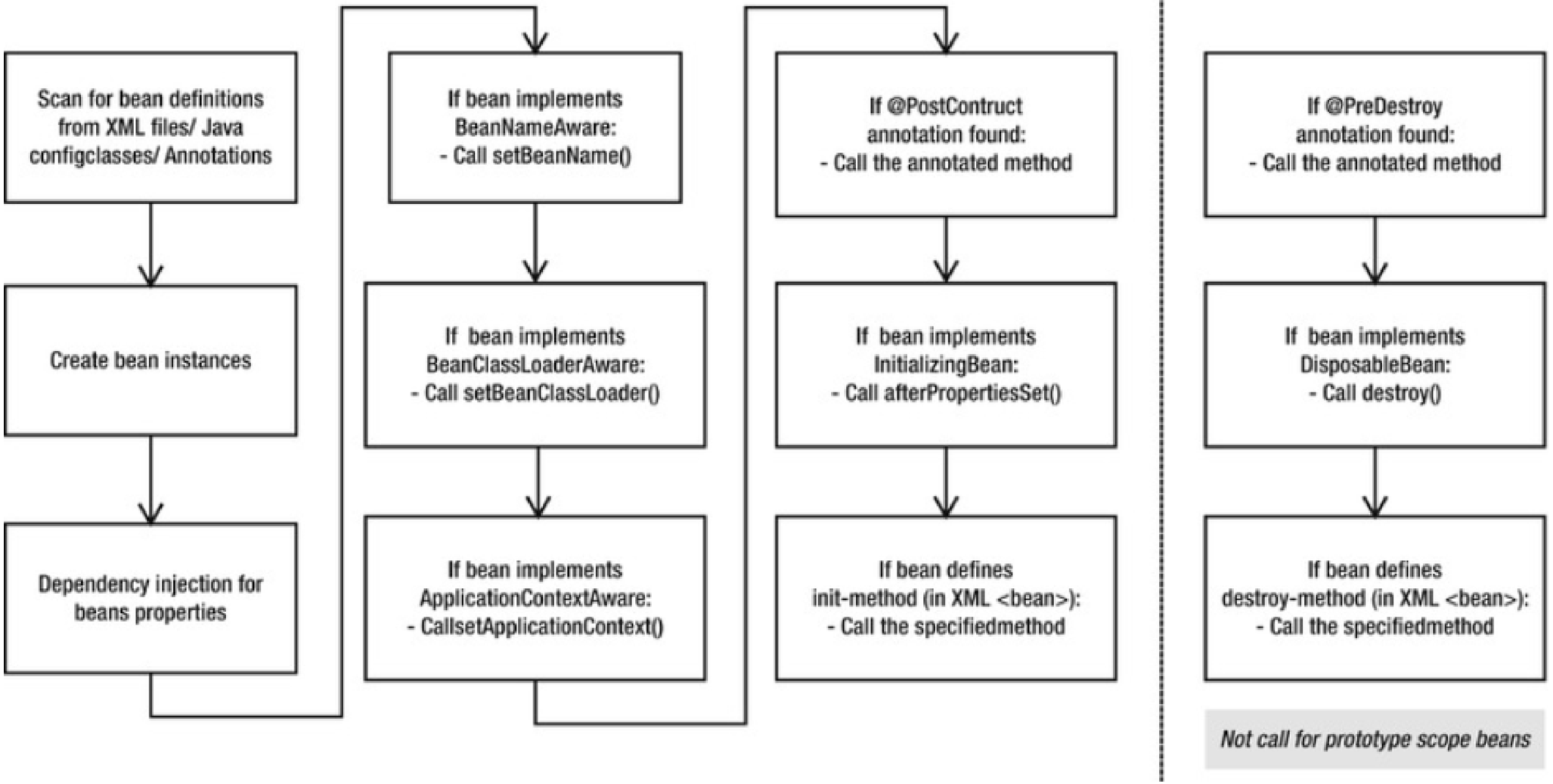
Bean Scopes





rgupta.mtech@gmail.com

Spring bean life cycle



rgupta.mtech@gmail.com

Spring bean life cycle: BeanPostProcessors

rgupta.mtech@gmail.com

Dependency Injection using annotations

- @Value - to inject a simple property
- @Autowire -to inject a property automatically
- @Component:@Controller @Service and @Repository
- @Qualifier - while autowiring, fix the name to an particular bean
- @Required - mandatory to inject, apply on setter
- @PostConstructs- Life cycle post
- @PreDestroy- Life cycle pre

- JSR 250 Annotations:
  - @Resource, @PostConstruct/ @PreDestroy, @Component
- JSR 330 Annotations:
  - @Named annotation in place of @Resouce
  - @Inject annotation in place of @Autowire

rgupta.mtech@gmail.com

DI using Java Configuration

```java
@Configuration
@ComponentScan(basePackages={"com.sample.bank.*"})
@Scope(value="prototype")
public class AppConfig {

    @Bean(autowire=Autowire.BY_TYPE)
    @Scope(value="prototype")
    public AccountService accountService(){
        AccountServiceImp accountService=new AccountServiceImp();
        //accountService.setAccountDao(accountDao());
        return accountService;
    }


    @Bean
    public AccountDao accountDao(){
        AccountDao accountDao=new AccountDaoImp();
        return accountDao;
    }
}
```

rgupta.mtech@gmail.com

@PropertySource & Using Environment to retrieve properties

```java
@Configuration
@ComponentScan(basePackages={"com.sample.bank.*"})
@PropertySource("classpath:db.properties")
public class AppConfig {
    @Autowired
    private Environment env;

    private Connection con;

    @Bean
    public Connection getConnection(){

        try{
            Class.forName("com.mysql.jdbc.Driver");
        }catch(ClassNotFoundException ex){
            ex.printStackTrace();
        }
        try{
            con=DriverManager.getConnection(env.getProperty("jdbc.url"),
                    env.getProperty("jdbc.username"),
                    env.getProperty("jdbc.password"));
        }catch(SQLException ex){
            ex.printStackTrace();
        }
        return con;
    }
}
```

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/foo
jdbc.username=root
jdbc.password=root
```

rgupta.mtech@gmail.com

Profile Using Java configuration

- What are Profiles?
  - @Profile allow developers to register beans by condition

```
public class Foo {
    private String name;

    public String getName() {
        return name;
    }

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

}
```

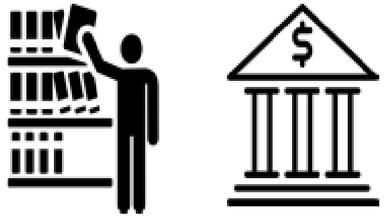
```
@org.springframework.context.annotation.Configuration
public class Configuration {

    @Bean
    @Profile("test")
    public Foo testFoo(){
        Foo foo=new Foo();
        foo.setName("test");
        return foo;
    }
    @Bean
    @Profile("dev")
    public Foo devFoo(){
        Foo foo=new Foo();
        foo.setName("dev");
        return foo;
    }
```

- We require to use caching in our book application we want to support two profile "dev" and "production"
- If profile “dev” is enabled, return a simple cache manager ConcurrentMapCacheManager
- If profile "production" is enabled, return an advanced cache manager – EhCacheCacheManager

rgupta.mtech@gmail.com

Spring Bean Configuration important tips

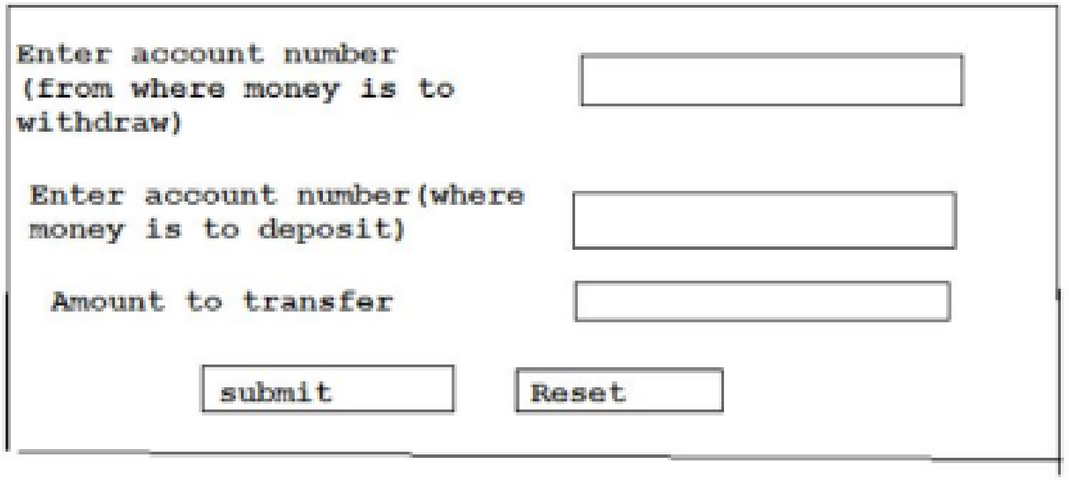
Infrastructure bean vs business bean

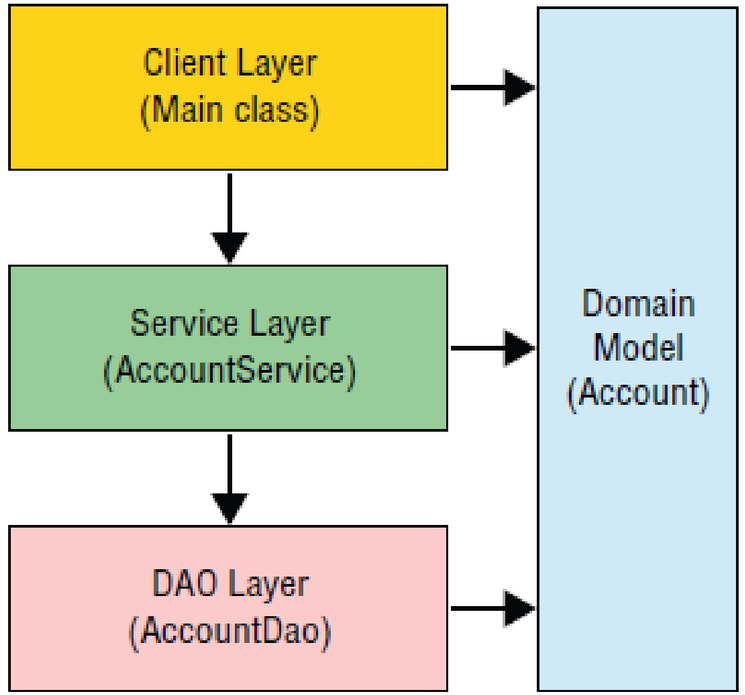


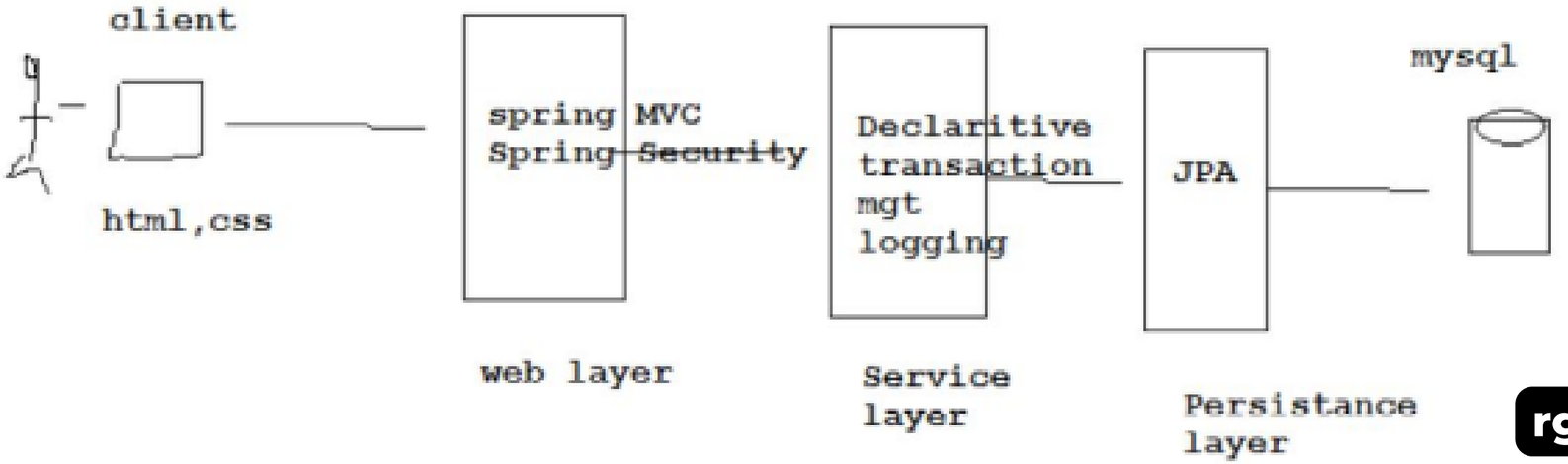
Infrastructure bean such as datasource connection, jms connection, hibernate connection is configured using xml or @Bean annotation while business bean such as ProductService, ProductDao, AccountService etc are configured using @Component family (@Service @Controller and @Repository)

rgupta.mtech@gmail.com

Bank Application







rgupta.mtech@gmail.com

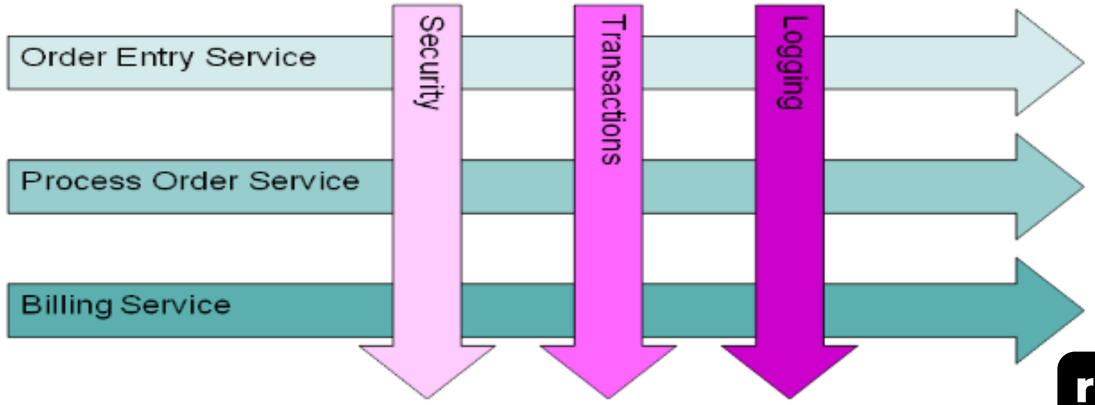
Module 3: Spring AOP

rgupta.mtech@gmail.com

Introduction to AOP

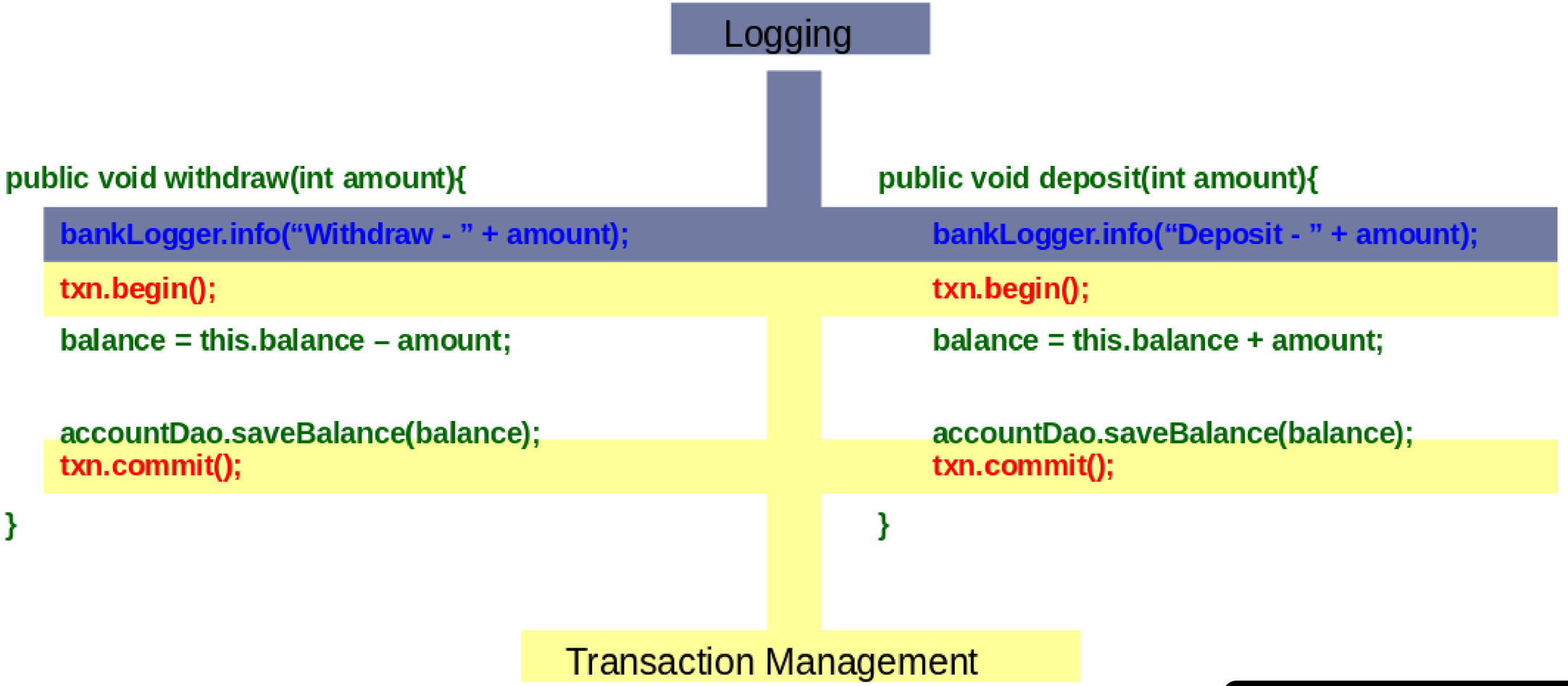
- What is AOP?
  - AOP is a style of programming, mainly good in separating cross cutting concerns
- How AOP works?

- Achieved usages Proxy design Pattern to separate CCC's form actual code
- Cross Cutting Concern ?
- Extra code mixed with the actual code is called CCC's Extra code mixed with code lead to maintenance issues
  - **Logging**
  - **validations**
  - **Auditing**
  - **Security**



rgupta.mtech@gmail.com

Handling Cross cutting Concerns



rgupta.mtech@gmail.com

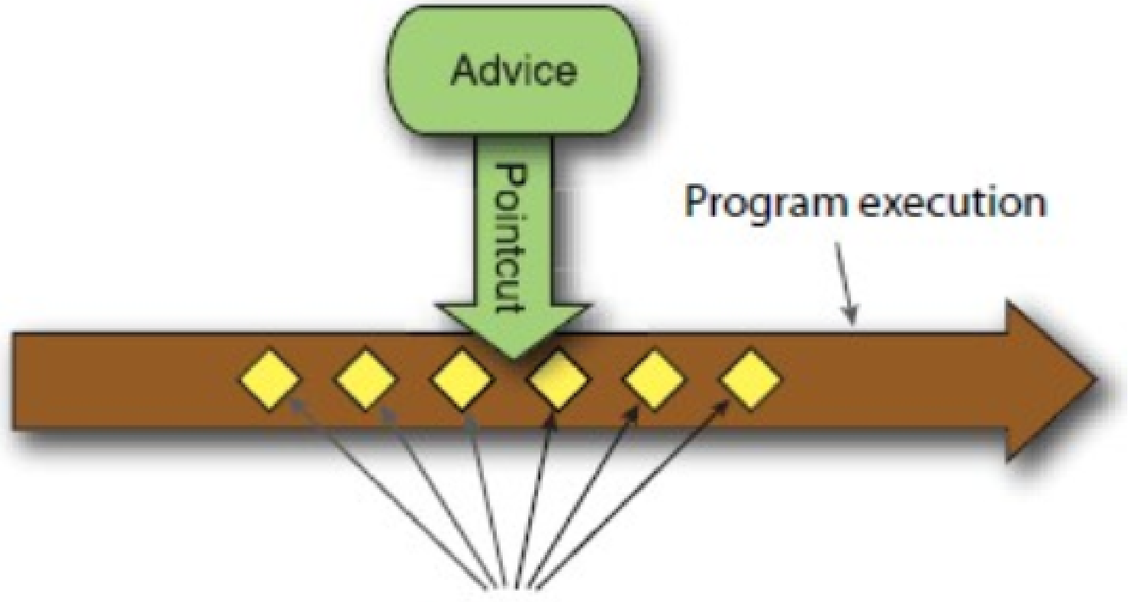
AOP terminology

Join points : are the options on the menu and
pointcuts : are the items you select

Aspect =Advices + Point Cut

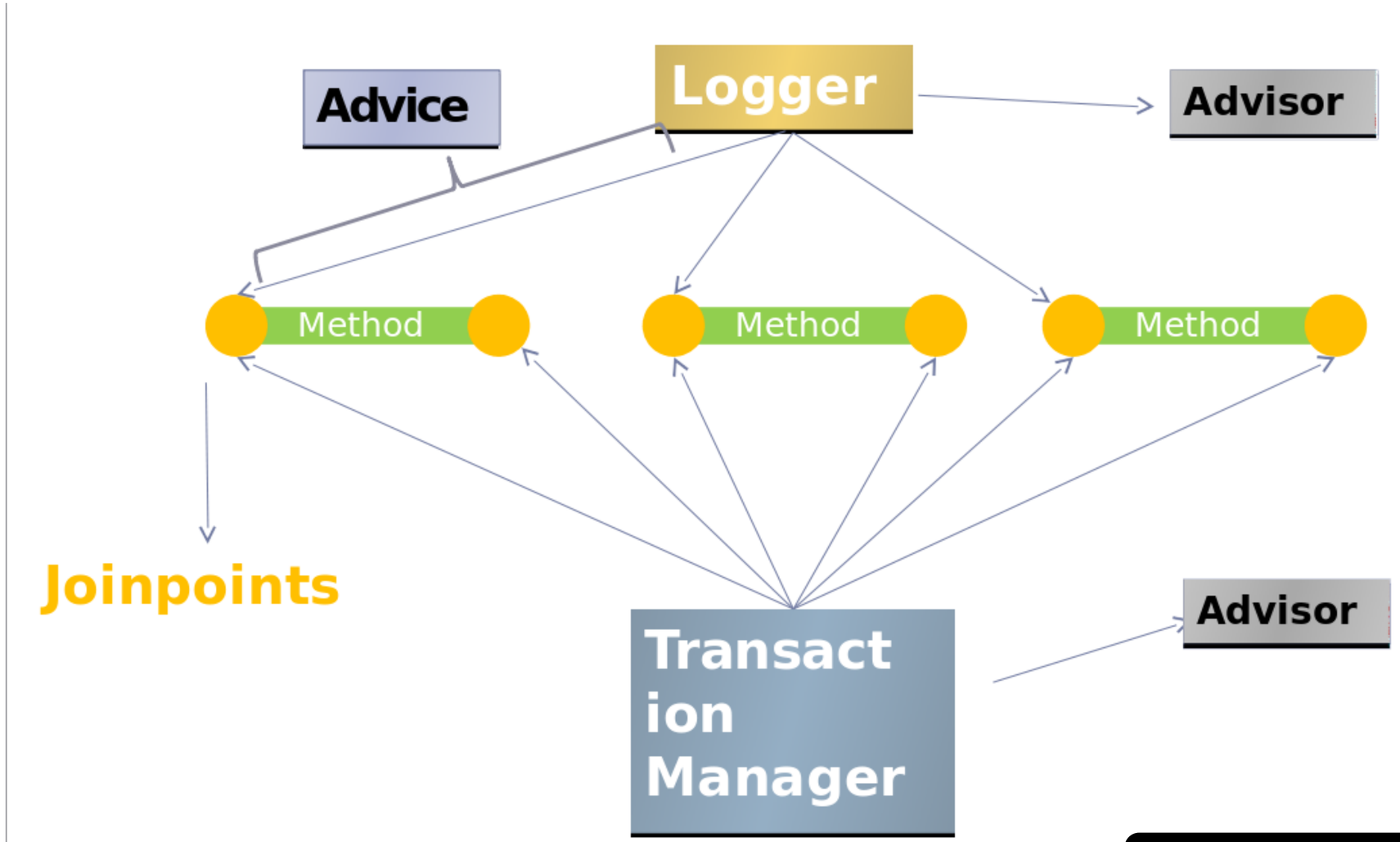
Aspect means
what (extra logic) and where it need to be applied (point cut)

- Aspect
- Joinpoint
- Advice
- Pointcut
- Target
- Object
- AOP Proxy
- Weaving



rgupta.mtech@gmail.com

AOP terminology



rgupta.mtech@gmail.com

Types of Advices

- Before Advice

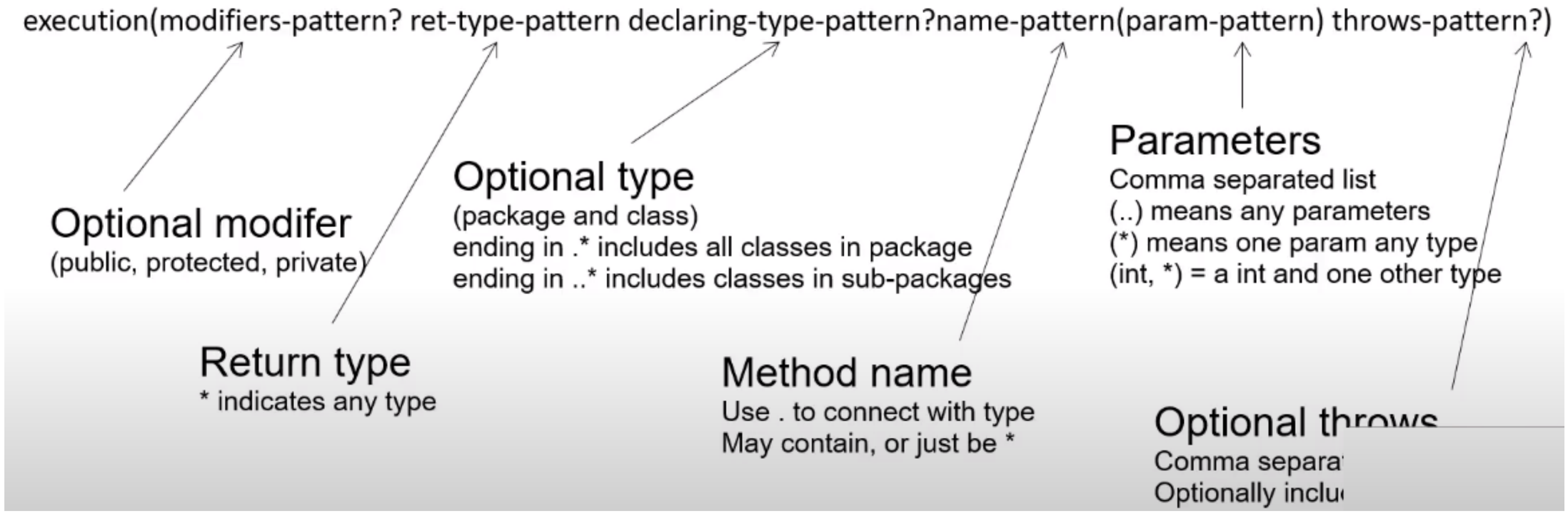
- After returning Advice

- Around Advice

- Throws Advice

rgupta.mtech@gmail.com

Understanding Point Cut wildcard



rgupta.mtech@gmail.com

Module 4: Spring JDBC

rgupta.mtech@gmail.com

Pain of JDBC

1. Define the connection parameters.
2. Access a data source, and establish a connection.
3. Begin a transaction.
4. Specify the SQL statement.
5. Declare the parameters, and provide parameter values.
6. Prepare and execute the statement.
7. Set up the loop to iterate through the results.
8. Do the work for each iteration--execute the business logic.
9. Process any exception.
10. Commit or roll back the transaction.
11. Close the connection, statement, and resultset.

rgupta.mtech@gmail.com

Pain of JDBC

```
    Connection connection=null;
    Statement stmt =null;
    ResultSet rs =null;
    try {
         connection = dataSource.getConnection();
         stmt = connection.createStatement();
         rs = stmt.executeQuery("select * from account");
        while (rs.next()) {
            accounts.add(new Account(rs.getInt("id"), rs.getString("name"), rs.getDouble("balance")));
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }finally {
        if(stmt!=null) {
            try {
                stmt.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        if(rs!=null) {
            try {
                rs.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }if(connection!=null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    return accounts;
}
```

rgupta.mtech@gmail.com

Boilerplate Code?

In computer programming, **boilerplate code** or **boilerplate** refers to sections of code that have to be included in many places with little or no alteration. It is often used when referring to languages that are considered *verbose*, i.e. the programmer must write a lot of code to do minimal jobs.

For instance, a lawyer may give you a five page contract to sign, but most of the contract is boilerplate — meaning it's the same for everyone who gets that contract, with only a few lines changed here and there.

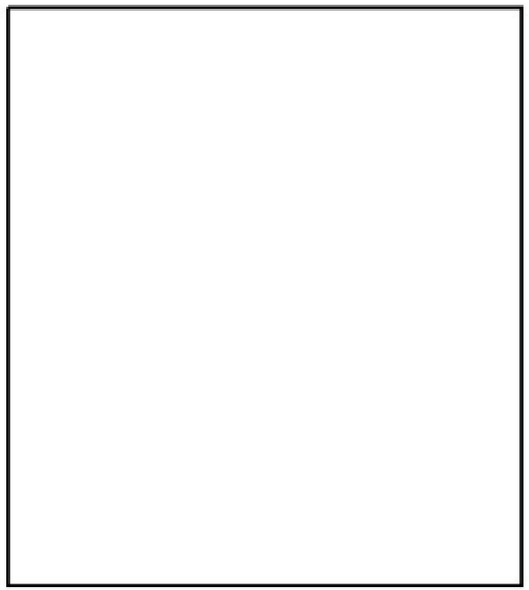
rgupta.mtech@gmail.com

JDBC vs Spring JDBC

JDBC	Spring
DriverManager / DataSource	DataSource
Statement / PreparedStatement / CallableStatement	JdbcTemplate / SimpleJdbcTemplate, SimpleJdbcCall, SimpleJdbcInsert MappingSqlQuery / StoredProcedure
ResultSet / RowSet	POJOs / List of POJOs or Maps / SqlRowSet

rgupta.mtech@gmail.com

Spring JDBC uses Template Design Pattern



Training Evaluation Form

Date of Presentation:

Presenter's Name:

Topic or Session:

Please complete the evaluation for today's training session – your feedback is valuable
AusDBF is committed to continual improvement and suggestions will be considered

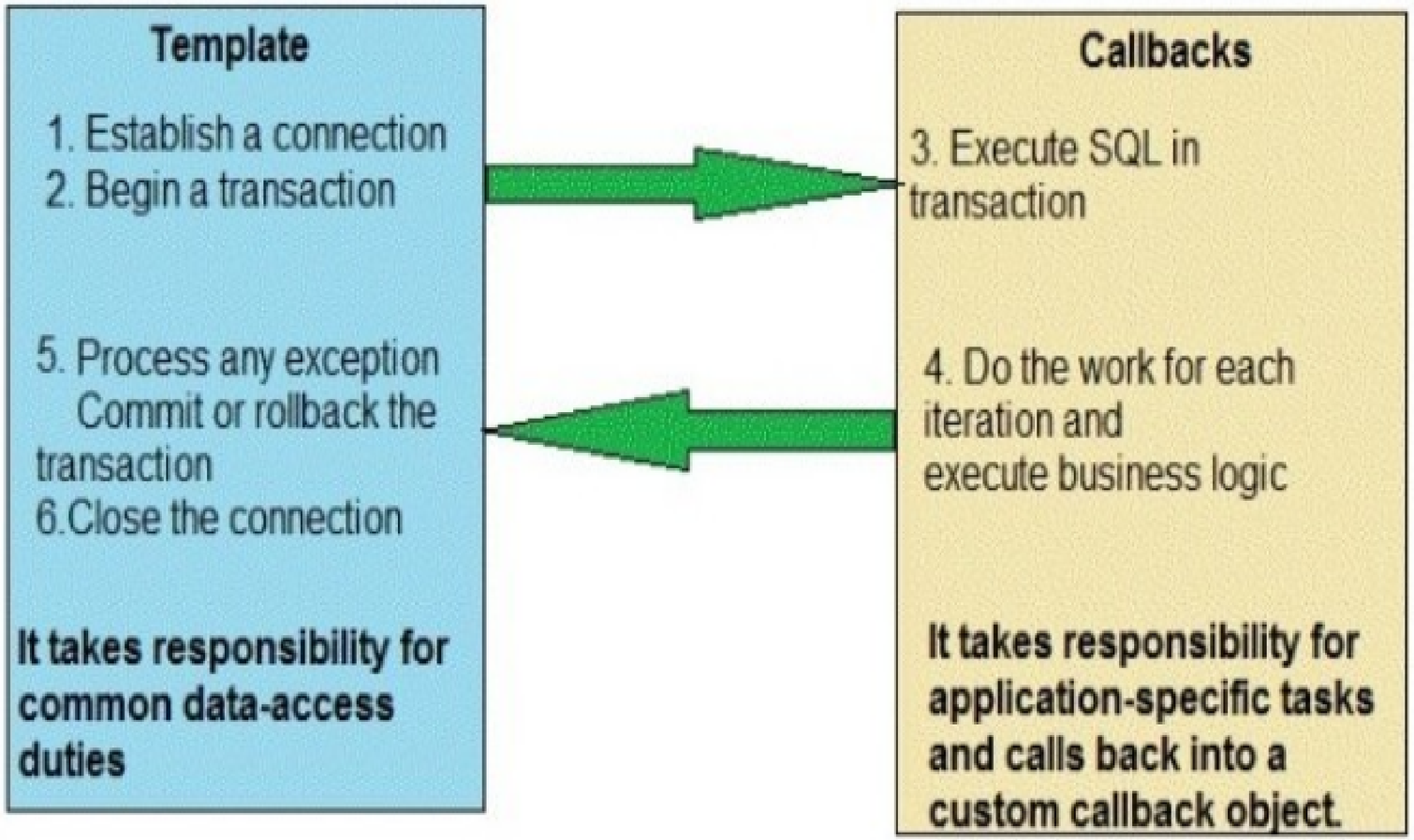
Criteria	Strongly agree 4	Agree 3	Disagree 2
Training was relevant to my needs			
Materials provided were helpful			
Length of training was sufficient			
Content was well organised			
Questions were encouraged			
Instructions were clear and understandable			
Training met my expectations			
The presenter and / or presentation was effective			

Which one is easy for you to provide feedback?

```
@Override
public void save(Account account) {
    String sql="insert into account(id, name, balance) values (?,?,?)";
    jdbcTemplate=new JdbcTemplate(dataSource);
    jdbcTemplate.update(sql, new Object[] {account.getId(),
            account.getName(), account.getBalance()});
}
```

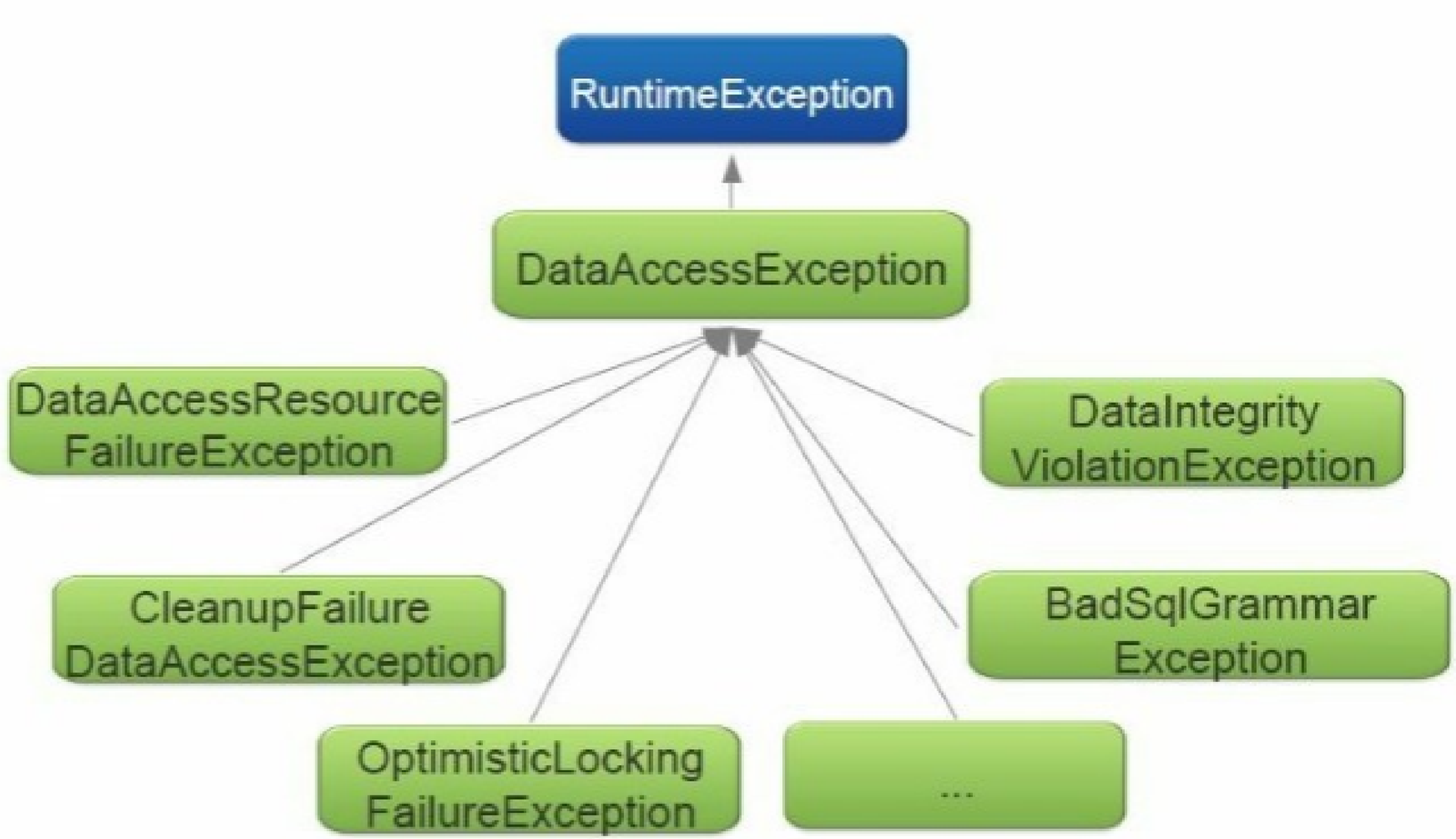
rgupta.mtech@gmail.com

Spring JDBC Template



rgupta.mtech@gmail.com

Exception Hierarchy Spring JDBC



rgupta.mtech@gmail.com

Spring JDBC Example

```java
@Override
public List<Account> getAllAccounts() {
    List<Account> accounts = new ArrayList<Account>();
    Connection connection=null;
    try {
         connection = dataSource.getConnection();

        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery("select * from account2");
        while (rs.next()) {
            accounts.add(new Account(Integer.parseInt(rs.getString("id")),
                    rs.getString("name"),
                    Integer.parseInt(rs.getString("balance"))));
        }
        System.out.println("conn is obtained...");
    } catch (SQLException e) {
        e.printStackTrace();
    }finally{
        if(connection!=null){
            try {
                connection.close();
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

    return accounts;
}
```

Less code less bug

```java
@Override
public List<Account> getAllAccounts() {

    template =new JdbcTemplate(dataSource);

    List<Account>accounts=template.query("select * from account2", new AccountRowMapper());
    return accounts;

}
```

rgupta.mtech@gmail.com

Spring JDBC Example

```
@Override
public Account getAccount(int id) {
    template=new JdbcTemplate(dataSource);
    Account account=template.queryForObject("select * from account2 where id=?",
            new AccountRowMapper(), id);

    return account;
}
```

Update account

```
@Override
public void update(Account account) {

    template=new JdbcTemplate(dataSource);
    template.update("update account2 set balance=? where id=?", new Object[]{account.get

}
```

```
public class AccountRowMapper implements RowMapper<Account>{

    @Override
    public Account mapRow(ResultSet rs, int no) throws SQLException {
        Account account=new Account();
        account.setId(rs.getInt("id"));
        account.setBalance(rs.getInt("balance"));
        account.setName(rs.getString("name"));
        return account;
    }

}
```

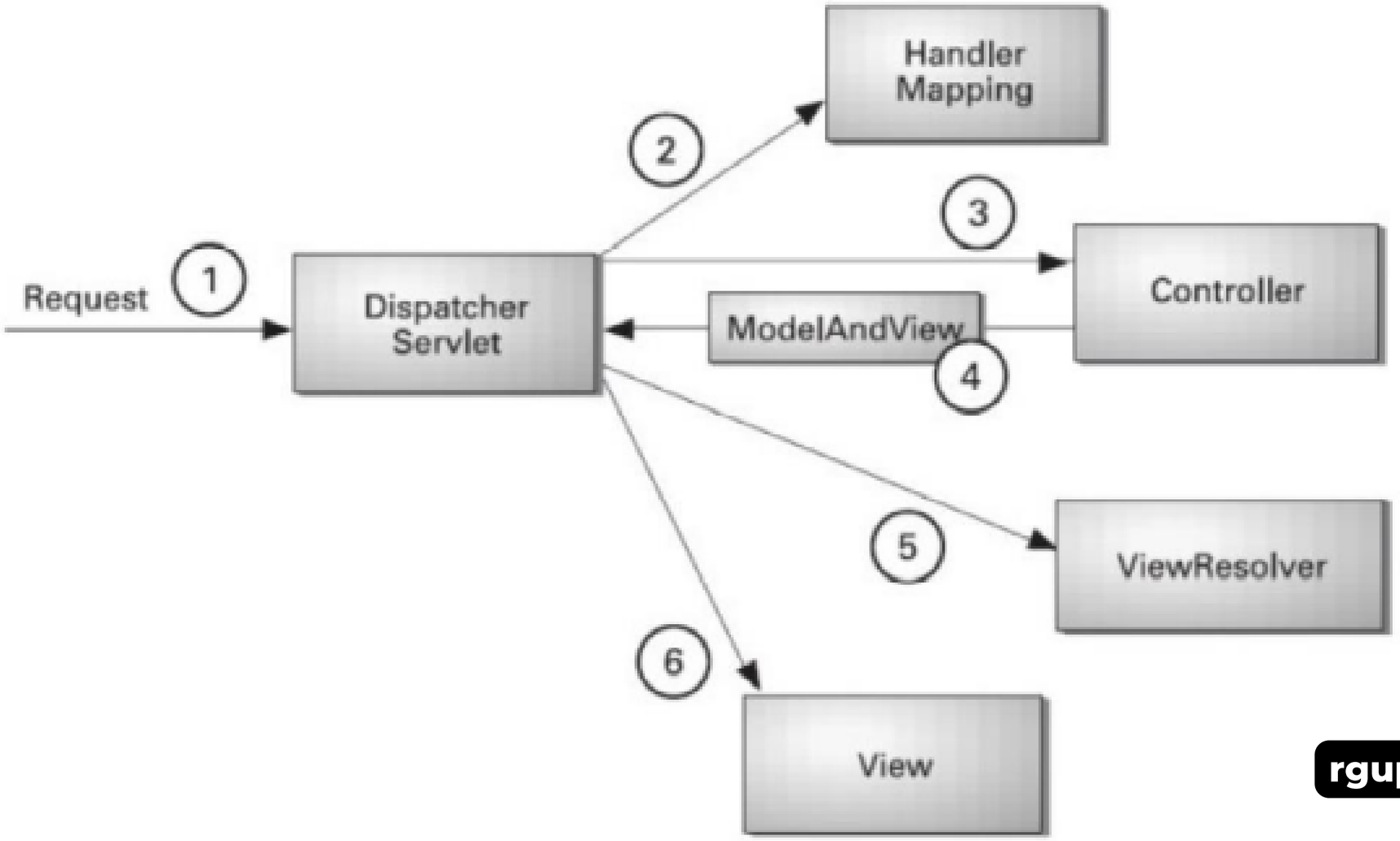
```
@Override
public void addAccount(Account account) {
    String sql="insert into account(id, name, balance) values (?,?,?)";
    jdbcTemplate =new JdbcTemplate(dataSource);
    jdbcTemplate.update(sql, new Object[] {account.getId(), account.getN
```

rgupta.mtech@gmail.com

Module 5: Spring MVC

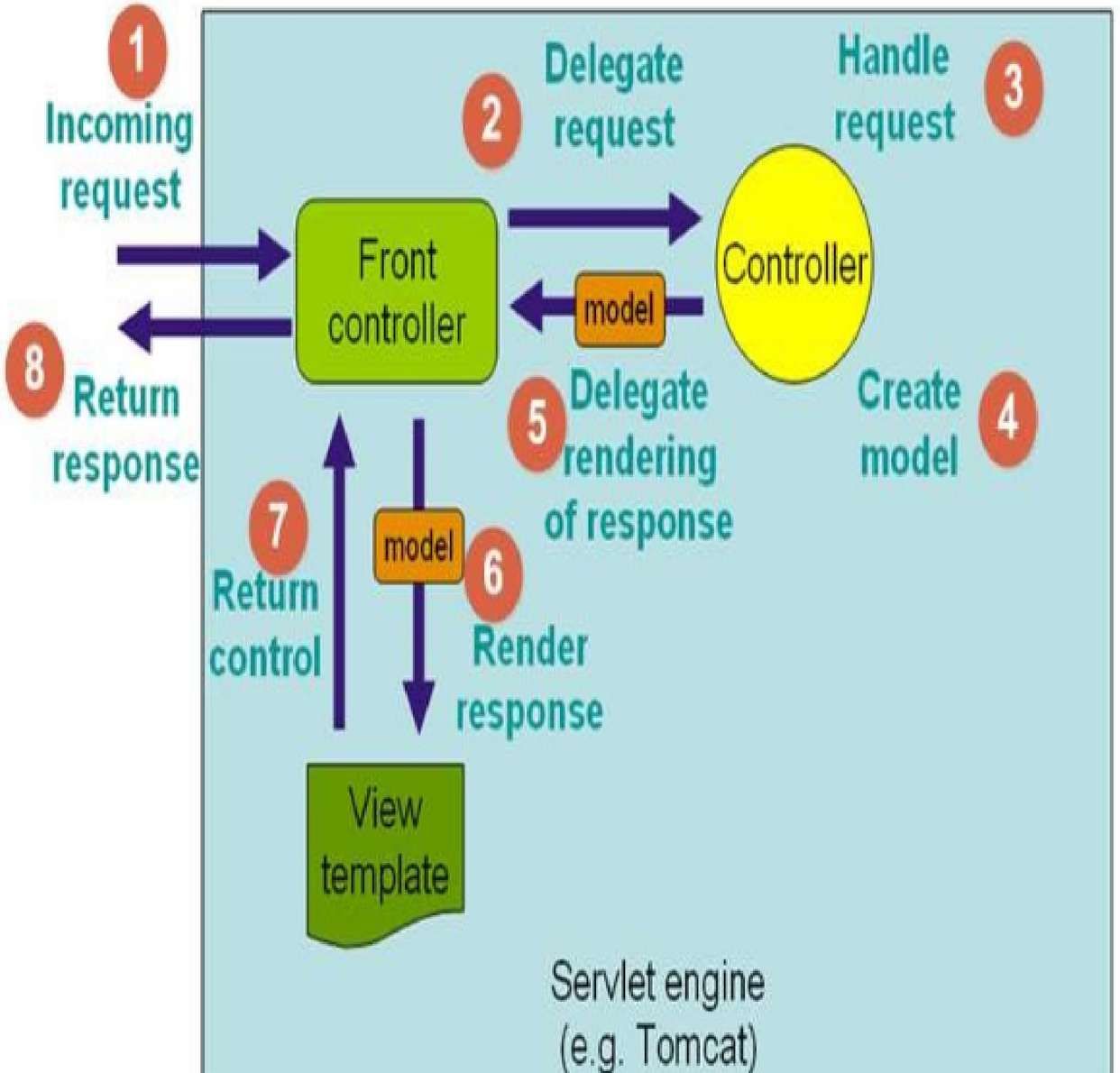
rgupta.mtech@gmail.com

Spring MVC Architecture



rgupta.mtech@gmail.com

Request Flow Spring MVC



- Any incoming request that comes to the web application will be sent to Front Controller (Dispatcher Servlet)
- Front Controller decides to whom (Controller) it has to hand over the request, based on the request headers.
- Controller that took the request, processes the request, by sending it to suitable service class.
- After all processing is done, Controller receives the model from the Service or Data Access layer.
- Controller sends the model to the Front Controller (Dispatcher Servlet).
- Dispatcher servlet finds the view template, using view resolver and send the model to it.
- Using View template, model and view page is build and sent back to the Front Controller.
- Front controller sends the constructed view page to the browser to render it for the user requested.

rgupta.mtech@gmail.com

Spring MVC Configuration steps

Step 1: Configure the web.xml with DispatcherServlet

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
</context-param>

<listener>
    <listener-class>
                org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

Step 2: Configure the dispatcher-servlet.xml

```
<context:component-scan base-package="com.controller" />

  <bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">

      <property name="prefix">
          <value>/WEB-INF/views/</value>
      </property>
      <property name="suffix">
          <value>.jsp</value>
      </property>
  </bean>

</beans>
```

rgupta.mtech@gmail.com

Spring MVC Configuration steps

Step 3: Write Back Controller

```
@Controller
public class HelloWorld {
    @RequestMapping("/helloworld")
    public ModelAndView helloWord() {
        String message = "Hello World, Spring 3.0!";
        return new ModelAndView("helloworld", "message", message);
    }
}
```

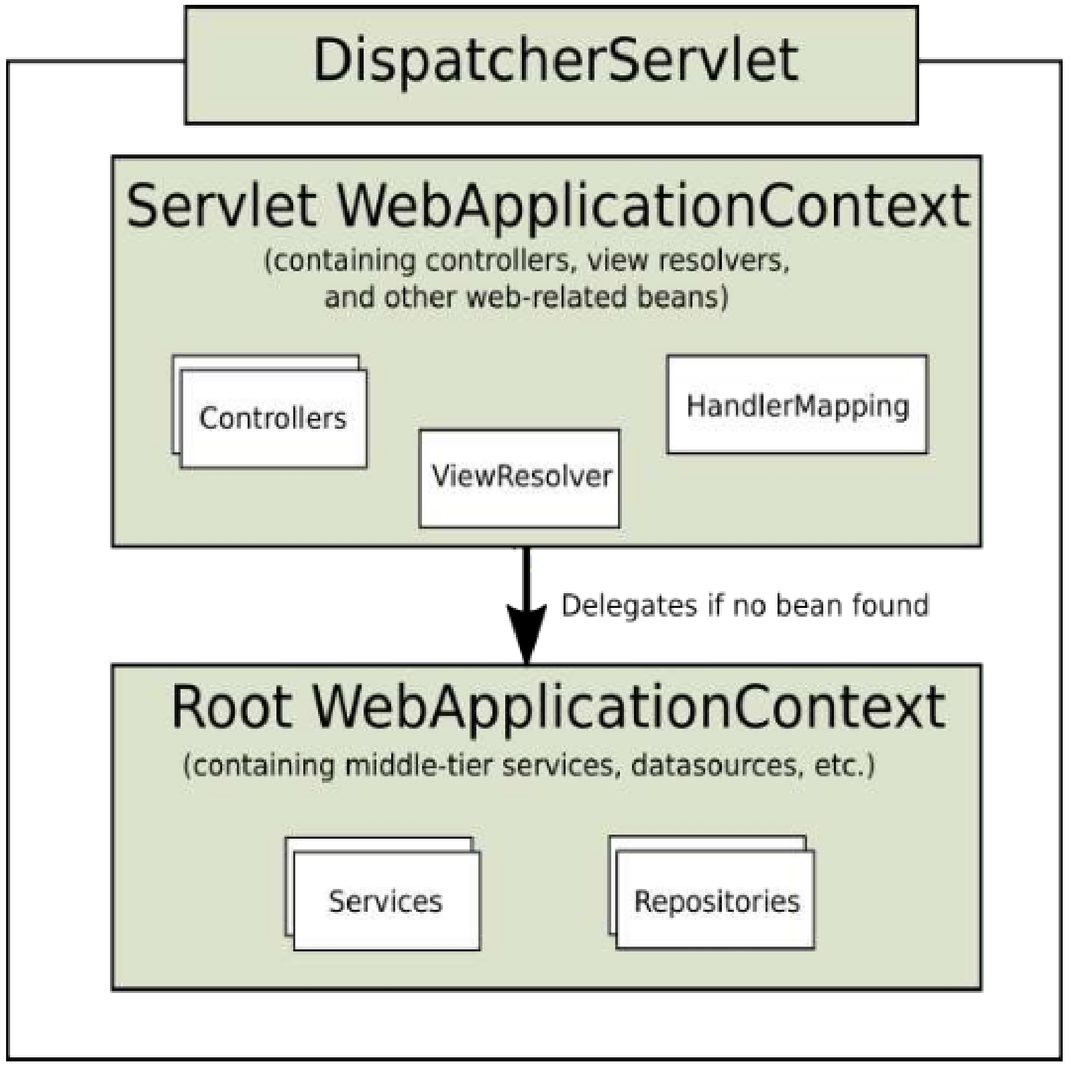
```
@Controller
@RequestMapping("/welcome")
public class HelloController {

        @RequestMapping(method = RequestMethod.GET)
        public String printWelcome(ModelMap model) {

                model.addAttribute("message", "Spring 3 MVC Hello World");
                return "hello";

        }
```

Step 2: Write JSP Page

```
<meta http-equiv="Content-Type" content=
<title>Insert title here</title>
</head>
<body>
${message}
</body>
</html>
```

rgupta.mtech@gmail.com

WebApplicationContext vs RootApplicatoinContext



Root Config Classes are actually used to Create Beans which are Application Specific and which needs to be available for Filters (As Filters are not part of Servlet).

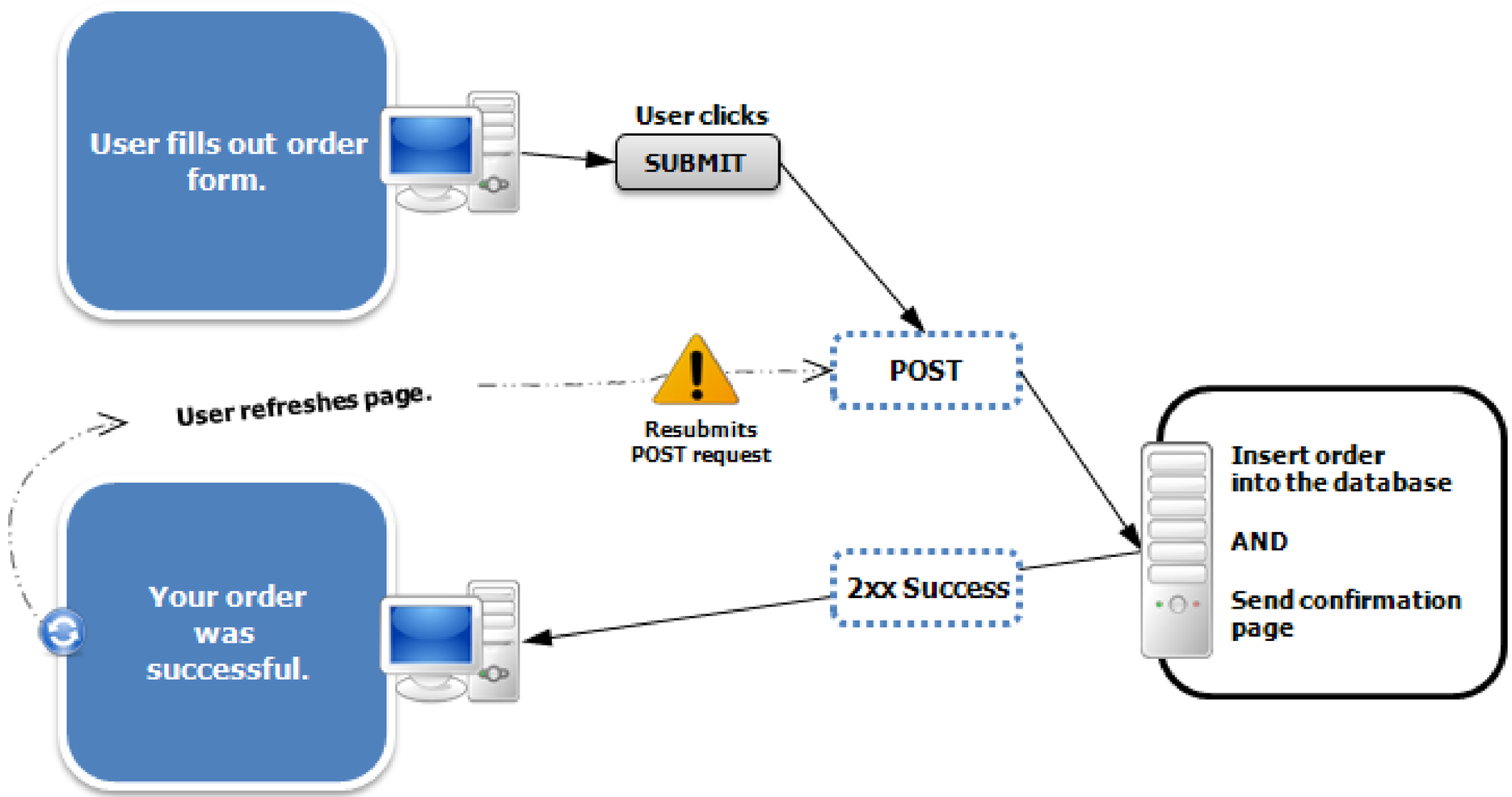
Servlet Config Classes are actually used to Create Beans which are DispatcherServlet specific such as ViewResolvers, ArgumentResolvers, Interceptor, etc.

Root Config Classes will be loaded first and then Servlet Config Classes will be loaded.

Root Config Classes will be the Parent Context

rgupta.mtech@gmail.com

The Post/Redirect/Get design pattern



rgupta.mtech@gmail.com

Spring MVC Java Configuration steps

Step 1: Configure Dispacher Servlet and bootstap the application

```
 4
 5  public class WebInit extends AbstractAnnotationConfigDispatcherServletInitializer{
 6
 7      @Override
 8      protected Class<?>[] getRootConfigClasses() {
 9          return null;
10      }
11
12      @Override
13      protected Class<?>[] getServletConfigClasses() {
14          return new Class[] {MvcConfig.class};
15      }
16
17      @Override
18      protected String[] getServletMappings() {
19          return new String[] {"/"};
20      }
21
22  }
```

Step 1: Configure View Resolver

```
@ComponentScan(basePackages = { "com.demo" })
public class MvcConfig extends WebMvcConfigurerAdapter{

      @Bean
        public InternalResourceViewResolver getInternalResourceViewResolver() {
            InternalResourceViewResolver resolver = new InternalResourceViewResolver();
            resolver.setPrefix("/WEB-INF/views/");
            resolver.setSuffix(".jsp");
            return resolver;
        }

        @Override
        public void addResourceHandlers(ResourceHandlerRegistry registry) {
             // Don't forget the ending "/" for location or you will hit 404.
            registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");

        }
}
```

rgupta.mtech@gmail.com

Module 6: Spring REST Fundamentals

rgupta.mtech@gmail.com

Why Spring Boot?-Too much configuration

```java
@Configuration
@ComponentScan(basePackages={"com.bookapp"})
@EnableAspectJAutoProxy
@PropertySource(value="db.properties")
@EnableTransactionManagement
public class ModelConfig {

    @Autowired
    private Environment environment;

    @Bean(name="dataSource")
    public DataSource getDataSource(){
        DriverManagerDataSource ds=new DriverManagerDataSource();
        ds.setDriverClassName(environment.getProperty("driver"));
        ds.setUrl(environment.getProperty("url"));
        ds.setUsername(environment.getProperty("username"));
        ds.setPassword(environment.getProperty("password"));
        return ds;
    }

    @Bean
    public LocalSessionFactoryBean getSessionFactory(){
        LocalSessionFactoryBean sf=new LocalSessionFactoryBean();
        sf.setDataSource(getDataSource());
        sf.setPackagesToScan("com.bookapp.model.persistance");
        sf.setHibernateProperties(getHibernateProperties());
        return  sf;
    }
```

```java
    public Properties getHibernateProperties() {
        Properties properties=new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "validate");
        properties.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
        properties.setProperty("hibernate.show_sql", "true");
        properties.setProperty("hibernate.formate_sql", "true");
        return properties;
    }

    @Bean
    public PersistenceExceptionTranslationPostProcessor
    getPersistenceExceptionTranslationPostProcessor(){
        PersistenceExceptionTranslationPostProcessor ps=
                new PersistenceExceptionTranslationPostProcessor();
        return ps;
    }
```

rgupta.mtech@gmail.com

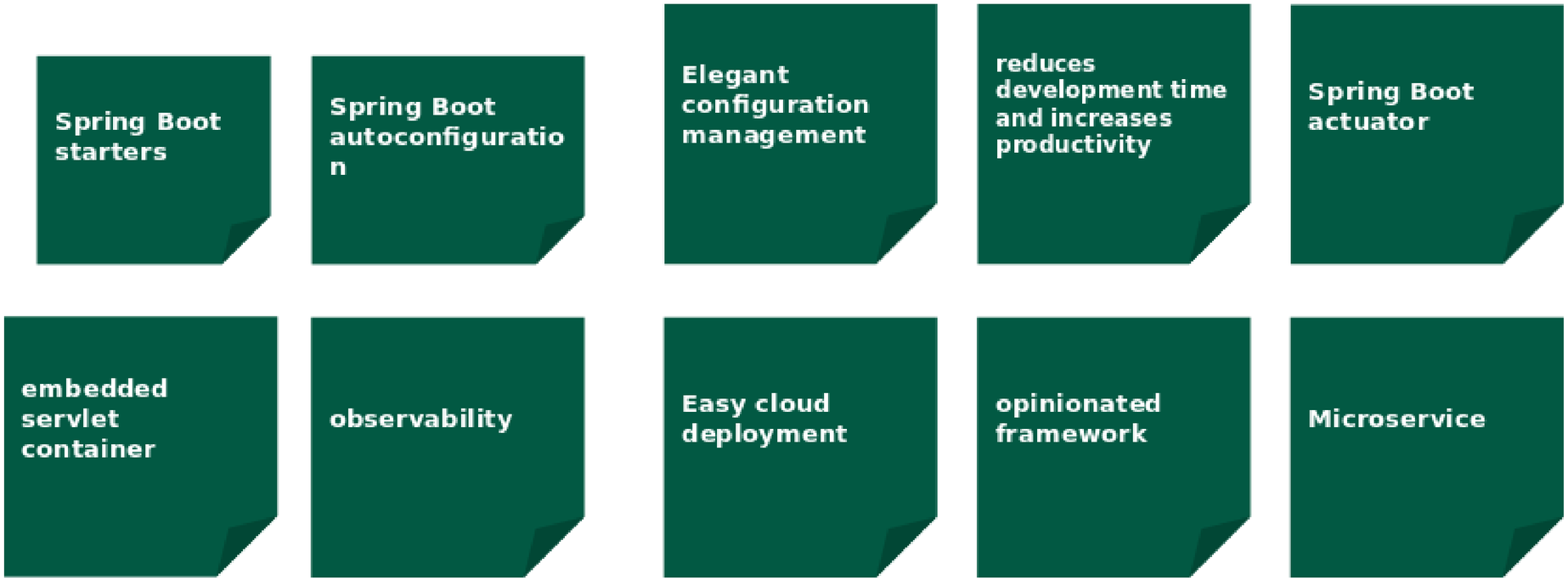
What is Spring Boot?

Spring Boot = Spring Framework + Embedded HTTP Servers (Tomcat, Jetty) - XML <bean> Configuration or @Configuration

rgupta.mtech@gmail.com

Spring Boot Features/Advantage

- Spring Boot is an opinionated framework that helps developers build Spring-based applications quickly and easily.
- Spring Boot is an open-source Java-based framework that is used to create stand-alone, production-ready applications that can be easily deployed to the cloud. It is built on top of the popular Spring Framework and provides developers with an easy and fast way to develop microservices and web applications.
- It comes with a pre-configured environment that includes an embedded server, a database,and other dependencies, reducing the amount of boilerplate code that developers need to write.



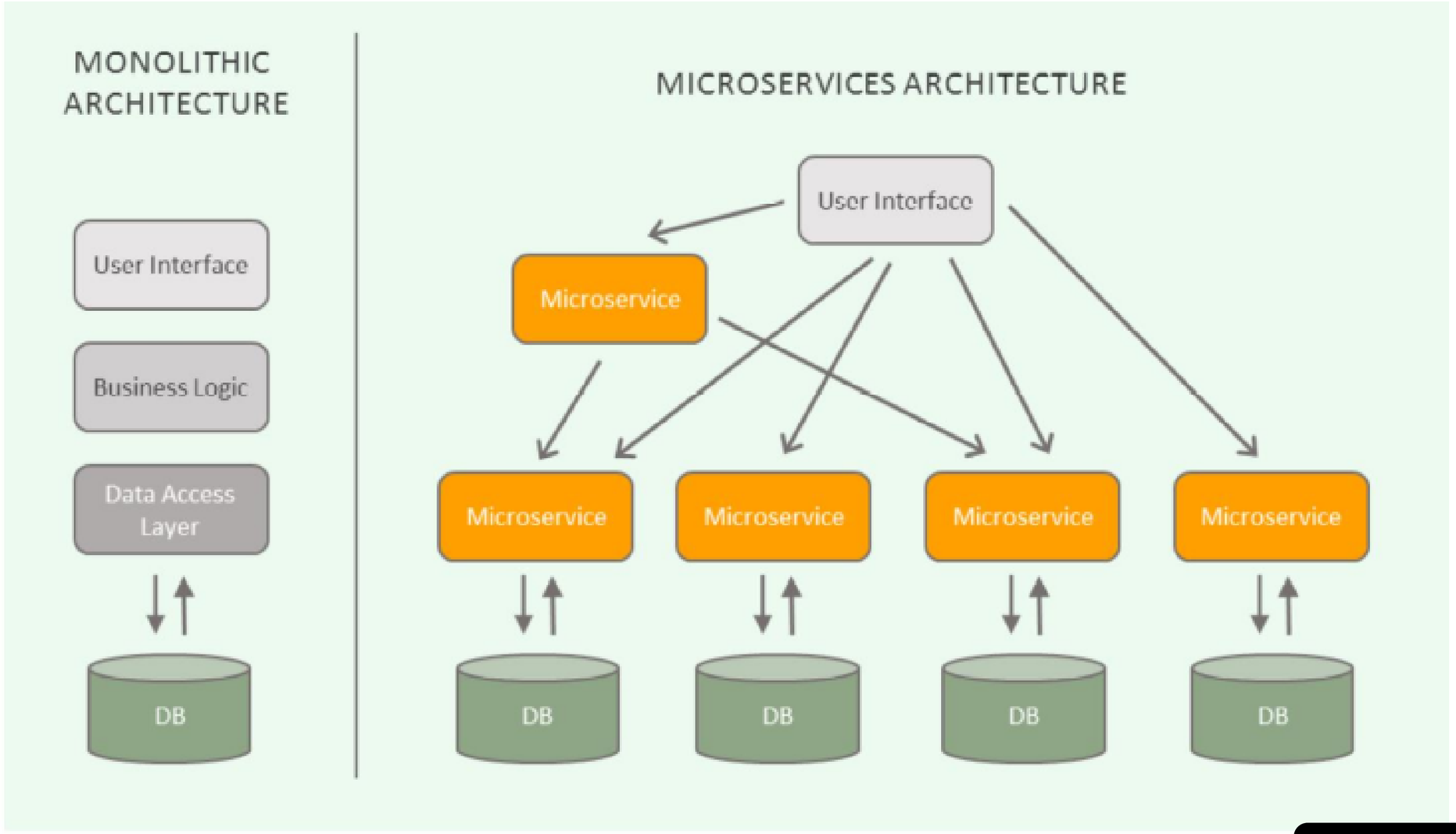
rgupta.mtech@gmail.com

New Features Spring Boot 3

Spring Boot 3.0 was released on November 2022. Here is a consolidated list of changes introduced/updated to the framework.

rgupta.mtech@gmail.com

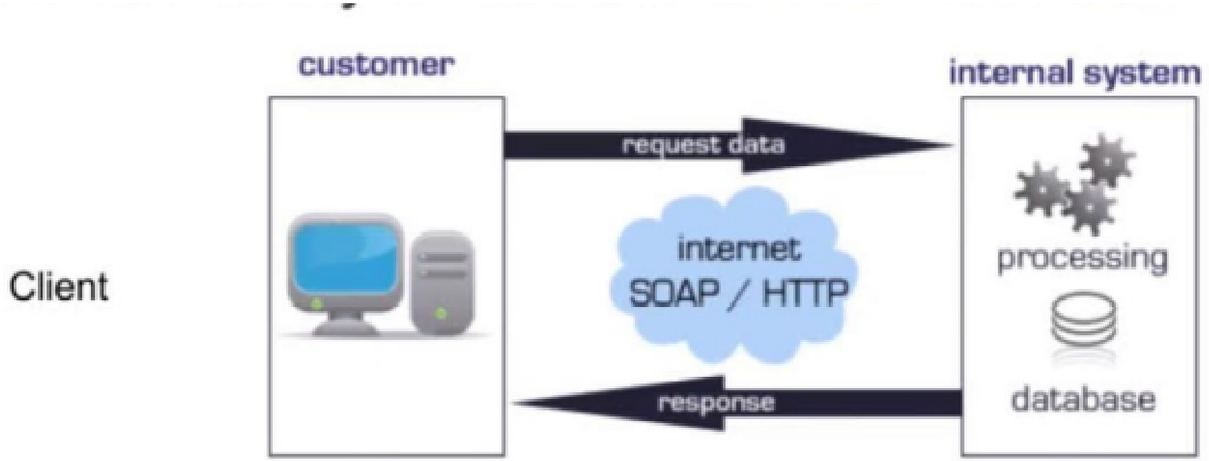
Spring Boot & microservice



rgupta.mtech@gmail.com

What is web service ?

Web services provide a common plateform that allow multiple applications build on various programming lannuages to have the ability to communiate with each other



rgupta.mtech@gmail.com

API (Application Programming Interface)

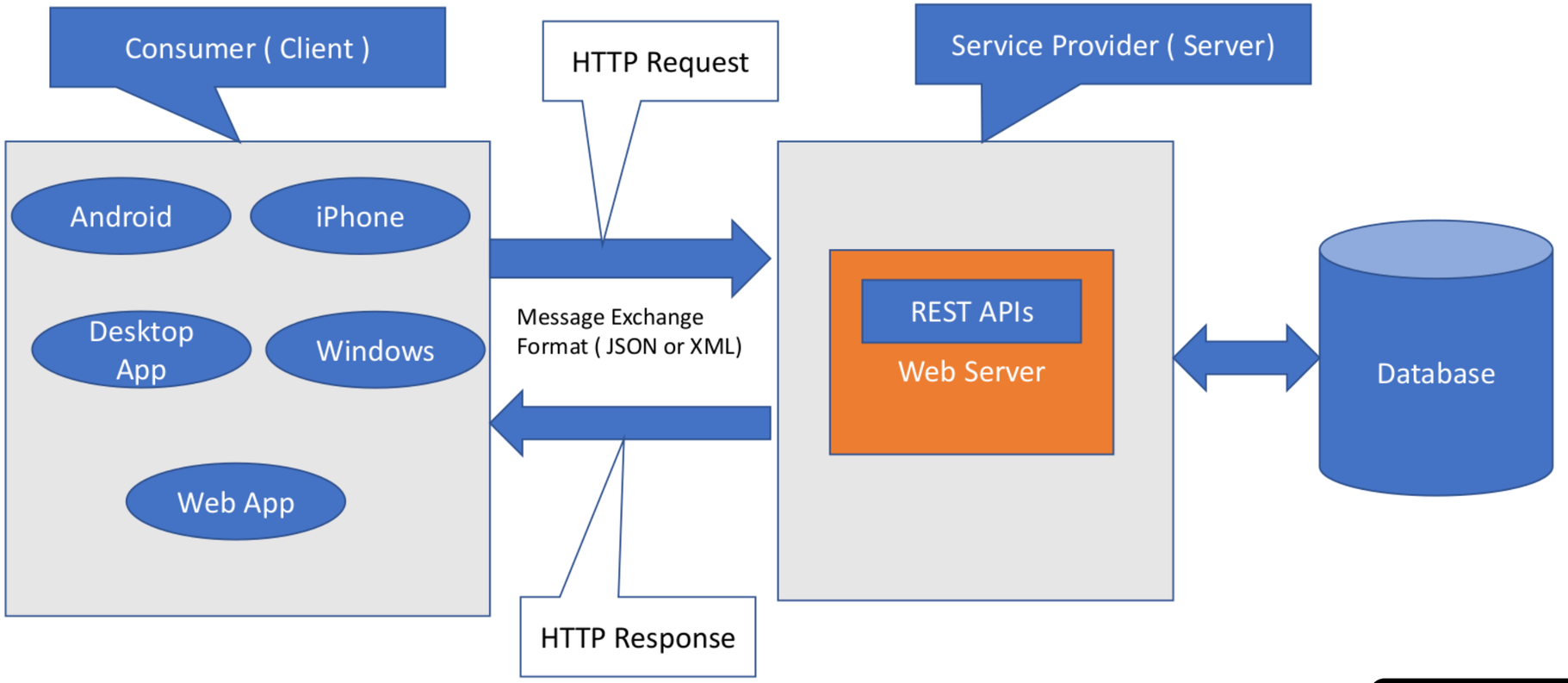
It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allow the client to talk to it.

rgupta.mtech@gmail.com

What is REST?

The REST stands for Representational State Transfer

- **State means data**
- **REpresentational means formats (such as xml, json, yaml, html, etc)**
- **Transfer means carry data between consumer and provider using HTTP protocol**



rgupta.mtech@gmail.com

REST - Representational State Transfer

REST was originally coined by Roy Fielding, who was also the inventor of the HTTP protocol.

• A REST API is an intermediary Application Programming Interface that enables two applications to communicate with each other over HTTP, much like how servers communicate to browsers.

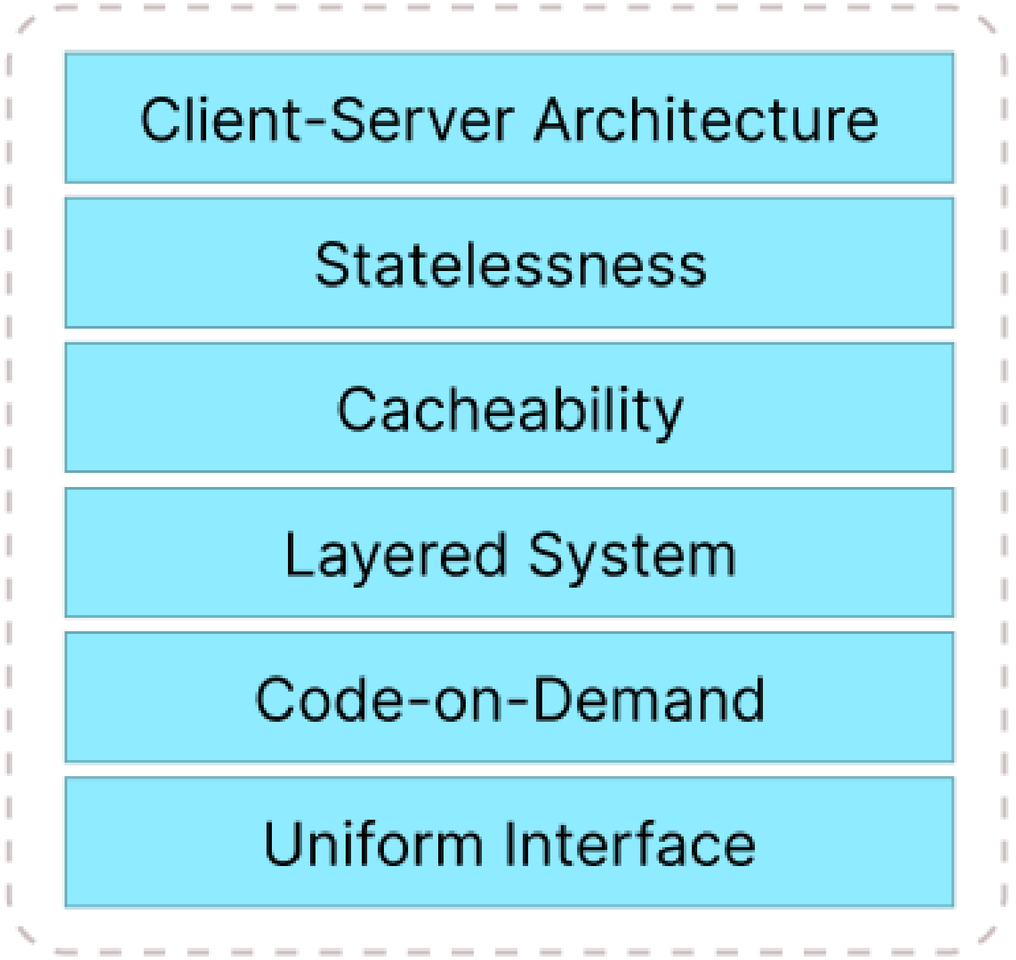
• The REST architectural style has quickly become very popular over the world for designing and architecting applications that can communicate.

• The need for REST APIs increased a lot with the drastic increase of mobile devices. It became logical to build REST APIs and let the web and mobile clients consume the API instead of developing separate applications.

rgupta.mtech@gmail.com

REST Architectural Constraints

6 Characteristics of a REST API



rgupta.mtech@gmail.com

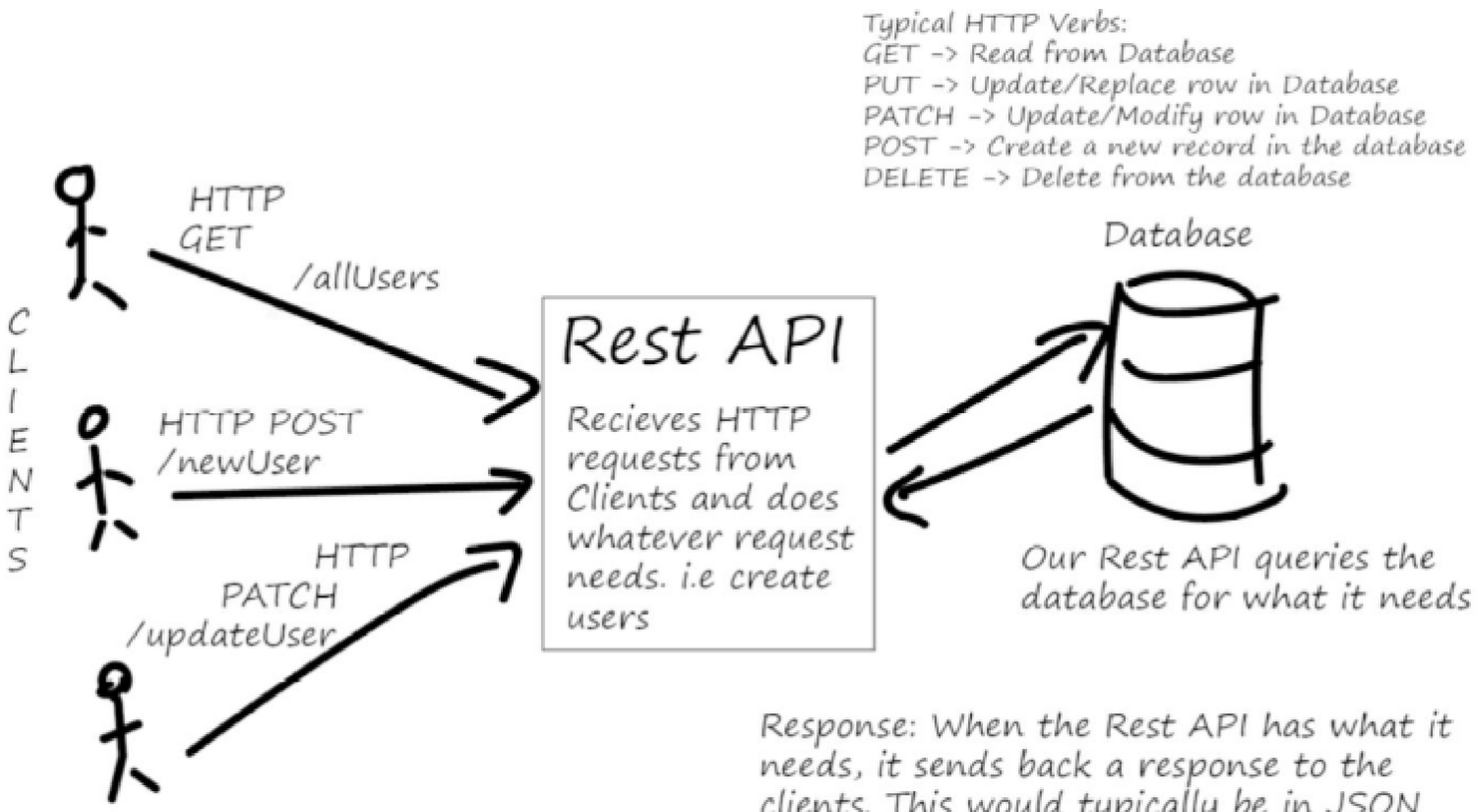
REST Architectural Constraints

An API that has following constraints is known as RESTful API:

- Client-server architecture: The client is the front-end and the server is the back-end of the service. It is important to note that both of these entities are independent of each other.
- Stateless: No data should be stored on the server during the processing of the request transfer. The state of the session should be saved at the client's end.
- Cacheable: The client should have the ability to store responses in a cache. This greatly improves the performance of the API.
- Uniform Interface: This constraint indicates a generic interface to manage all the interactions between the client and server in a unified way, which simplifies and decouples the architecture.
- Layered System: The server can have multiple layers for implementation. This layered architecture helps to improve scalability by enabling load balancing.
- Code on Demand: This constraint is optional. This constraint indicates that the functionality of the client applications can be extended at runtime by allowing a code download from the server and executing the code.

rgupta.mtech@gmail.com

REST Basics



rgupta.mtech@gmail.com

Common Http Status code

Sr. No.	API Name	HTTP Method	Path	Status Code	Description
(1)	GET Users	GET	`/api/v1/users`	200 (OK)	All User resources are fetched.
(2)	POST User	POST	`/api/v1/users`	201 (Created)	A new User resource is created.
(3)	GET User	GET	`/api/v1/users/{id}`	200 (OK)	One User resource is fetched.
(4)	PUT User	PUT	`/api/v1/users/{id}`	200 (OK)	User resource is updated.
(5)	DELETE User	DELETE	`/api/v1/users/{id}`	204 (No Content)	User resource is deleted.

rgupta.mtech@gmail.com

Http Status code in details

HTTP Status Codes

For great REST services the correct usage of the correct HTTP status code in a response is vital.

1xx – Informational

This class of status code indicates a provisional response, consisting only of the Status-Line and optional headers, and is terminated by an empty line

100 – Continue
101 – Switching Protocols
102 – Processing

2xx – Successful

This class of status code indicates that the client's request was successfully received, understood, and accepted.

200 – OK
201 – Created
202 – Accepted
203 – Non-Authoritative Information
204 – No Content
205 – Reset Content
206 – Partial Content
207 – Multi-Status

3xx – Redirection

This class of status code indicates that further action needs to be taken by the user agent in order to fulfill the request.

300 – Multiple Choices
301 – Moved Permanently
302 – Found
303 – See Other
304 – Not Modified
305 – Use Proxy
307 – Temporary Redirect

4xx – Client Error

The 4xx class of status code is intended for cases in which the client seems to have erred.

400 – Bad Request
401 – Unauthorised
402 – Payment Required
403 – Forbidden
404 – Not Found
405 – Method Not Allowed
406 – Not Acceptable
407 – Proxy Authentication Required
408 – Request Timeout
409 – Conflict
410 – Gone
411 – Length Required
412 – Precondition Failed
413 – Request Entity Too Large
414 – Request URI Too Long
415 – Unsupported Media Type
416 – Requested Range Not Satisfiable
417 – Expectation Failed
422 – Unprocessable Entity
423 – Locked
424 – Failed Dependency
425 – Unordered Collection
426 – Upgrade Required

5xx – Server Error

Response status codes beginning with the digit "5" indicate cases in which the server is aware that it has erred or is incapable of performing the request.

500 – Internal Server Error
501 – Not Implemented
502 – Bad Gateway
503 – Service Unavailable
504 – Gateway Timeout
505 – HTTP Version Not Supported
506 – Variant Also Negotiates
507 – Insufficient Storage
510 – Not Extended

Examples of using HTTP Status Codes in REST

201 – When doing a POST to create a new resource it is best to return 201 and not 200.
204 – When deleting a resources it is best to return 204, which indicates it succeeded but there is no body to return.
301 – If a 301 is returned the client should update any cached URI's to point to the new URI.
302 – This is often used for temporary redirect's, however 303 and 307 are better choices.
409 – This provides a great way to deal with conflicts caused by multiple updates.
501 – This implies that the feature will be implemented in the future.

Special Cases

306 – This status code is no longer used. It used to be for switch proxy.
418 – This status code from RFC 2324. However RFC 2324 was submitted as an April Fools' Joke. The message is *I am a teapot*.

Key	Description
Black	HTTP version 1.0
Blue	HTTP version 1.1
Aqua	Extension RFC 2295
Green	Extension RFC 2518
Yellow	Extension RFC 2774
Orange	Extension RFC 2817
Purple	Extension RFC 3648
Red	Extension RFC 4918

Spring boot hello world REST

```java
@RestController
@RequestMapping(path = "api")
public class Hello {

    private Logger logger = LoggerFactory.getLogger(Hello.class);

    @GetMapping(path = "hello/{name}/{address}")
    public String helloWithName(@PathVariable(name = "name") String name,
            @PathVariable(name = "address") String address) {
        return "hello spring rest:" + name + ": " + address;
    }

    @GetMapping(path = "hello")
    public String helloWithRequestParam(
            @RequestParam(name = "name", required = false, defaultValue = "amit") String name,
            @RequestParam(name = "age", required = false, defaultValue = "30") int age) {

        logger.info("spring boot hello world.....");
        return "hello spring rest:" + name + ": " + age;
    }

}
```

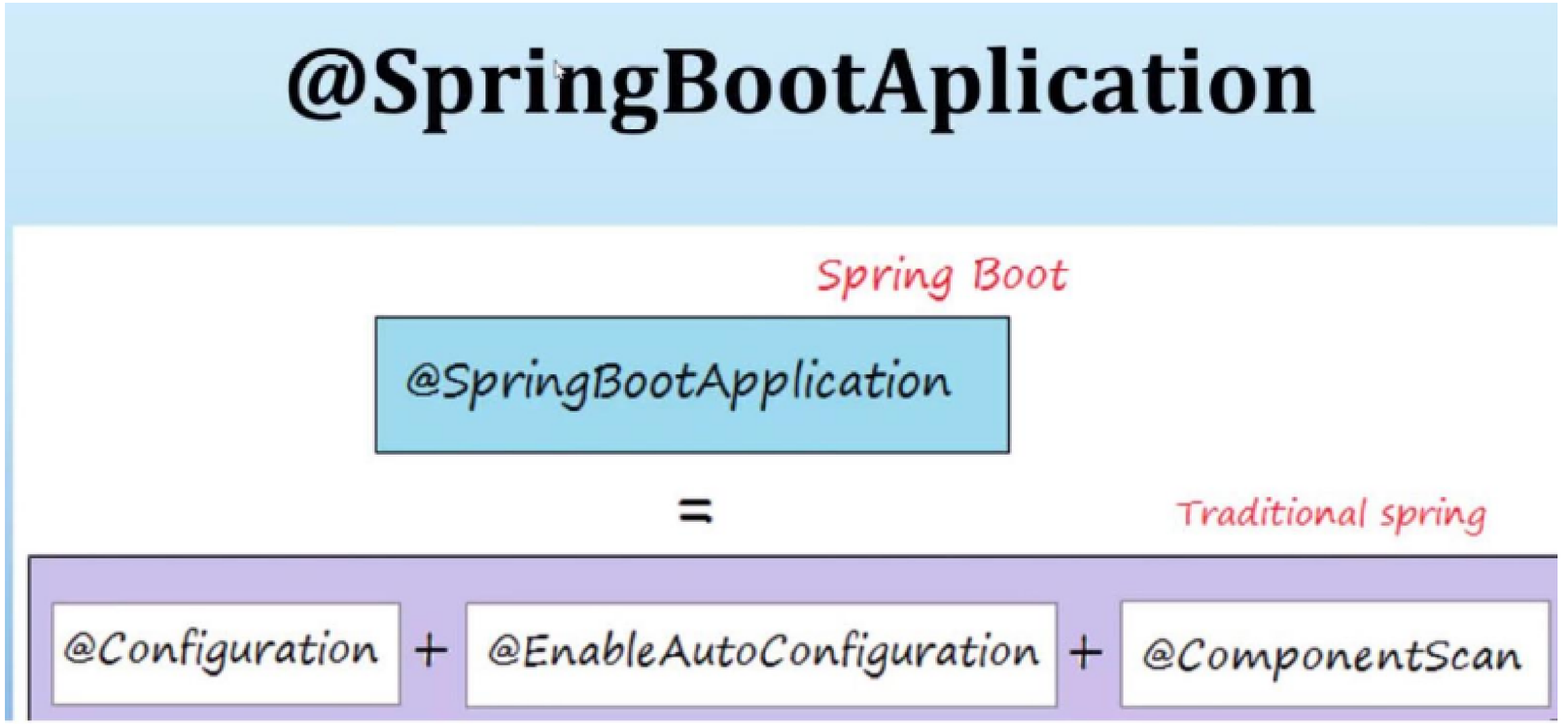
rgupta.mtech@gmail.com

Spring Boot A closer look

- Dependency management using Spring Boot starters
- How auto-configuration works
- Configuration properties
- Overriding auto-configuration
- Using CommandLineRunner

rgupta.mtech@gmail.com

@SpringBootApplication annotation in spring boot



rgupta.mtech@gmail.com

How Spring Boot Application Work Internally?

Main method is not required for the typical deployment scenario of building a war but if you want to launch the application from within an IDE (eg with eclipse run as -> java application) or from command line using command java -jar demo.jar

High level flow of how spring boot works?

From the run method, the main application context is kicked off which in turn searches for the classes annoted with @Configuration, initializes all the declared beans in those configuration classes and store throse beans in JVM, specifically in a space inside JVM which is know as IOC container. After creating of all the beans, automatically configure the dispacher servlet and registers the default handing mapping, messageconverters and all other basic things.

run(...) internal flow

- Create applicationContext
- Check applicationType
- Register the annotated class beans with the context
- Create an instance of TomcatEmbeddedServletContainer and add the context

rgupta.mtech@gmail.com

Spring bean auto configuration

Condition	Description
OnBeanCondition	Checks if a bean is in the Spring factory
OnClassCondition	Checks if a class is on the classpath
OnExpressionCondition	Evalutates a SPeL expression
OnJavaCondition	Checks the version of Java
OnJndiCondition	Checks if a JNDI branch exists
OnPropertyCondition	Checks if a property exists
OnResourceCondition	Checks if a resource exists
OnWebApplicationCondition	Checks if a WebApplicationContext exists

rgupta.mtech@gmail.com

Reading Property file Spring Boot and spring boot profiles

- External properties & Property sources
- Reading property file @Value @ConfigurationProperties
- Using Spring profiles to swap implementation without changing code

rgupta.mtech@gmail.com

REST Key Concepts

REST Key Concepts

- Resource
- Sub-resource
- URI
- HTTP Methods
- HTTP Status Codes

rgupta.mtech@gmail.com

REST Key Concepts: REST - Resource

The fundamental concept of a REST-based system is the resource. A resource is anything you want to expose to the outside world, through your application.

rgupta.mtech@gmail.com

REST Key Concepts: URI - Uniform Resource Identifier

The resource an be identified by a Uniform Resource Identifier (URI). For web-based systems, HTTP is the most commonly used protocol for communicating with external systems.
You can identify a unique resource using a URI.

Consider, we are developing a simple blog application and you can define URIs for a blog Post resource:

GET—http://localhost:8080/api/posts/: Returns a list of all posts
GET—http://localhost:8080/api/posts/2: Returns a post whose ID is 2
POST—http://localhost:8080/api/posts/: Creates a new Post resource
PUT—http://localhost:8080/api/posts/2: Updates a POST resource whose ID is 2
DELETE—http://localhost:8080/api/posts/2: Deletes a POST resource whose ID is 2

rgupta.mtech@gmail.com

REST Key Concepts: REST -Sub-resource

In REST, the relationships are often modeled by a sub-resource. Use the following pattern for sub-resources

```
GET  /{resource}/{resource-id}/{sub-resource}
GET  /{resource}/{resource-id}/{sub-resource}/{sub-resource-id}
POST /{resource}/{resource-id}/{sub-resource}
```

```
GET  /{post}/{post-id}/{comments}
GET  /{post}/{post-id}/{comments}/{comment-id}
POST /{post}/{post-id}/{comments}
```

Use sub-resources child object cannot exist without its parent.

rgupta.mtech@gmail.com

REST Key Concepts: HTTP Methods

HTTP Method	Description	Example
GET	To get a collection or a single resource	http://localhost:8080/api/users http://localhost:8080/api/users/1
POST	To create a new resource	http://localhost:8080/api/users
PUT	To update an existing resource	http://localhost:8080/api/users/1
DELETE	To delete a collection or a single resource	http://localhost:8080/api/users/1

All HTTP Methods at https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

rgupta.mtech@gmail.com

REST Key Concepts:HTTP Status Codes

Level 200 (Success)	Level 400	Level 500
200 : OK	400 : Bad Request	500 : Internal Server Error
201 : Created	401 : Unauthorized	503 : Service Unavailable
203 : Non-Authoritative Information	403 : Forbidden	501 : Not Implemented
204 : No Content	404 : Not Found	504 : Gateway Timeout
	409 : Conflict	599 : Network timeout
		502 : Bad Gateway

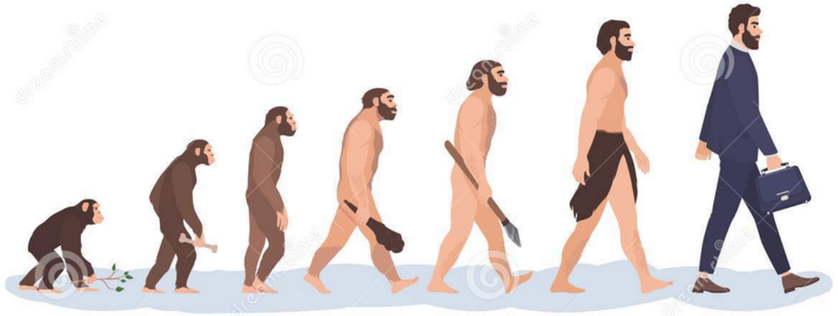
rgupta.mtech@gmail.com

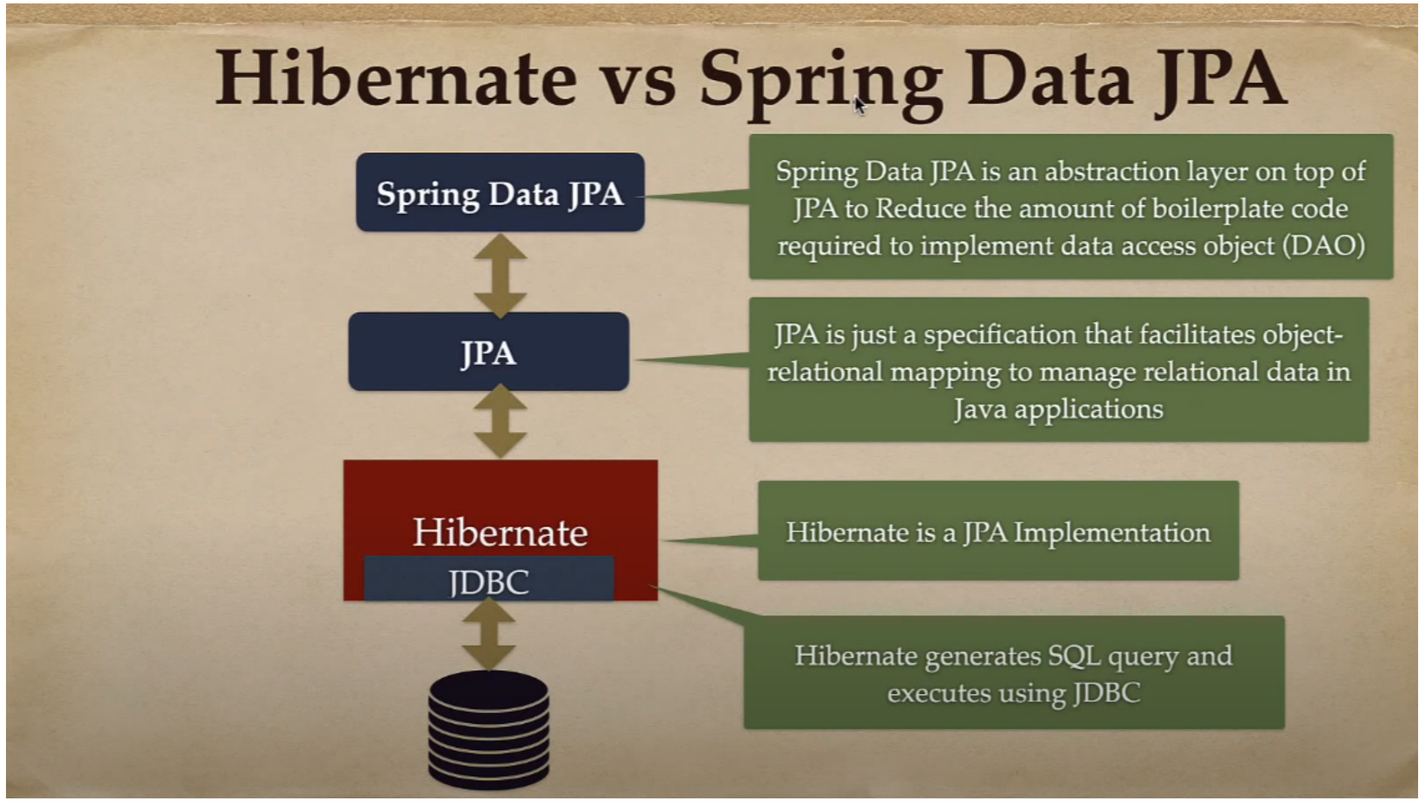
ORM, Hibernate vs JPA

and Spring Data basics

Data Layer Evolution Process

Database layer evolution process is very similer to human evolution process

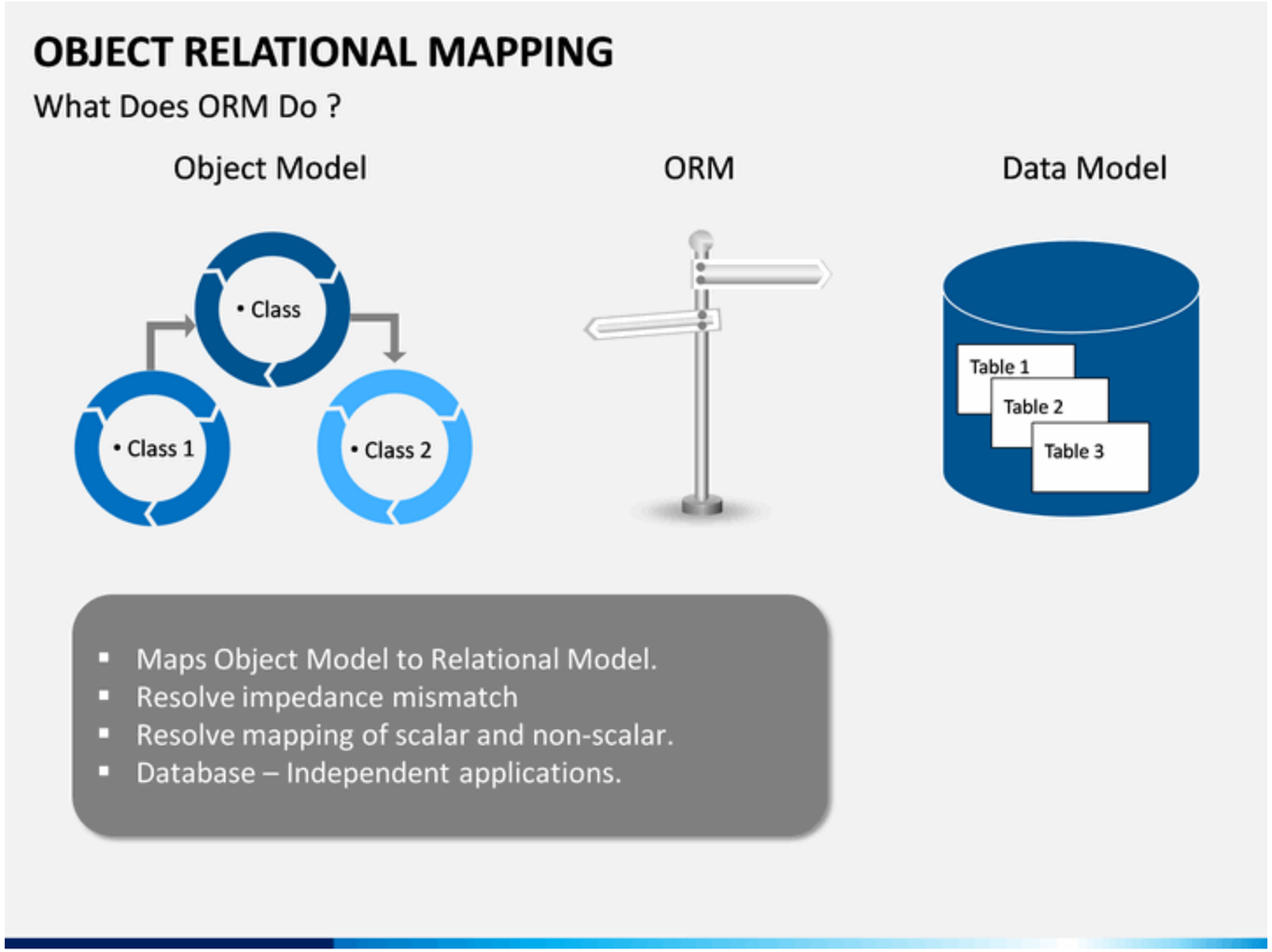




Busy Coder Academy

What is Object Relational Mapping ORM?

Object Relational Mapping (ORM) is a technique used in creating a "bridge" between object-oriented programs and relational databases.



Busy Coder Academy

What is Hibernate Framework?

Hibernate ORM is an object–relational mapping tool for the Java programming language.

It provides a framework for mapping an object-oriented domain model to a relational database

Busy Coder Academy

What is JPA?

Jakarta Persistence API is a Jakarta EE (Earlier called J2EE) application programming interface specification that describes the management of relational data in enterprise Java applications

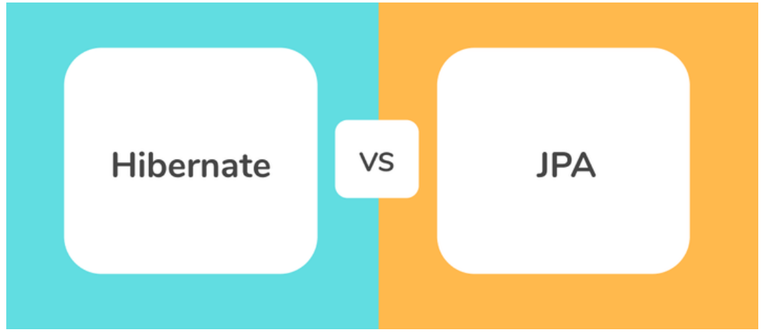
JSR

Busy Coder Academy

JPA vs Hibernate

The major difference between Hibernate and JPA is that Hibernate is a framework while JPA is API specifications

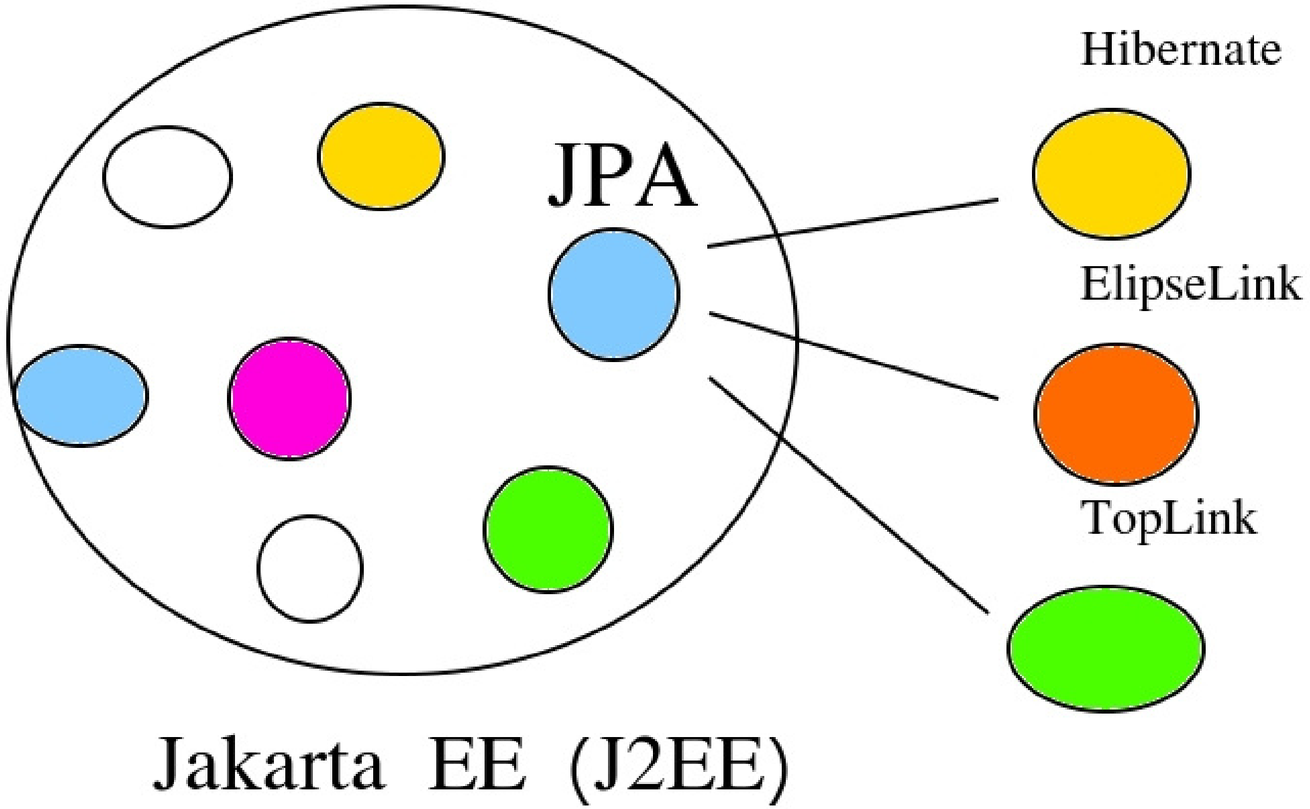
- Hibernate is an implementation of JPA guidelines.
- It helps in mapping Java data types to SQL data types.
- It is the contributor of JPA.



Busy Coder Academy

JPA vs Hibernate

Advantage of JPA is that You can swap ORM in your project without changing the code

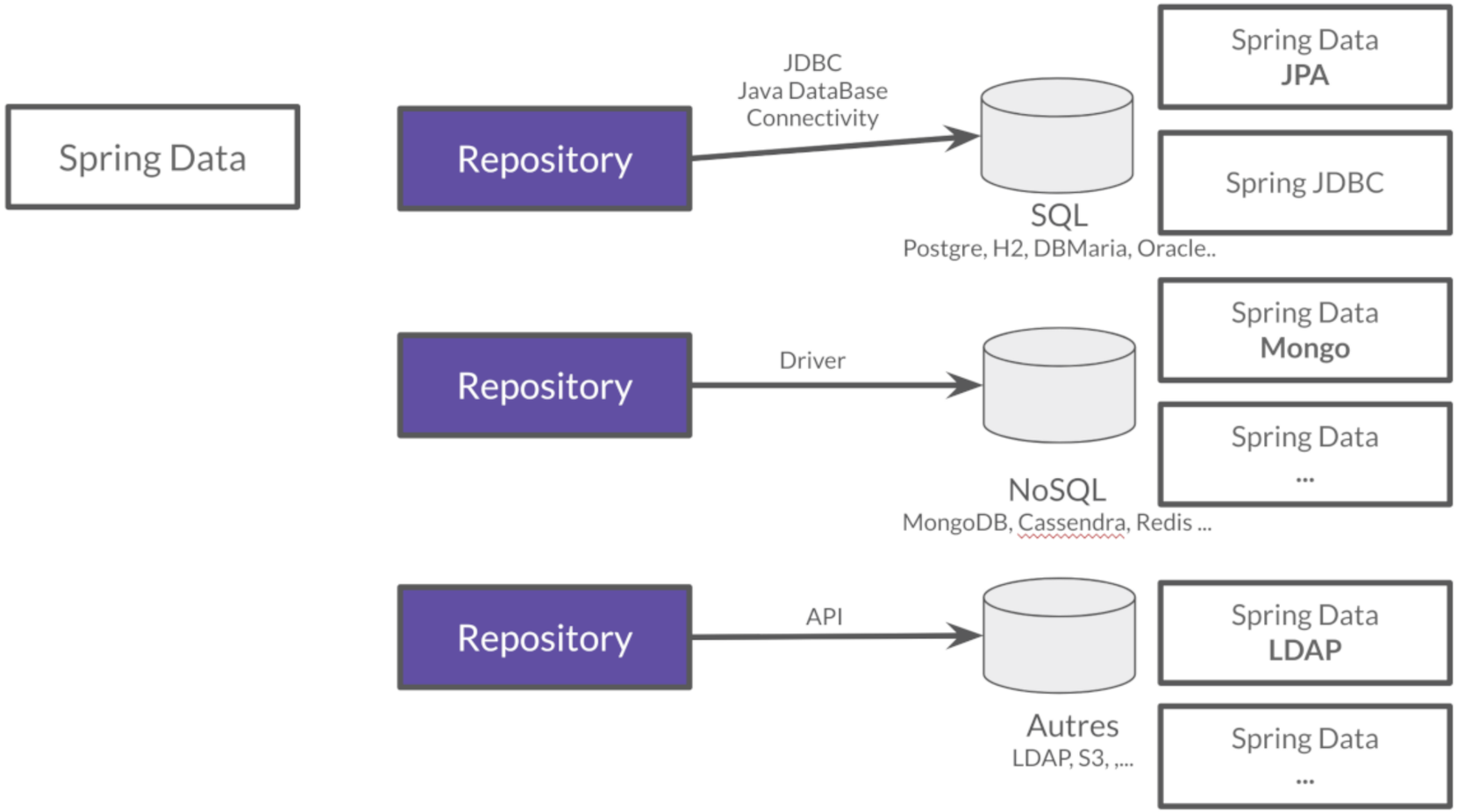


Busy Coder Academy

JPA vs Hibernate

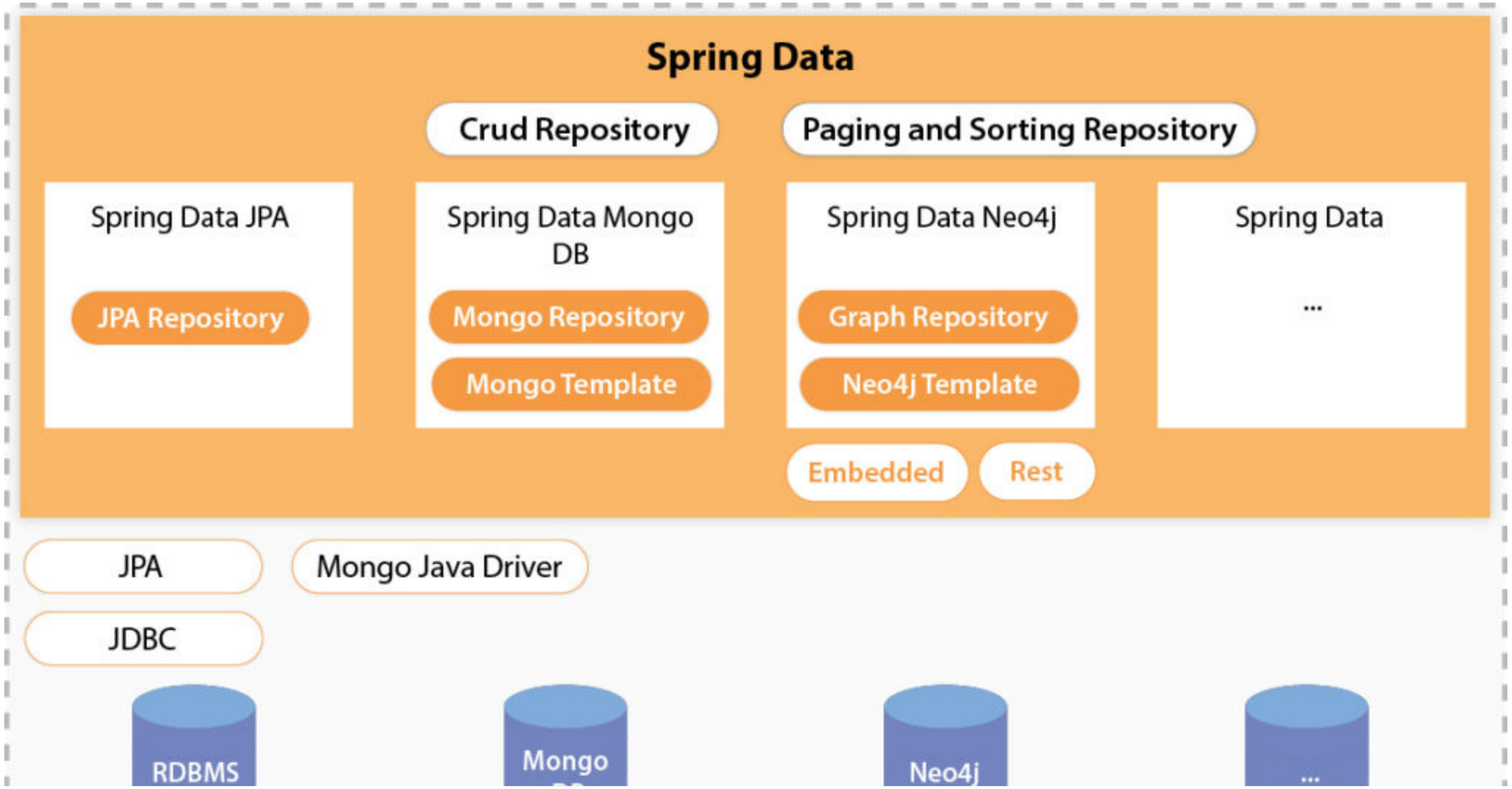
JPA	Hibernate
JPA is described in **javax.persistence** package.	Hibernate is described in **org.hibernate** package.
It describes the handling of relational data in Java applications.	Hibernate is an Object-Relational Mapping (ORM) tool that is used to save the Java objects in the relational database system.
It is not an implementation. It is only a Java specification.	Hibernate is an implementation of JPA. Hence, the common standard which **is** given by JPA is followed by Hibernate.
It is a standard API that permits to perform database operations.	It is used in mapping Java data types with SQL data types and database tables.
As an object-oriented query language, it uses **Java Persistence Query Language (JPQL)** to execute database operations.	As an object-oriented query language, it uses **Hibernate Query Language (HQL)** to execute database operations.
To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager.	To create Session instances, it uses **SessionFactory** interface.
To make, read, and remove actions for instances of mapped entity classes, it uses **EntityManager** interface. This interface interconnects with the persistence condition.	To make, read, and remove actions for instances of mapped entity classes, it uses **Session** interface. It acts as a runtime interface between a Java application and Hibernate.

What is Spring Data?



The Spring-Data is an umbrella project having many sub-projects or modules to provide uniform abstractions and uniform utility methods for the Data Access Layer in an application and support a wide range of databases and datastores.

What is Spring Data?



By using Spring data We dont have to write Dao layer, we just need to declare Dao layer and it provide uniform interface to interact with polyglot of databases

Spring support Dependency Injection

Keyword	Sample	JPQL snippet
And	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
Or	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
Is,Equals	`findByFirstname`,`findByFirstnameIs`,`findByFirstnameEquals`	`… where x.firstname = ?1`
Between	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`
LessThan	`findByAgeLessThan`	`… where x.age < ?1`
LessThanEqual	`findByAgeLessThanEqual`	`… where x.age <= ?1`
GreaterThan	`findByAgeGreaterThan`	`… where x.age > ?1`
GreaterThanEqual	`findByAgeGreaterThanEqual`	`… where x.age >= ?1`
After	`findByStartDateAfter`	`… where x.startDate > ?1`
Before	`findByStartDateBefore`	`… where x.startDate < ?1`

rgupta.mtech@gmail.com

Spring
support Dependency Injection

NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

rgupta.mtech@gmail.com

Creating DAO, DTO

```java
@Entity
@Table(name = "product_table")
public class Product {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private BigDecimal price;
    private LocalDate mfgDate;
    private String category;
```

```java
@Repository
public interface ProductDao extends JpaRepository<Product, Integer>{
    public Product findByName(String name);
}
```

```
server.servlet.context-path=/productapp
server.port=8082
spring.datasource.url=jdbc:mysql://localhost:3306/demo_boot?useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
logging.level.org.springframework.web: DEBUG
logging.level.org.hibernate: ERROR
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

rgupta.mtech@gmail.com

Derived quaries Examples

```java
@Repository
public interface ProductDao extends JpaRepository<Product, Integer>{
    public List<Product>findByNameLike(String title);
    public List<Product>findByNameContaining(String title);
    public List<Product>findByNameStartingWith(String title);
    public List<Product>findByNameEndingWith(String title);
    public List<Product>findByNameIgnoreCase(String title);
    public Product findByName(String name);
}
```

```java
@Repository
public interface ProductDao extends JpaRepository<Product, Integer>{

    public List<Product>findByMfgDateAfter(LocalDate mfgDate);
    public List<Product>findByMfgDateBefore(LocalDate mfgDate);
    public List<Product>findByMfgDateBetween(LocalDate mfgDate1, LocalDate mfgDate2);

}
```

rgupta.mtech@gmail.com

@Query annotation examples

```
@Repository
public interface ProductDao extends JpaRepository<Product, Integer>{

    @Query("select p from Product p")
    public List<Product>getListOfProducts();


    @Query("select p from Product p where p.category=?1")
    public List<Product>getListOfProductsByCategory(String category);

    @Query("select p from Product p where p.category=:category")
    public List<Product>getListOfProductsByCategoryParam(@Param("category") String category);

}
```

rgupta.mtech@gmail.com

Spring boot bootstrapping if Bootstrap class in other package

```java
@SpringBootApplication
@ComponentScan({"com.productapp"})
@EntityScan({"com.productapp.entities"})
@EnableJpaRepositories({"com.productapp.repo"})
public class Application {


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

}
```

rgupta.mtech@gmail.com

Productstore Service layer

```
public interface ProductService {
    public List<Product> findAll();
    public Product getById(int id);
    public Product addProduct(Product product);
    public Product updateProduct(int id, Product product);
    public Product deleteProduct(int id);

}
```

rgupta.mtech@gmail.com

Productstore Service layer

```java
@Service
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    @Autowired
    public ProductServiceImpl(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Override
    public List<Product> findAll() {
        return productDao.findAll();
    }

    @Override
    public Product getById(int id) {
        return productDao.findById(id)
                .orElseThrow(() -> new ProductNotFoundException("product with id" + id + " is not found"));
    }

    @Override
    public Product addProduct(Product product) {
        productDao.save(product);
        return product;
    }

    @Override
    public Product updateProduct(int id, Product product) {
        Product productToUpdate = getById(id);
        productToUpdate.setPrice(product.getPrice());
        productDao.save(productToUpdate);
        return productToUpdate;
    }

    @Override
    public Product deleteProduct(int id) {
        Product productToDelete = getById(id);
        productDao.delete(productToDelete);
        return productToDelete;
    }

}
```

rgupta.mtech@gmail.com

Rest controller

```java
@RestController
public class ProductController {

    private ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }
    @GetMapping(path = "products")
    public List<Product>findAll(){
        return productService.findAll();
    }

    @GetMapping(path = "products/{id}")
    public Product findById(@PathVariable(name = "id") int id){
        return productService.getById(id);
    }

    @PostMapping(path = "products")
    public Product addProduct( @RequestBody Product product){
        return productService.addProduct(product);
    }

    @DeleteMapping(path = "products/{id}")
    public Product deleteProduct(@PathVariable(name = "id") int id){
        return productService.deleteProduct(id);
    }

    @PutMapping(path = "products/{id}")
    public Product updateProduct(@PathVariable(name = "id") int id, @RequestBody Product product){
        return productService.updateProduct(id, product);
    }
}
```

rgupta.mtech@gmail.com

Spring Boot Pagination

Pagination is important techenique for large result set to displayed to a web page
Breaking down larger data set to sub set into logical pages. Spring provide paging and sorting repo out of the box
it contain several method to support pagination

rgupta.mtech@gmail.com

Pagination

```
@Bean
public CommandLineRunner commandLineRunner(){
    return  args-> {

        System.out.println("hello to all");
        List<Product> products= IntStream.rangeClosed(1,1000)
                .mapToObj(i-> new Product( name: "product "+i,
                        category: "EL", new BigDecimal(new Random().nextInt( bound: 1000)))).toList();

        productRepo.saveAll(products);
    };
}
```

```
@GetMapping(path = "productssorted")
public List<Product> getAllSorted(@RequestParam(name = "fieldName") String fieldName){

    return productRepo.findAll(Sort.by(Sort.Direction.ASC, fieldName));
}
@GetMapping(path = "productspage")
public Page<Product> getAllPagination(@RequestParam(name = "offset") Integer offset,
                                      @RequestParam(name = "pageSize") Integer pageSize){
    return productRepo.findAll(PageRequest.of(offset, pageSize));
}
@GetMapping(path = "productspagesorted")
public Page<Product> getAllSortedAndPagination(@RequestParam(name = "fieldName") String fieldName,
                                               @RequestParam(name = "offset") Integer offset,
                                               @RequestParam(name = "pageSize") Integer pageSi
    return productRepo.findAll(PageRequest.of(offset, pageSize).withSort(Sort.by(fieldName)));
}
```

rgupta.mtech@gmail.com

Module 7: Spring REST Advance

rgupta.mtech@gmail.com

Agenda

RESTful Application with Spring Boot Advance

- REST with HTTP Status coe
- REST Exceptioin Handling
- REST validation using JSR 303
- Implementation of custom valiation logic
- REST Hatoas
- XML support
- Spring boot logging customization
- Spring Boot Swagger Documentation using OpenAPI 3.0
- Spring boot caching Supports
- Spring boot schedule processes
- Spring Boot packaging options, JAR or WAR

Actuators, Metrics and Health Indicators

- Exposing Spring Boot Actuator endpoints
- Custom Metrics
- Health Indicators
- Creating custom Health Indicators
- Extemal monitoring systems

Spring boot 3.0 Enhancements

- Observability
- Problem details
- Jakarta, Dependency Upgrade
- Spring native
- Httpexchange

rgupta.mtech@gmail.com

Handling HttpStatus code

@RestController

// @RestController=@Controller + @ResponseBody

HTTP Status Codes

Code	Description	Code	Description
200	OK	400	Bad Request
201	Created	401	Unauthorized
202	Accepted	403	Forbidden
301	Moved Permanently	404	Not Found
303	See Other	410	Gone
304	Not Modified	500	Internal Server Error
307	Temporary Redirect	503	Service Unavailable

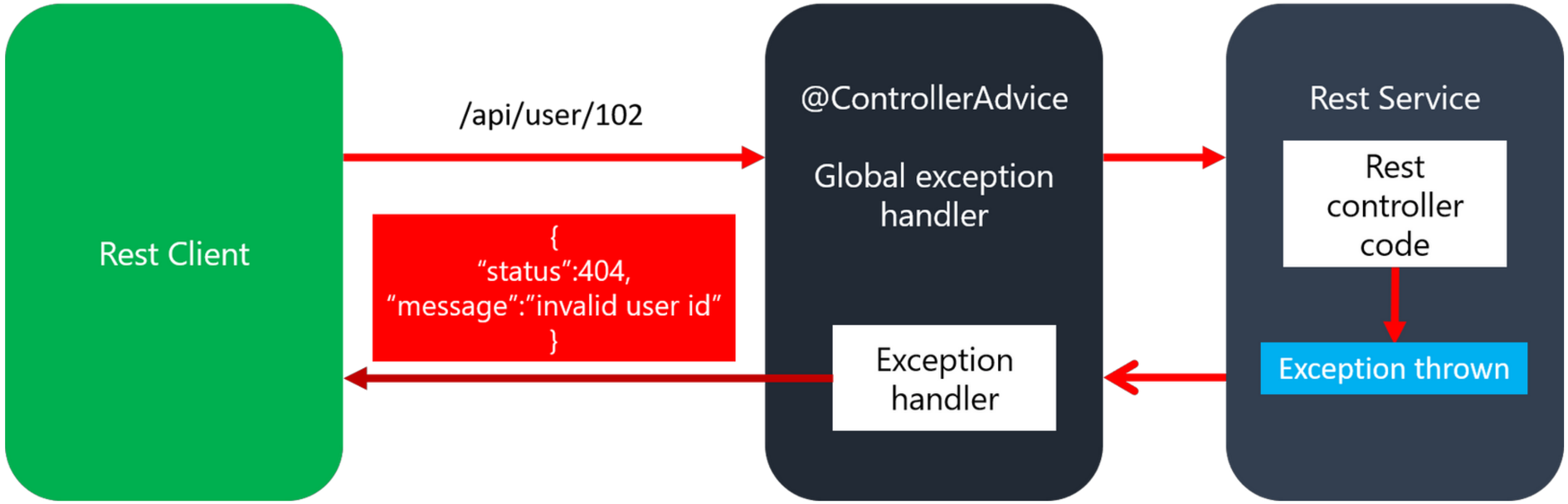
rgupta.mtech@gmail.com

REST with HttpStatus code

```java
@RestController
public class ProductController {

    private ProductService productService;

    @Autowired
    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping(path = "products")
    public ResponseEntity<List<Product>> findAll(){
        return ResponseEntity.status(HttpStatus.OK).body(productService.findAll());
    }

    @GetMapping(path = "products/{id}")
    public ResponseEntity<Product> findById(@PathVariable(name = "id") int id){
        return ResponseEntity.ok(productService.getById(id));
    }

    @PostMapping(path = "products")
    public ResponseEntity<Product> addProduct( @RequestBody Product product){
        return ResponseEntity.status(HttpStatus.CREATED).body(productService.addProduct(product));
    }

    @DeleteMapping(path = "products/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable(name = "id") int id){
         productService.deleteProduct(id);
         return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    }

    @PutMapping(path = "products/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable(name = "id") int id, @RequestBody Product product){
        return ResponseEntity.status(HttpStatus.CREATED).body(productService.updateProduct(id, product));
    }
}
```

rgupta.mtech@gmail.com

Spring REST Exception Handling



rgupta.mtech@gmail.com

Spring REST Exception Handling

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class ErrorDetails {
    private String message;
    private String details;

    private String name;
    private Date date;

}
```

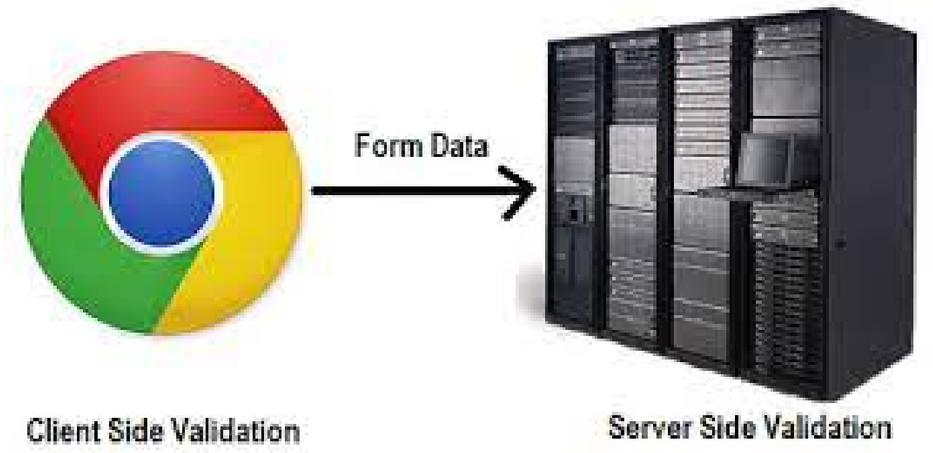
```java
@RestControllerAdvice
public class ExHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorDetails> handle404(Exception ex, WebRequest req){
        ErrorDetails details=new ErrorDetails();
        details.setDate(new Date());
        details.setDetails(req.getDescription(true));
        details.setName("rgupta.mtech@gmail.com");
        details.setDetails(ex.toString());
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(details);
    }


    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorDetails> handle500(Exception ex, WebRequest req){
        ErrorDetails details=new ErrorDetails();
        details.setDate(new Date());
        details.setDetails(req.getDescription(true));
        details.setName("rgupta.mtech@gmail.com");
        details.setDetails(ex.toString());
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(details);
    }
}
```

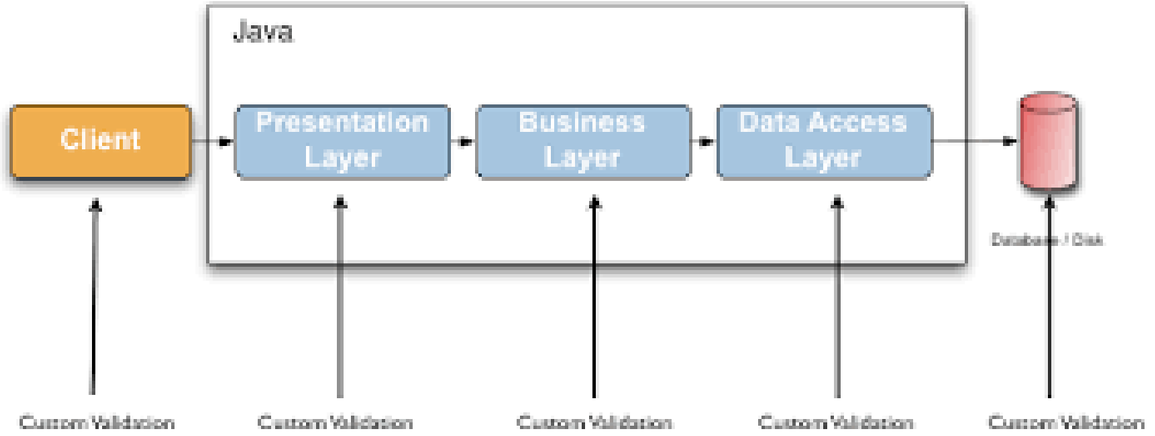
rgupta.mtech@gmail.com

JSR 303validation

JSR-303 bean validation is an specification whose objective is to standardize the validation of Java beans through annotations.

- @ NotNull - javax.validation.constraints
- @ Pattern - javax.validation.constraints
- @ AssertFalse - javax.validation.constraints
- @ AssertTrue - javax.validation.constraints
- @ DecimalMax - javax.validation.constraints
- @ DecimalMin - javax.validation.constraints
- @ Digits - javax.validation.constraints
- @ Future - javax.validation.constraints
- @ Max - javax.validation.constraints
- @ Min - javax.validation.constraints
- @ Null - javax.validation.constraints
- @ Past - javax.validation.constraints
- @ Size - javax.validation.constraints





rgupta.mtech@gmail.com

JSR 303validation

Step 1: Apply JSR 303 annoation on POJO

```java
@NotEmpty(message = "name should be left black")
private String name;

@NotNull(message = "product.price.absent")
@Range(min = 100, max = 100000, message = "product.price.invalid")
private BigDecimal price;
```

Step 2: Force exception in case valiation is failed

```java
@PostMapping(path = "products")
public ResponseEntity<Product> addProduct(@Valid @RequestBody  Product product ){
    return  ResponseEntity.status(HttpStatus.CREATED).body(productService.addProduct(product));
}
```

Step 3: Create Handler to Handle 400 error

```java
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ErrorInfo> handle400(MethodArgumentNotValidException ex){

    String errorMessage=ex.getBindingResult() BindingResult
            .getAllErrors() List<ObjectError>
            .stream() Stream<ObjectError>
            .map(x->x.getDefaultMessage()) Stream<String>
            .collect(Collectors.joining( delimiter: " , "));

    ErrorInfo errorInfo=ErrorInfo.builder()
            .errorMessage(errorMessage)
            .timestamp(LocalDateTime.now())
            .statusCode(HttpStatus.BAD_REQUEST.toString())
            .build();
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorInfo);
}
```

rgupta.mtech@gmail.com

Implementation of custom valiation logic

We need to ensure that product category should be Electronics or Books?

No Readymade annoation form
JSR 303

rgupta.mtech@gmail.com

Implementation of custom valiation logic

Step 1: Create custom annotation

```java
@Documented
@Constraint(validatedBy = { ProductValidator.class})
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER, TYPE_USE })
@Retention(RUNTIME)
public @interface ValidateProductType {
    String message() default "Invalid product type :EL or BOOK FMCG";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };
}
```

Step 3: Apply Custom Anntation

```java
//BOOK, EL,FMCG
@ValidateProductType
private String category;
```

Step 1: Create ConstraintValidator

```java
@Component
public class ProductValidator implements ConstraintValidator<ValidateProductType, String> {
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        //logic
        List<String> productType= Arrays.asList("EL","BOOK");
        return productType.contains(value);
    }
}
```

rgupta.mtech@gmail.com

Supporting both xml and json

```
@GetMapping(path = "products", produces = {MediaType.APPLICATION_JSON_VALUE,
        MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<List<Product>> findAll(){
    return ResponseEntity.status(HttpStatus.OK).body(productService.findAll());
}
```

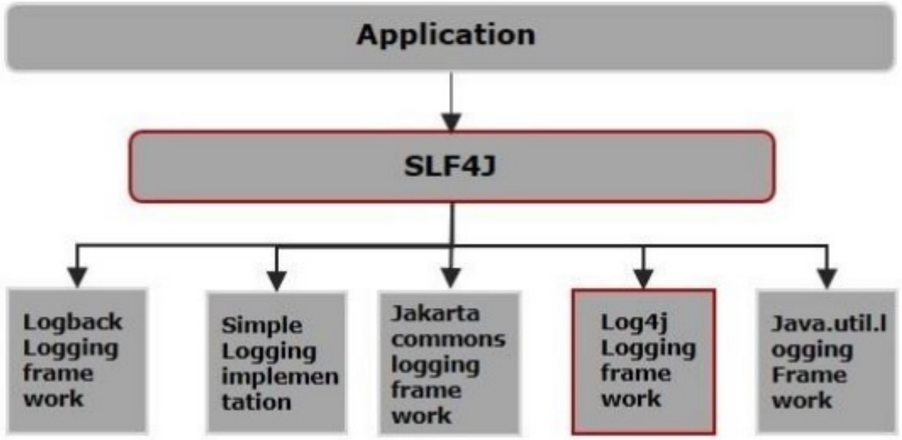
```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

rgupta.mtech@gmail.com

Spring boot logging customization

Spring boot
with log4j2?


```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.2.RELEASE)

2017-03-23 18:11:30.706  INFO 43614 --- [           main] c.q.springboot.LogdemoApplication
k/projects/sts-spring-boot/logdemo/build/classes/main started by jj in /Volumes/JJPlay/quick/p
2017-03-23 18:11:30.711 DEBUG 43614 --- [           main] c.q.springboot.LogdemoApplication
2017-03-23 18:11:30.712  INFO 43614 --- [           main] c.q.springboot.LogdemoApplication
2017-03-23 18:11:33.098  INFO 43614 --- [           main] c.q.springboot.LogdemoApplication
2017-03-23 18:11:50.307 ERROR 43614 --- [nio-8080-exec-1] c.q.springboot.HelloWorldController
2017-03-23 18:11:53.008  WARN 43614 --- [nio-8080-exec-2] c.q.springboot.HelloWorldController
2017-03-23 18:11:53.011  WARN 43614 --- [nio-8080-exec-3] c.q.springboot.HelloWorldController
```



rgupta.mtech@gmail.com

Using OpenAPI 3.0

Documenting a Spring REST API Using OpenAPI 3.0
Swagger is almost equivalent to SOAP formate, used for documentation of REST api

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.4</version>
</dependency>
```

```
@SpringBootApplication
@OpenAPIDefinition(info = @Info(title = "productstore app API", version = "2.0" ,
   description = "Champs productstore API"))
public class Application implements CommandLineRunner {
```

rgupta.mtech@gmail.com

Implementing Enable cacheing

Ehcache is an open source Java distributed cache for general-purpose caching, Java EE and light-weight containers. Ehcache is available under an Apache open source license

For caching declaration, the abstraction provides a set of Java annotations:

- `@Cacheable` triggers cache population
- `@CacheEvict` triggers cache eviction
- `@CachePut` updates the cache without interfering with the method execution
- `@Caching` regroups multiple cache operations to be applied on a method
- `@CacheConfig` shares some common cache-related settings at class-level

rgupta.mtech@gmail.com

Implementing Enable Scheduled process

Scheduling is a process of executing the tasks for the specific time period. Spring Boot provides a good support to write a scheduler on the Spring applications

```java
@Service
public class ScheduledJob {
    private Logger logger = LoggerFactory.getLogger(ScheduledJob.class);
    @Autowired
    private ProductService service;

    @Scheduled(cron = "0,30 * * * * *")
    public void cronJob() {
        List<Product> products = service.findAll();
        logger.info("There are {} products in the data store.", products.size());
    }
    // after application startup delay of 5 sec, schedule to run each after 15
    @Scheduled(initialDelay = 5000, fixedRate = 15000)
    public void fixedRateJob() {
        logger.info("> fixedRateJob");

        // Add scheduled logic here

        List<Product> products = service.findAll();

        logger.info("There are {} books in the data store.", products.size());

        logger.info("< fixedRateJob");
    }
}
```

rgupta.mtech@gmail.com

Spring Boot packaging options, JAR or WAR

In Spring boot applications, the default packaging is jar which the application is deployed in embedded servers.

If you want to generate a war file for deployment in separate application server instances such as Jboss, Weblogic or Tomcat, follow the instructions below.

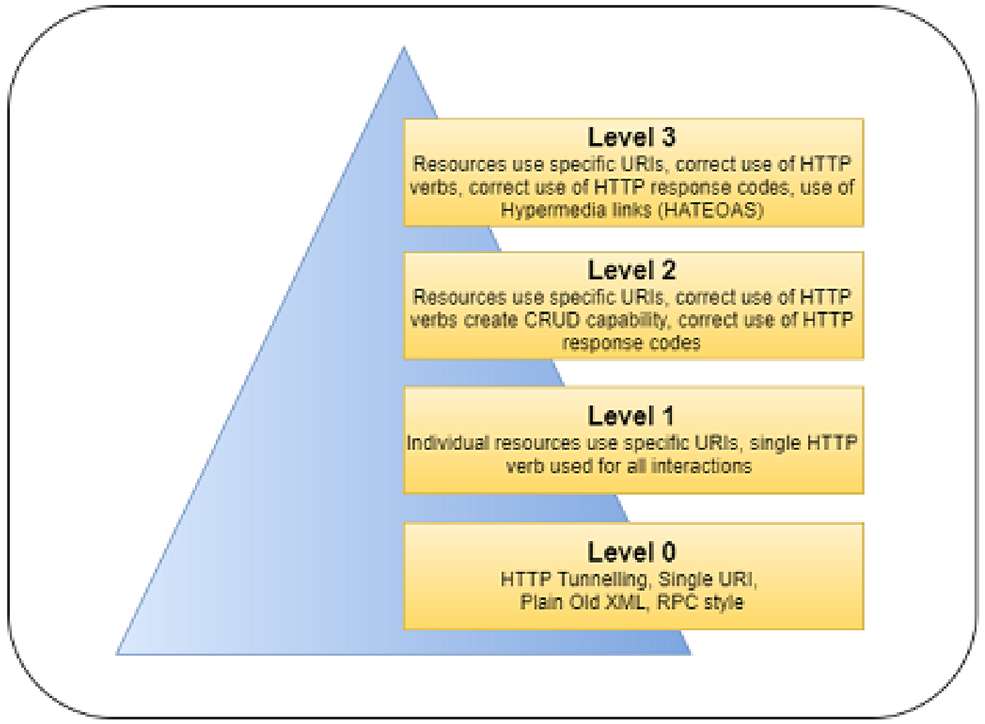
```
<packaging>war</packaging>
```

```
@SpringBootApplication
public class WarInitializerApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        return builder.sources(WarInitializerApplication.class);
    }

}
```

rgupta.mtech@gmail.com

HATEOAS

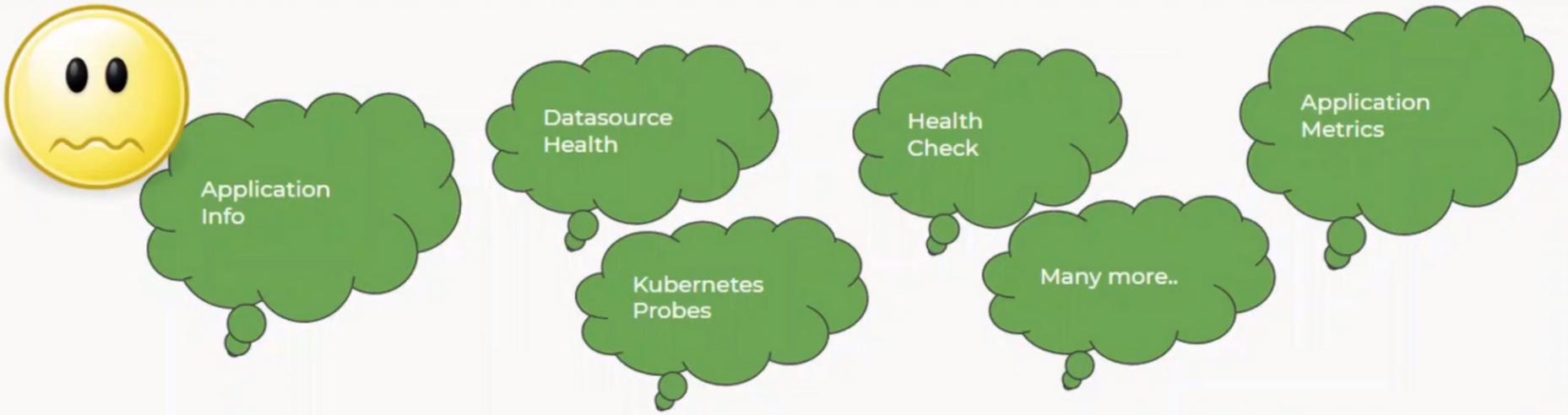
Hypermedia as the Engine of Application State (HATEOAS) is a constraint of the REST application architecture that distinguishes it from other network application architectures.
With HATEOAS, a client interacts with a network application whose application servers provide information dynamically through hypermedia




```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.account+json
Content-Length: ...

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": 100.00
        },
        "links": {
            "deposit": "/accounts/12345/deposit",
            "withdraw": "/accounts/12345/withdraw",
            "transfer": "/accounts/12345/transfer",
            "close": "/accounts/12345/close"
        }
    }
}
```

rgupta.mtech@gmail.com

What is Spring boot Actuators?



Spring Boot provides an excellent support for production ready features just by adding single Actuator dependency.

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

Actuators, Metrics and Health Indicators

Spring Boot Actuator is a sub-project of the Spring Boot Framework. It includes a number of additional features that help us to monitor and manage the Spring Boot application. It contains the actuator endpoints (the place where the resources live).

Actuator?

actuator

noun

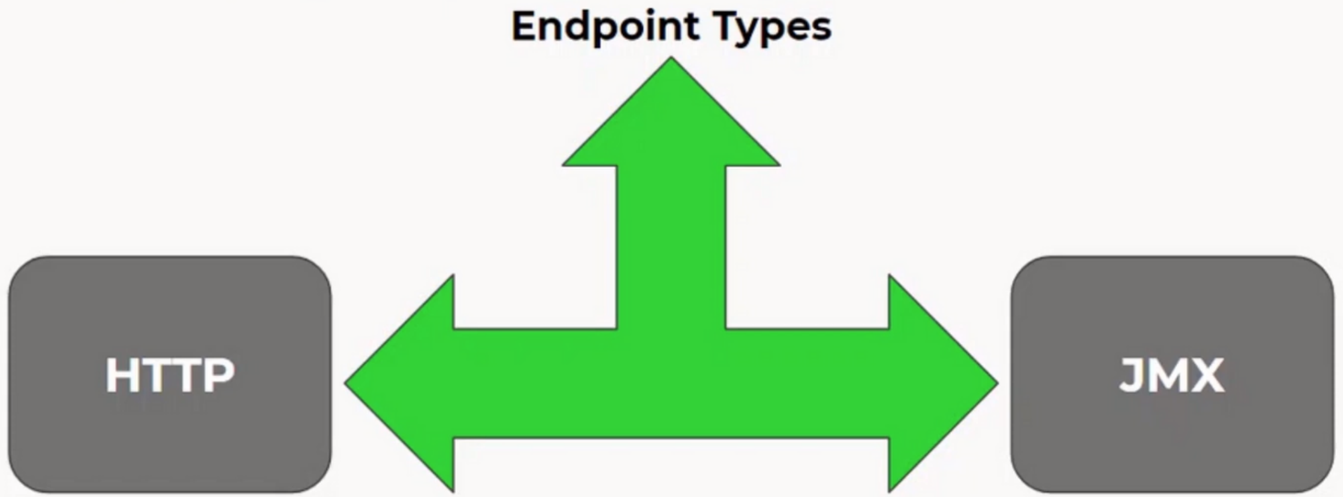
module for controlling Spring Boot applications

mechanical device for moving or controlling something

Spring Boot Actuator

Actuators Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own.



Actuators Endpoints

Actuator Endpoints contd.

ID	JMX	Web
auditevents	Yes	No
beans	Yes	No
caches	Yes	No
conditions	Yes	No
configprops	Yes	No
env	Yes	No
flyway	Yes	No
health	Yes	Yes
sessions	Yes	No
shutdown	Yes	No
startup	Yes	No

ID	JMX	Web
heapdump	N/A	No
httptrace	Yes	No
info	Yes	Yes
integrationgraph	Yes	No
jolokia	N/A	No
logfile	N/A	No
loggers	Yes	No
liquibase	Yes	No
metrics	Yes	No
mappings	Yes	No
prometheus	N/A	No

Endpoints Behavior

To change which endpoints are exposed, use the following technology-specific include and exclude properties:

Property	Default Value
management.endpoints.jmx.exposure.exclude	
management.endpoints.jmx.exposure.include	*
management.endpoints.web.exposure.exclude	
management.endpoints.web.exposure.include	info, health

Enabling/Disabling Endpoints

By default, all endpoints except for shutdown are enabled. To configure the enablement of an endpoint, use its **management.endpoint.<id>.enabled** property.

```
management.endpoint.shutdown.enabled=true
```

If you prefer endpoint enablement to be opt-in rather than opt-out, set the **management.endpoints.enabled-by-default** property to **false** and use individual endpoint **enabled** properties to opt back in. The following example enables the info endpoint and disables all other endpoints:

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

Custom Endpoints

We can add custom endpoints to actuator as well. For this, we need to add @Endpoint (with id as argument) at class level. Inside the class we can use annotation like:

@ReadOpeation for GET methods,
@WriteOperation for POST, and
@DeleteOperation for DELETE.

```
@Configuration
@Endpoint(id = "custom-endpoint")
public class CustomEndpoints {
    @ReadOperation
    public String getCustomData() { return "This is custom Data"; }
}
```

CORS Support

CORS support is disabled by default and is only enabled once the **management.endpoints.web.cors.allowed-origins** property has been set. The following configuration permits GET and POST calls from the example.com domain

```
management.endpoints.web.cors.allowed-origins=https://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

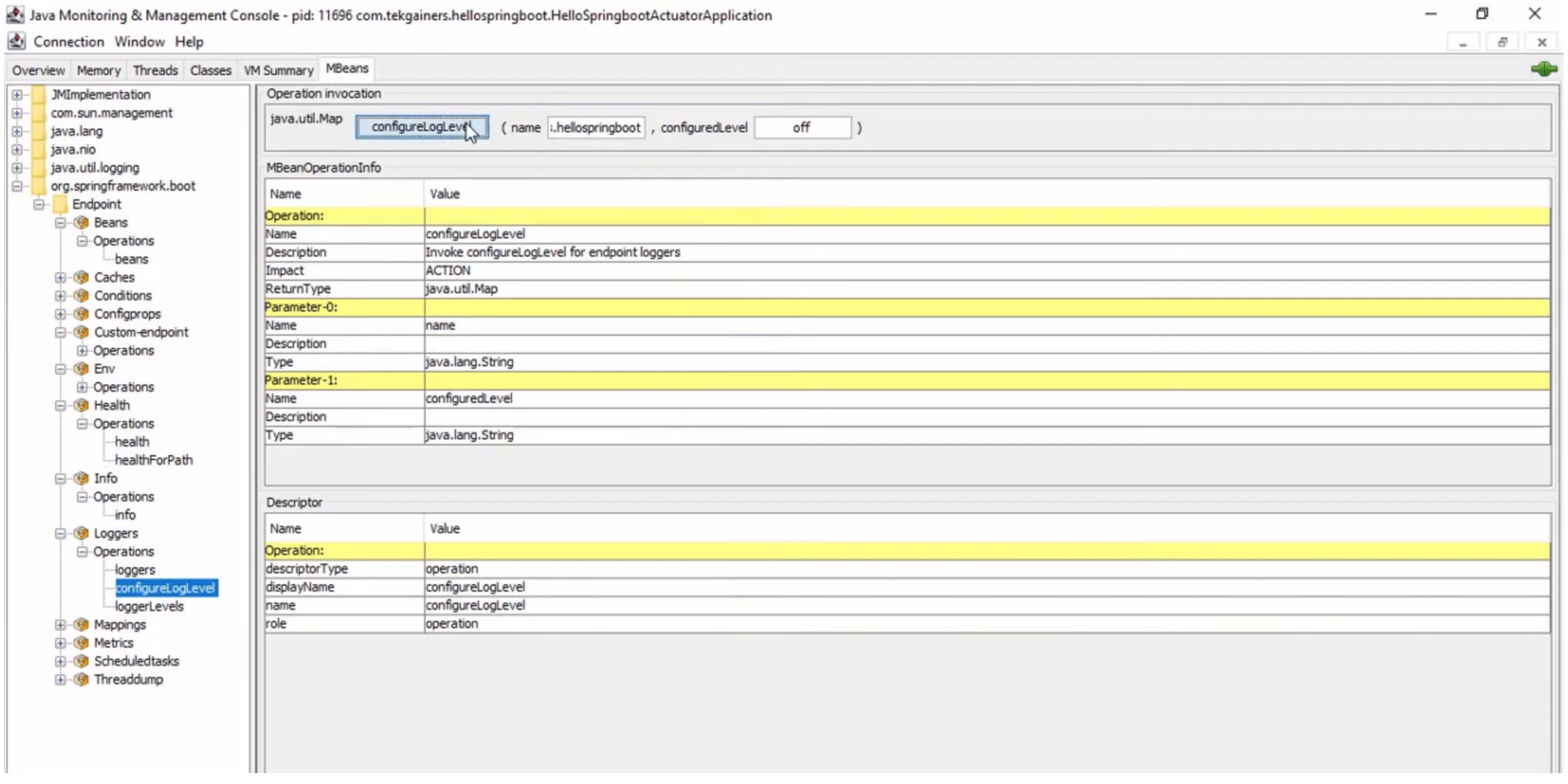
Actuators: Monitoring and Management Over JMX

By default, JMX endpoints are not enabled. It can be turned on by setting the following configuration property.

```
spring.jmx.enabled = true
```

Spring Boot exposes management endpoints as JMX MBeans under the org.springframework.boot domain by default.

Actuators: Monitoring and Management Over JMX



Actuators Custom EndPoints

```java
@Configuration
@Endpoint(id = "custom-endpoint")
public class CustomEndpoints {
    @ReadOperation
    public String getCustomData(){
        return "This is custom Data";
    }
}
```

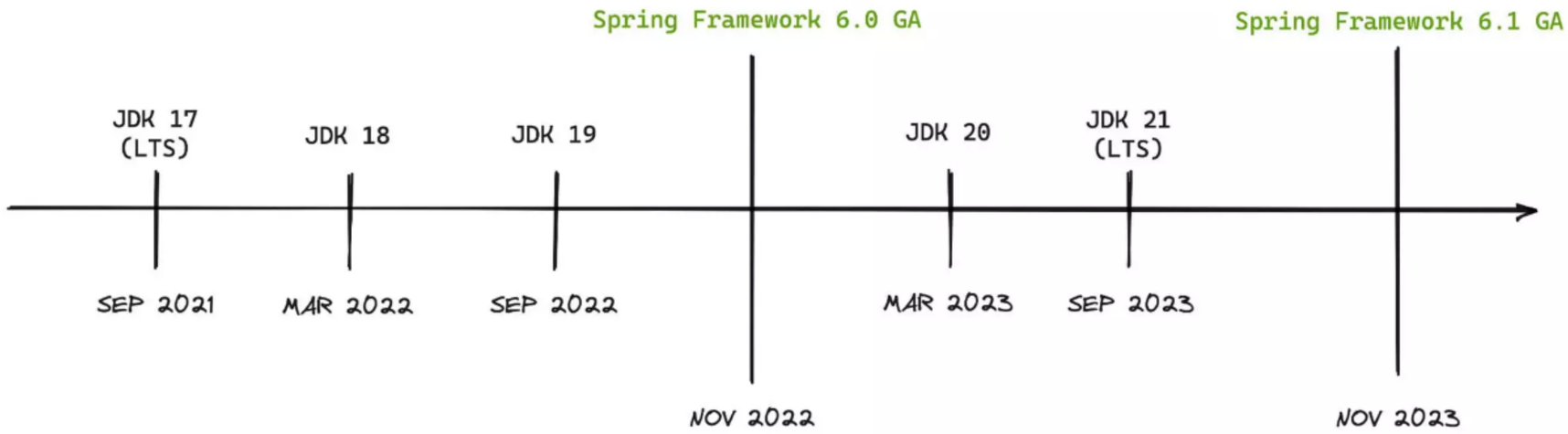
Spring boot 3.0 Enhancements

Spring Boot 3.0 was released on November 2022. Here is a consolidated list of changes introduced/updated to the framework.

rgupta.mtech@gmail.com

Spring boot 3.0 Enhancements

Spring Boot 3.0 was released on November 2022. Here is a consolidated list of changes introduced/updated to the framework.



rgupta.mtech@gmail.com

Spring boot 3.0 Enhancements

- On Nov 24, 2022, Spring announced that Spring 3.0.0 is now generally available
- This was the first major revision of Spring boot since 2.0 was released about 5 years ago.
- There's far too many features to list them all here, so we are listing the highlights of this release

rgupta.mtech@gmail.com

Spring 3 : Java 17 Baseline

- On Nov 24, 2022, Spring announced that Spring 3.0.0 is now generally available
- This was the first major revision of Spring boot since 2.0 was released about 5 years ago.
- There's far too many features to list them all here, so we are listing the highlights of this release

Releases follow a major.minor.patch version scheme

Some versions contains M, RC and Snapshot what it means?

Snapshot: means under development. Snapshot might update. ie. downloading 1.0-SNAPSHOT today might give a different jar file then downloaded yesterday.

M indicate a milestone release: probably not feature complete, should be vaguely stable (i.e. it is more then just a nightly snapshot) but may still have problems.

RC indicating a release candidate release: Probably feature complete and should be pretty stable- problems should be relatively rare and minor. Its is the build release internally to check if any critical problems have gone undetected into the code during the previous development period. Release candidates are not for production deployment, but for testing purpose only

GA indicates a General Availability (a release): should be very stable and feature complete

rgupta.mtech@gmail.com

Jakarta EE

Ever heard of Java EE? How about Java 2EE, J2EE, or now Jakarta EE? Actually, these are all different names for the same thing: a set of enterprise specifications that extend Java SE.

History

In the first version of Java, Java enterprise extensions were simply a part of the core JDK.

Then, as part of Java 2 in 1999, these extensions were broken out of the standard binaries, and J2EE, or Java 2 Platform Enterprise Edition, was born. It would keep that name until 2006.

For Java 5 in 2006, J2EE was renamed to Java EE or Java Platform Enterprise Edition. That name would stick all the way to September 2017, when something major happened.

See, in September 2017, Oracle decided to give away the rights for Java EE to the Eclipse Foundation (the language is still owned by Oracle).

Version	Date
J2EE 1.2	December 1999
J2EE 1.3	September 2001
J2EE 1.4	November 2003
Java EE 5	May 2006
Java EE 6	December 2009
Java EE 7	April 2013
Java EE 8	August 2017
Jakarta EE	February 2018*

Actually, the Eclipse Foundation legally had to rename Java EE. That's because Oracle has the rights over the “Java” brand.

So to choose the new name, the community voted and picked: Jakarta EE. In a certain way, it's still JEE.

rgupta.mtech@gmail.com

What is observability?

Observability is the ability to observe the internal state of a running system from the outside. In other words, “how well you can understand the internals of your system by examining its outputs”. It consists of the three pillars.

- **Logging**: A text record of an event that happened at a particular time and includes a timestamp that tells when it occurred and a payload that provides context.

- **Metrics**: It is a numeric value measured over an interval of time and includes specific attributes such as timestamp, name, KPIs and value.

- **Traces**: A trace represents the end-to-end journey of a request through a distributed system.

rgupta.mtech@gmail.com

Observability vs Monitoring

Observability

- Tells why a system is at fault
- Acts as a knowledge base in defining what to monitor
- Focuses on giving the context to the data
- It serves as traversable map
- It creates the potential to monitor different events

Monitoring

- Notifies that the system is at fault
- Focuses on monitoring the systems and discovering faults
- Focuses on the collection of data
- Monitoring is a single plane
- It is a process of using the observability

rgupta.mtech@gmail.com

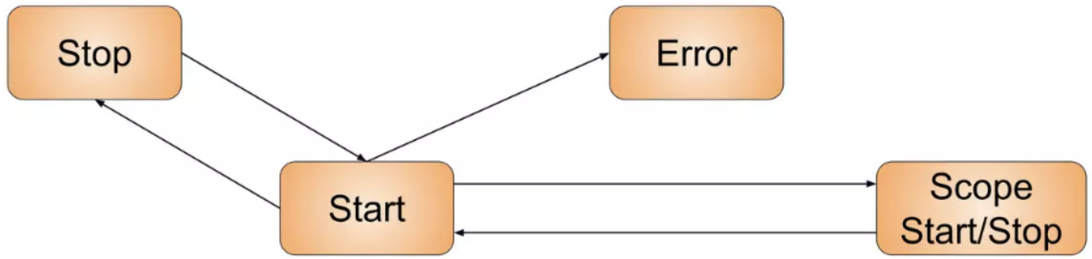
Micrometer Observation API?

Micrometer Observation API

- Observation
- ObservatioRegistry
- ObservationHandler

To create an observation, we need an ObservationRegistry. For any Observation to happen, you need to register ObservationHandler objects through an ObservationRegistry.

An ObservationHandler can create timers, spans, and logs by reacting to the lifecycle events of an Observation



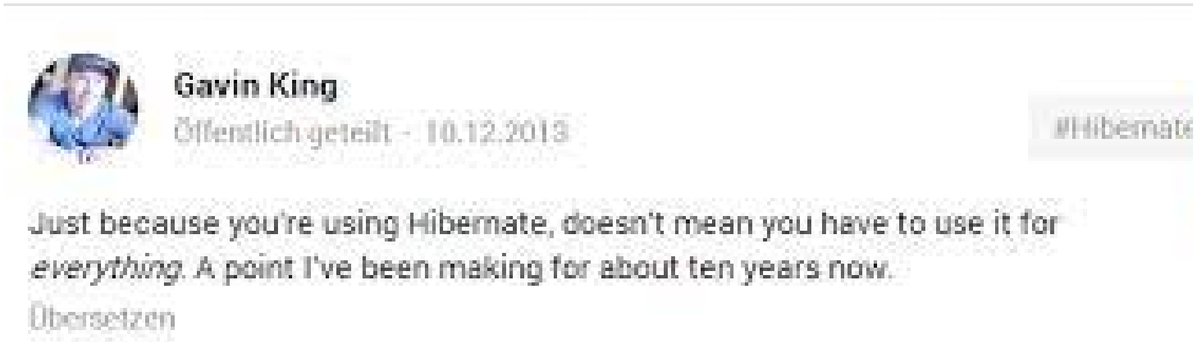
rgupta.mtech@gmail.com

Observability in Spring Boot 3

- Spring Boot 3.0 supports the new observation APIs introduced in Micrometer 1.10
- The new ObservationRegistry interface can be used to create observations which provides a single API for both metrics and traces
- Spring Boot now auto-configures an instance of ObservationRegistry for us
- Spring Boot now auto-configures Micrometer Tracing for you. This includes support for Brave, OpenTelemetry, Zipkin and Wavefront
- Spring Cloud Sleuth is replaced by the Micrometer Tracing Framework in the new version

rgupta.mtech@gmail.com

Module 8: Hibernate, Spring Data Joins



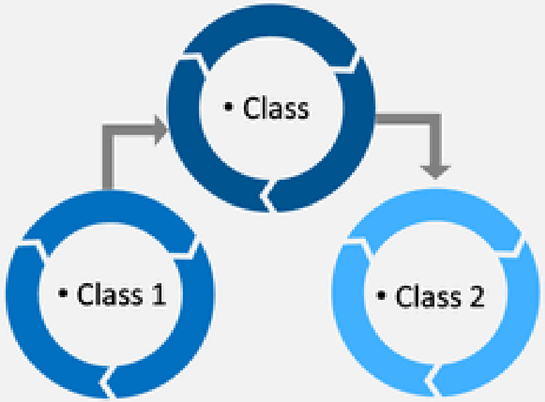
rgupta.mtech@gmail.com

Hibernate ORM

OBJECT RELATIONAL MAPPING

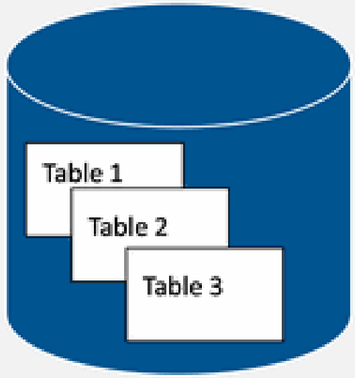
What Does ORM Do ?

Object Model



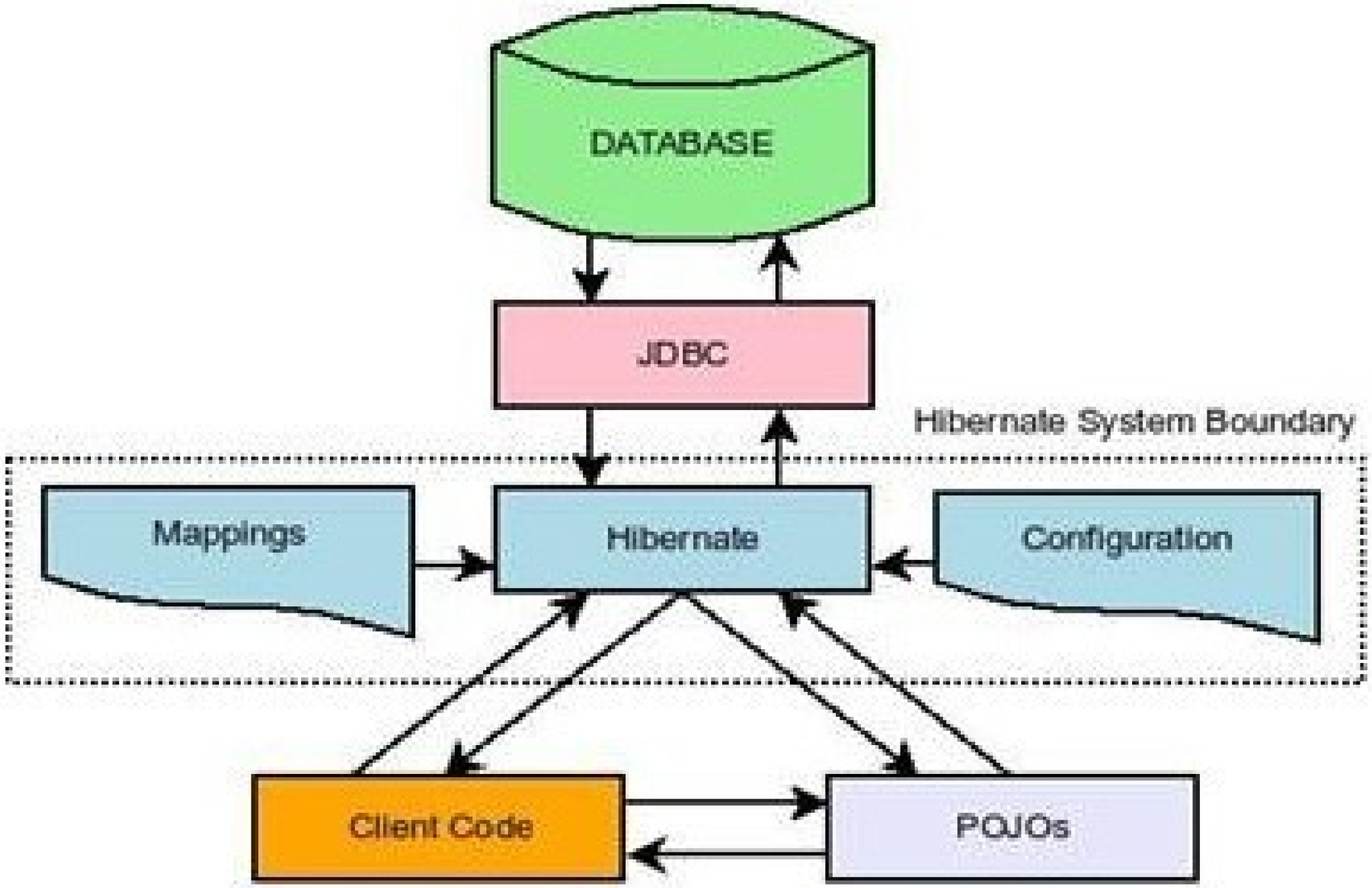
ORM

Data Model



- Maps Object Model to Relational Model.
- Resolve impedance mismatch
- Resolve mapping of scalar and non-scalar.
- Database – Independent applications.

Hibernate architecture



Relationship Mapping

Most of time we required to map relationship between entities
Every relationship has four characteristics:

Directionality: Unidirectional vs Bidirectional ?
Role: Each entity in the relationship is said to play a role. Depending on directionality, we can identity the entity playing the role of sources and entity playing the role of target
Cardinality: The number of entity instances that exists on each side of the relationship
Ownership: One of the two entities in the relationship is said to own the relationship. Employee is called owner of relationship and Department is called reverse-owner of relationship

Relationship Mapping

Most of time we required to map relationship between entities
Every relationship has four characteristics:

Directionality: Unidirectional vs Bidirectional ?
Role: Each entity in the relationship is said to play a role. Depending on directionality, we can identity the entity playing the role of sources and entity playing the role of target
Cardinality: The number of entity instances that exists on each side of the relationship
Ownership: One of the two entities in the relationship is said to own the relationship. Employee is called owner of relationship and Department is called reverse-owner of relationship

Relationship Mapping

Most of time we required to map relationship between entities
Every relationship has four characteristics:

Directionality: Unidirectional vs Bidirectional ?
Role: Each entity in the relationship is said to play a role. Depending on directionality, we can identity the entity playing the role of sources and entity playing the role of target
Cardinality: The number of entity instances that exists on each side of the relationship
Ownership: One of the two entities in the relationship is said to own the relationship. Employee is called owner of relationship and Department is called reverse-owner of relationship

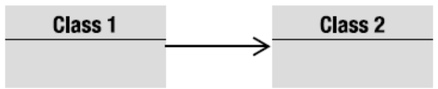
Relationship Mapping

OO relations

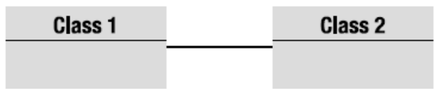
Association between the Objects

IS-A, HAS-A, USE-A

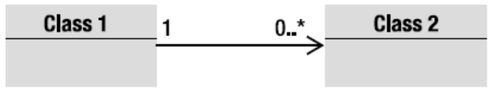
An association has a direction:



Unidirectional



Bidirectional

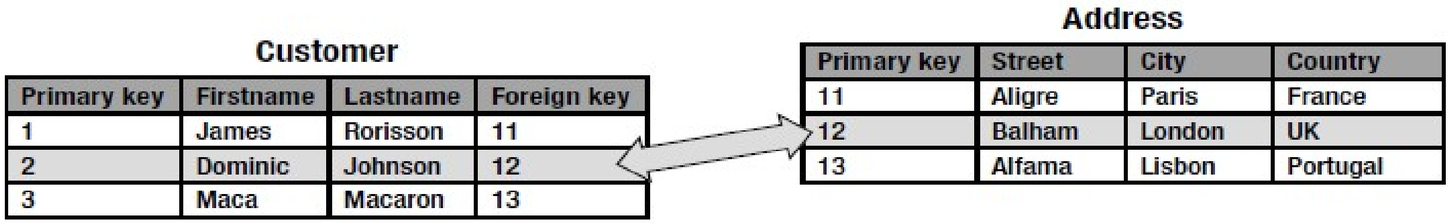


Multiplicity on class associations

Entity Relationships

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

Relationships in Relational Databases



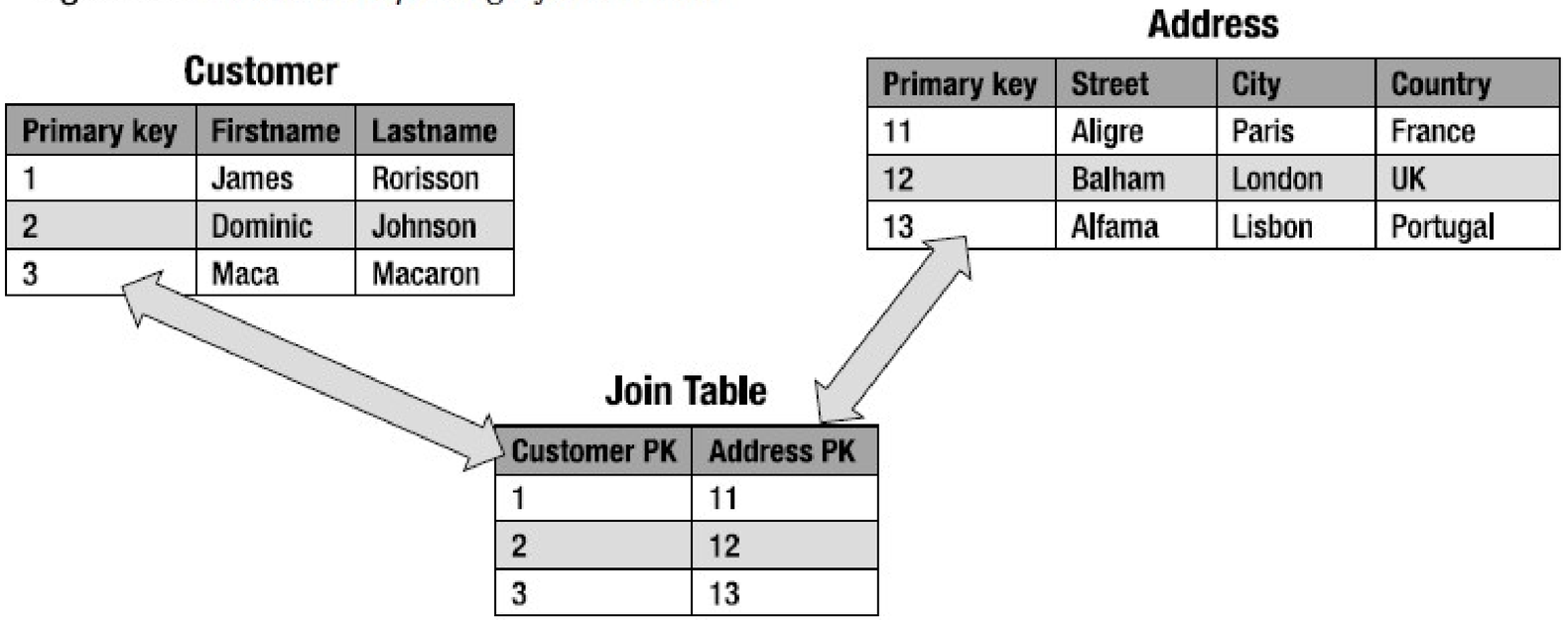
Customer

Primary key	Firstname	Lastname	Foreign key
1	James	Rorisson	11
2	Dominic	Johnson	12
3	Maca	Macaron	13

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Figure 3-9. A relationship using a join column



Customer

Primary key	Firstname	Lastname
1	James	Rorisson
2	Dominic	Johnson
3	Maca	Macaron

Address

Primary key	Street	City	Country
11	Aligre	Paris	France
12	Balham	London	UK
13	Alfama	Lisbon	Portugal

Join Table

Customer PK	Address PK
1	11
2	12
3	13

Figure 3-10. A relationship using a join table

Entity Relationships

Table 3-1. All Possible Cardinality-Direction Combinations

Cardinality	Direction
One-to-one	Unidirectional
One-to-one	Bidirectional
One-to-many	Unidirectional
Many-to-one/one-to-many	Bidirectional
Many-to-one	Unidirectional
Many-to-many	Unidirectional
Many-to-many	Bidirectional

one-to-one mapping

- In a one-to-one mapping the owner can be either the source entity or the target entity.
- one-to-one mapping is defined by annotating the owner entity with the @OneToOne annotations
- If the one-to-one mapping is bidirectional the inverse side of the relationship need to be specified too
- In the non owner entity, the @OneToOne annotations must come with the mappedBy element

@OneToOne annotation in Employee.java

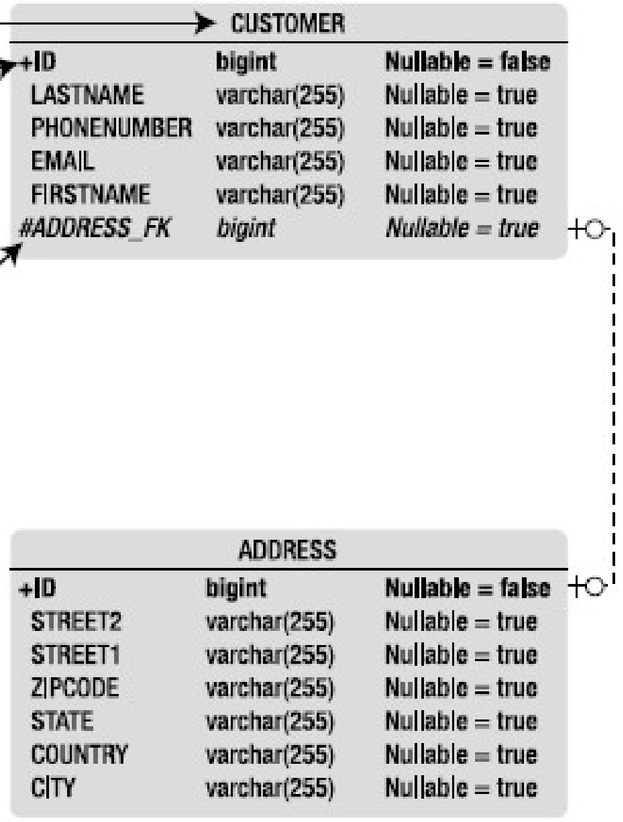
```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne
    private ParkingSpace parkingSpace;
    ...
}
```

@OneToOne annotation (inverse side) in ParkingSpace.java

```
@Entity
public class ParkingSpace {
    @Id private int id;
    @OneToOne(mappedBy="parkingSpace")
    private Employee employee;
    ...
}
```

One to one Bidirectional

```java
@Entity
public class Customer

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @OneToOne
    @JoinColumn(name = "address_fk")
    private Address address;
}

@Entity
public class Address {

    @Id @GeneratedValue
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    @OneToOne(mappedBy = "address")
    private Customer customer;
}
```



One to many mapping

- In an Many-to-one mapping the owner of the relationship is the source entity.
- A Many-to-one mapping is defined by annotating the source entity with @ManyToOne annotation

@ManyToOne annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToOne
    private Department department;
    ...
}
```

@OneToMany annotation in Department.java

```
@Entity
public class Department {
    @Id private int id;
    @OneToMany(mappedBy="department")
    private Collection<Employee> employees;
    ...
}
```

The attribute on the target entity that owns the relationship

Many-to-Many mapping

- In a Many-to-Many mapping there is no join column . The only way to implement such a mapping is by means of a join
- table Therefore, we can arbitrarily specify as owner either the source entity or the target entity
- If the many-to-many mapping is bi-directional, the inverse side of the relationship need to be specified too.
- In the non owner entity the @ManyToMany annotation must come with mappedBy elements

@ManyToMany annotation in Employee.java

```
@Entity
public class Employee {
    @Id private int id;
    @ManyToMany
    private Collection<Project> projects;
        ...
}
```

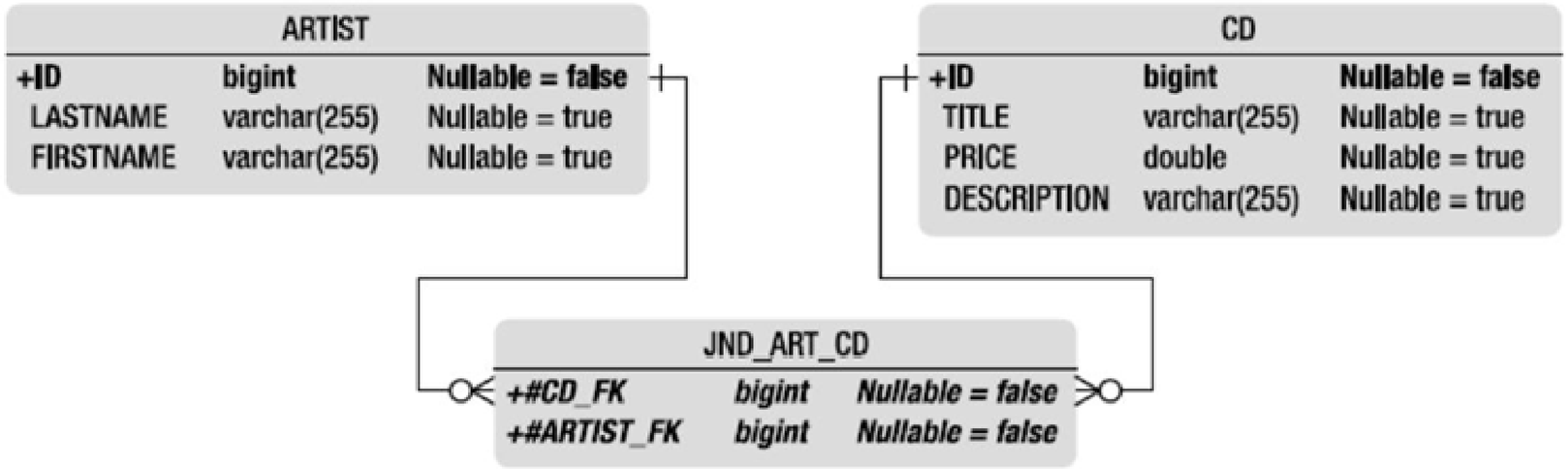
@ManyToMany annotation (inverse side) in Project.java

```
@Entity
public class Project {
    @Id private int id;
    @ManyToMany(mappedBy="projects")
    private Collection<Employee> employees;
    ...
}
```

Many to Many Bi-directional

```java
@Entity
public class CD {

    @Id @GeneratedValue
    private Long id;
    private String title;
    private Float price;
    private String description;
    @ManyToMany(mappedBy = "appearsOnCDs")
    private List<Artist> createdByArtists;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Artist {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    @ManyToMany
    @JoinTable(name = "jnd_art_cd", ↪
        joinColumns = @JoinColumn(name = "artist_fk"), ↪
        inverseJoinColumns = @JoinColumn(name = "cd_fk"))
    private List<CD> appearsOnCDs;

    // Constructors, getters, setters
}
```



@OrderBy

Dynamic ordering can be done with the @OrderBy annotation. “Dynamically” means that the ordering of the elements of a collection is made when the association is retrieved

```
@Entity
public class Comment {

    @Id @GeneratedValue
    private Long id;
    private String nickname;
    private String content;
    private Integer note;
    @Column(name = "posted_date")
    @Temporal(TemporalType.TIMESTAMP)
    private Date postedDate;

    // Constructors, getters, setters
}
```

```
@Entity
public class News {

    @Id @GeneratedValue
    private Long id;
    @Column(nullable = false)
    private String content;
    @OneToMany(fetch = FetchType.EAGER)
    @OrderBy("postedDate DESC")
    private List<Comment> comments;

    // Constructors, getters, setters
}
```

Lazy Loading

- The performance can be optimized by deferring the fetching of such data until they are actually needeD. This design pattern is called lazy loading
- At relationship level, lazy loading can be of great help in enhancing performance because it can reduce the amount of SQL get executed.
- the fetch mode can be specified on any of the four relationship mapping types
- The parkingSpace attributers may not be loaded each time employee is Loaded

Lazy loading of the `parkingSpace` attribute

```
@Entity
public class Employee {
    @Id private int id;
    @OneToOne(fetch=FetchType.LAZY)
    private ParkingSpace parkingSpace;
    ...
}
```

When the fetch mode is not specified

- On a single valued relationship the related object is guaranteed to be loaded eagerly and Collection-valued relationship default to be lazily loaded
- In case of bi-directional relationship the mode might be lazy on one side but eager on the other
- Quite common situation, relationship are often accessed in different way depending on the direction from which navigation occurs.
- The directive to lazily fetch an attribute is meant only to be hint to the persistence provider
- The provider is not required to respect the request as the behaviour of the entity will not be compromised if the provider decides to eagerly load data.
- The converse is not true because specifying that an attributes be eagerly fetched might be critical to access the entity once detached.

Cascading operations

- Hibernate/JPA provides a mechanism to define when operations such as save()/persist() should be automatically cascaded across relationships
- You need to be sure that Address instance bas been set on Employee instance before invoking persist() on it.
- The cascade attribute accepts several possible values coming from the cascade Type enumerations
- PERSIST, REFRESH, REMOVE, MERGE and DETACH
- The constant ALL is a shorthand for declaring that all five operations should be cascaded.
- As for relationship, cascade settings are unidirectional They must be explicitly set on both sides of a relationship if the same behaviours is required

Enabling cascade persist

```
@Entity
public class Employee {
    @ManyToOne(cascade=CascadeType.PERSIST)
    Address address;
}
```

Collection of Basic Types

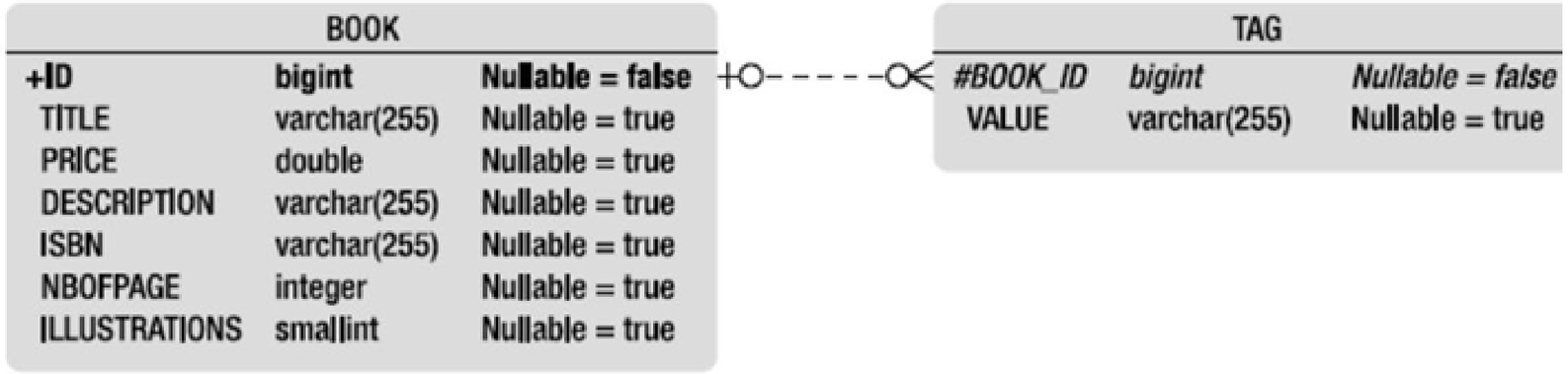
- **@ElementCollection annotation is used to indicate that an attribute of type java.util.Collection contains a collection of instances of basic types (i.e., nonentities)**

Attribute can be of the following types:

1. **java.util.Collection: Generic root interface in the collection hierarchy.**
2. **java.util.Set: Collection that prevents the insertion of duplicate elements.**
3. **java.util.List: Collection used when the elements need to be retrieved in some user-defined order.**

Collection of Basic Types

```java
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Float price;
    private String description;
    private String isbn;
    private Integer nbOfPage;
    private Boolean illustrations;
    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(name = "Tag")
    @Column(name = "Value")
    private List<String> tags = new ArrayList<String>();

    // Constructors, getters, setters
}
```



Embeddables

```java
@Embeddable
public class Address {

    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;

    // Constructors, getters, setters
}
```

```java
@Entity
public class Customer {

    @Id @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;
    @Embedded
    private Address address;

    // Constructors, getters, setters
}
```

Listing 3-35. *Structure of the CUSTOMER Table with All the Address Attributes*

```
create table CUSTOMER (
   ID BIGINT not null,
   LASTNAME VARCHAR(255),
   PHONENUMBER VARCHAR(255),
   EMAIL VARCHAR(255),
   FIRSTNAME VARCHAR(255),
   STREET2 VARCHAR(255),
   STREET1 VARCHAR(255),
   ZIPCODE VARCHAR(255),
   STATE VARCHAR(255),
   COUNTRY VARCHAR(255),
   CITY VARCHAR(255),
   primary key (ID)
);
```

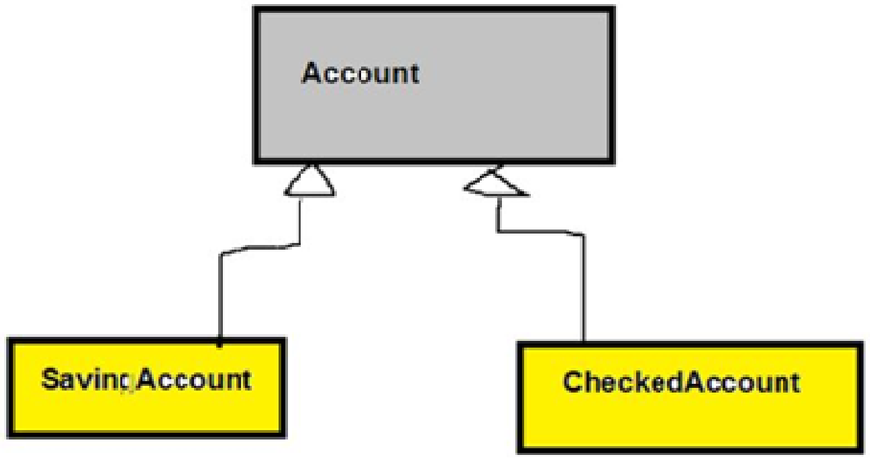
Inheritance Mapping

Inheritance can be used also for persistent objects, for factoring out data members inherited by multiple subclasses

The mapping of a hierarchy to the database can follow

different strategies: -

1. **Single table per hierarchy**
2. **Table per class(table per concrete class)**
3. **Joined**



Inheritance Mapping

Single table (single table per hierarchy)

- Here only one table is going to be created, all fields mapped to single table.
- Not very memory efficient, Faster, support polymorphic quaries

Joined (as per normalization)

- Separate table mapped to all classes in the hierarchy
- Three table is created Account, SavingAccount, CurrentAccount, as per normalization, table are need to join to get all data (sql outer join)

Table per class(Table per concrete class)

- Two table is going to create SavingAccount and CurrentAccount
- Sql Union operation is required to get data, Identity auto gen key is not supported
- AUB, Not Supported by all vendors

Single table (single table per hierarchy)

```java
@Entity
@Table(name = "Account")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="accountType",discriminatorType=DiscriminatorType.STRING)
public class Account{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int accountId;
    private String accountHolderName;
    private double balance;
```

```java
@Entity
@Table(name="Account")
@DiscriminatorValue("S")
public class SavingAccount extends Account{
    private double intrestRate;
```

```java
@Entity
@Table(name="Account")
@DiscriminatorValue("C")
public class CurrentAccount extends Account{
    private double overdraft;
```

```
mysql> desc account;
+-------------------+--------------+------+-----+---------+
| Field             | Type         | Null | Key | Default |
+-------------------+--------------+------+-----+---------+
| accountType       | varchar(31)  | NO   |     | NULL    |
| accountId         | int(11)      | NO   | PRI | NULL    |
| accountHolderName | varchar(255) | YES  |     | NULL    |
| balance           | double       | NO   |     | NULL    |
| overdraft         | double       | YES  |     | NULL    |
| intrestRate       | double       | YES  |     | NULL    |
+-------------------+--------------+------+-----+---------+
```

Table per class(Table Per Concreat class)

```java
@Entity
@Table(name = "Account")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)

public class Account{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    private int accountId;
    private String accountHolderName;
    private double balance;
```

```java
@Entity
@Table(name="SavingAccount")
public class SavingAccount extends Account{
    private double intrestRate;
```

```java
@Entity
@Table(name="CurrentAccount")
public class CurrentAccount extends Account{
    private double overdraft;
```

Module 9: Spring Transaction Management

rgupta.mtech@gmail.com

Need of Transaction Management

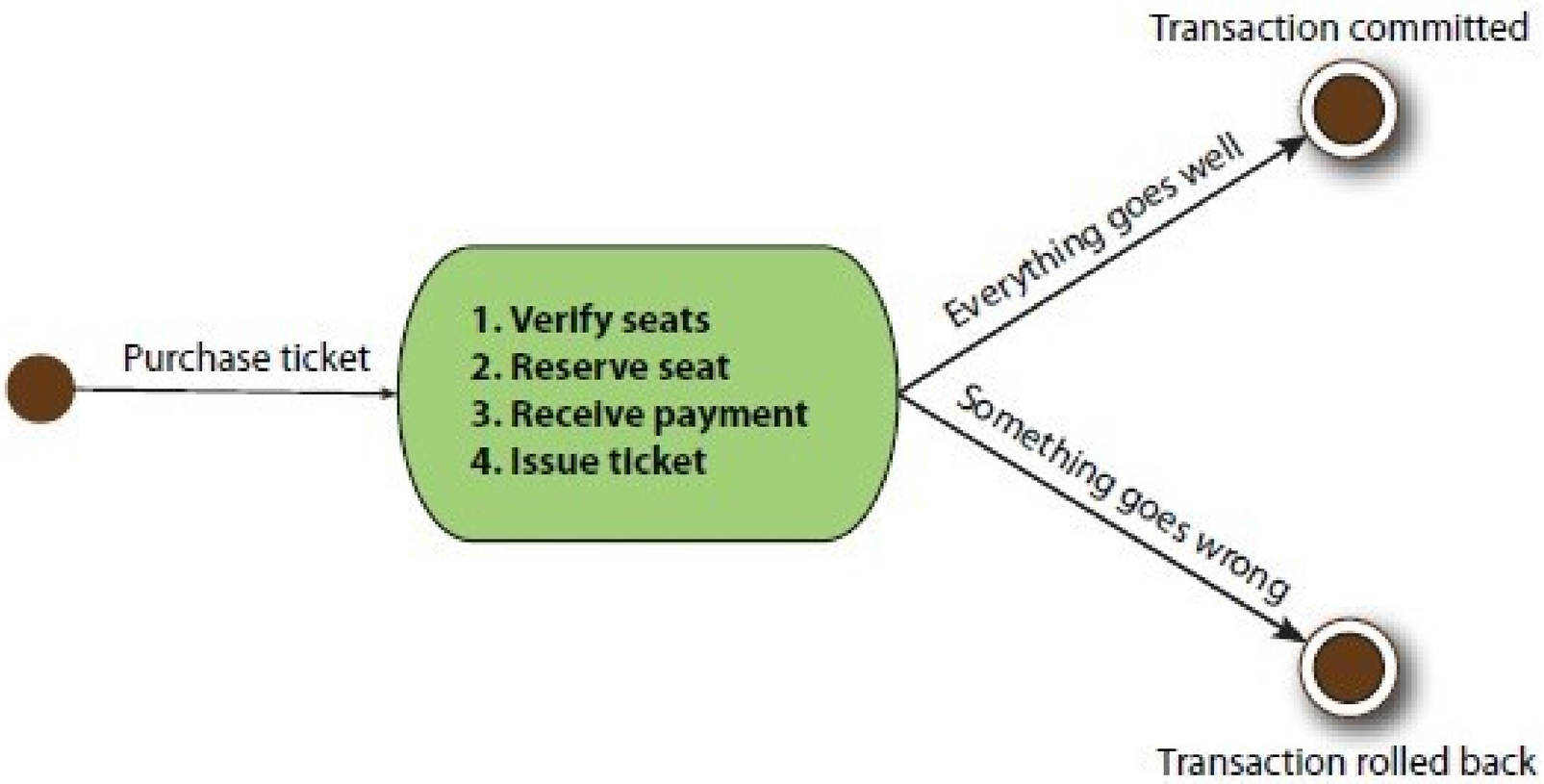


Figure 6.1 The steps involved when purchasing a movie ticket should be all or nothing. If every step is successful, then the entire transaction is successful. Otherwise, the steps should be rolled back—as if they never happened.

rgupta.mtech@gmail.com

Transactions

- **Transaction is a series of actions that must occured as a group. If any portion of that group fails then entire transaction must roll back.**
- **A.C.I.D. is the well-known acronym for the characteristics of a successful and valid transaction.**

> In software, all-or-nothing operations are called *transactions*. Transactions allow you to group several operations into a single unit of work that either fully happens or fully doesn't happen. If everything goes well, then the transaction is a success. But if anything goes wrong, the slate is wiped clean and it's as if nothing ever happened.

rgupta.mtech@gmail.com

ACID properties

Property	Description
Atomicity	A transaction is composed of one or more operations grouped in a unit of work. At the conclusion of the transaction, either these operations are all performed successfully (commit) or none of them is performed at all (rollback) if something unexpected or irrecoverable happens.
Consistency	At the conclusion of the transaction, the data are left in a consistent state.
Isolation	The intermediate state of a transaction is not visible to external applications.
Durability	Once the transaction is committed, the changes made to the data are visible to other applications.

Spring tx support

- Spring doesn’t directly manage transactions .
- Sprnig comes with a selections of transaction managements that delegate responsibility for tansaction manamgement to a plateform specific transaction implemenation provided by either JTA or persistance mechanism.
- Each of these transaction managers act as a facade to a plateform specific transaction management implementation

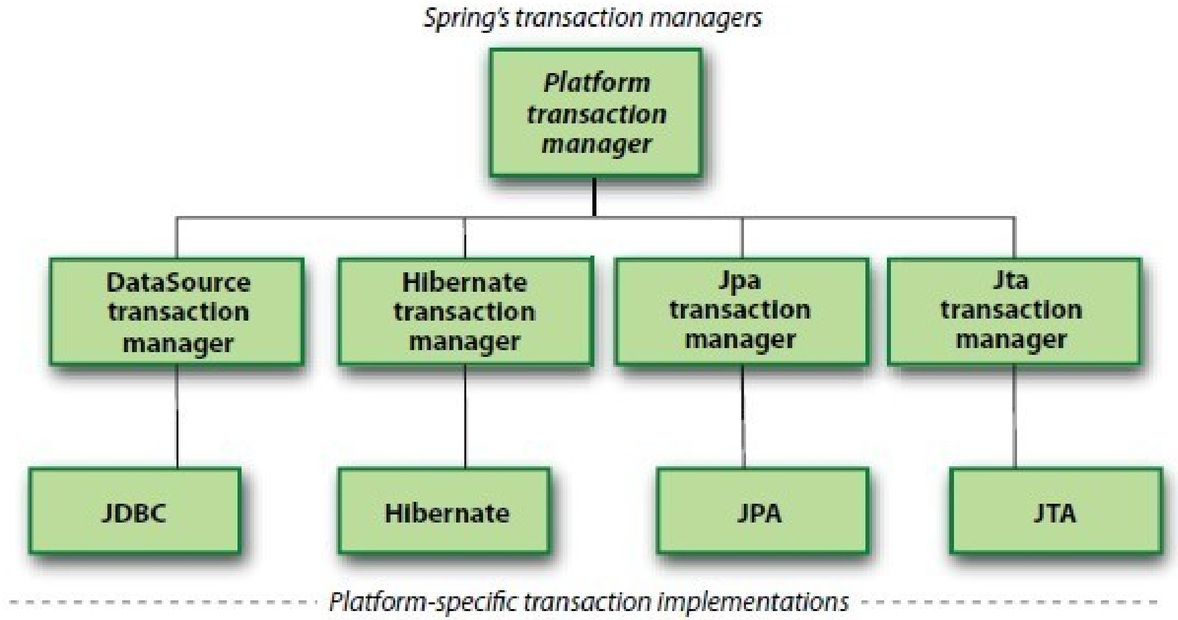


Figure 6.2 Spring’s transaction managers delegate transaction-management responsibility to platform-specific transaction implementations.

rgupta.mtech@gmail.com

Spring supported transaction managers

Transaction manager (`org.springframework.*`)	Use it when...
`jca.cci.connection.CciLocalTransactionManager`	Using Spring's support for Java EE Connector Architecture (JCA) and the Common Client Interface (CCI).
`jdbc.datasource.DataSourceTransactionManager`	Working with Spring's JDBC abstraction support. Also useful when using iBATIS for persistence.
`jms.connection.JmsTransactionManager`	Using JMS 1.1+.
`jms.connection.JmsTransactionManager102`	Using JMS 1.0.2.
`orm.hibernate3.HibernateTransactionManager`	Using Hibernate 3 for persistence.
`orm.jdo.JdoTransactionManager`	Using JDO for persistence.
`orm.jpa.JpaTransactionManager`	Using the Java Persistence API (JPA) for persistence.
`transaction.jta.JtaTransactionManager`	You need distributed transactions or when no other transaction manager fits the need.
`transaction.jta.OC4JJtaTransactionManager`	Using Oracle's OC4J JEE container.
`transaction.jta.WebLogicJtaTransactionManager`	You need distributed transactions and your application is running within WebLogic.
`transaction.jta.WebSphereUowTransactionManager`	You need transactions managed by a `UOWManager` WebSphere.

JDBC Transaction

- If we are using JDBC, springs DataSourceTransactionManager will handle transactional boundaries for us.
- To use DataSourceTransaction manager write it into your applicationcontext defination using XML or annotation

- Behind the scenes, DataSourceTransactionManager manage transaction by making call to java.sql.Connection object retrive for datasource.

- For Instance an successful transaction is commited by calling commit() method and a rollback by calling rollback() method

```
<bean id="transactionManager" class="org.springframework.jdbc.
    ➥datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

rgupta.mtech@gmail.com

Hibernate /JPA Transaction

- If we are using Hibernate then we need to configure HibernateTransactionManager The sessionFactory property should be wired with a Hibernate SessionFactory

```
<bean id="transactionManager" class="org.springframework.
      ➥orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

If we are using JPA then we need to configure JpaTransactionManager

```
<bean id="transactionManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

rgupta.mtech@gmail.com

Defining transaction attributes

- In Spring, declarative transactions are defined with *transaction attributes.*
- ***A transaction attribute is a description of how transaction policies should be applied to a method.***
- *There are **five** facets of a transaction attribute*
- *Although Spring provides several mechanisms for declaring transactions, all of them rely on these five parameters to govern how transaction policies are administered.*
- *Regardless of which declarative transaction mechanism we use, we have to define these attributes*

Figure 6.3 Declarative transactions are defined in terms of propagation behavior, isolation level, read-only hints, timeout, and rollback rules.

rgupta.mtech@gmail.com

PROPAGATION Transaction

Propagation behavior	What it means
PROPAGATION_MANDATORY	Indicates that the method must run within a transaction. If no existing transaction is in progress, an exception will be thrown.
PROPAGATION_NESTED	Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction. If no enclosing transaction exists, behaves like PROPAGATION_REQUIRED. Vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported.
PROPAGATION_NEVER	Indicates that the current method shouldn't run within a transactional context. If an existing transaction is in progress, an exception will be thrown.
PROPAGATION_NOT_SUPPORTED	Indicates that the method shouldn't run within a transaction. If an existing transaction is in progress, it'll be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_REQUIRED	Indicates that the current method must run within a transaction. If an existing transaction is in progress, the method will run within that transaction. Otherwise, a new transaction will be started.
PROPAGATION_REQUIRES_NEW	Indicates that the current method must run within its own transaction. A new transaction is started and if an existing transaction is in progress, it'll be suspended for the duration of the method. If using JTATransactionManager, access to TransactionManager is required.
PROPAGATION_SUPPORTS	Indicates that the current method doesn't require a transactional context, but may run within a transaction if one is already in progress.

rgupta.mtech@gmail.com

ISOLATION LEVE S

In a typical application, multiple transactions run concurrently, often working with the same data to get their jobs done. Concurrency, while necessary, can lead to the following problems:

- *Dirty reads* occur when one transaction reads data that has been written but not yet committed by another transaction. If the changes are later rolled back, the data obtained by the first transaction will be invalid.
- *Nonrepeatable reads* happen when a transaction performs the same query two or more times and each time the data is different. This is usually due to another concurrent transaction updating the data between the queries.
- *Phantom reads* are similar to nonrepeatable reads. These occur when a transaction (T1) reads several rows, and then a concurrent transaction (T2) inserts rows. Upon subsequent queries, the first transaction (T1) finds additional rows that weren't there before.

rgupta.mtech@gmail.com

ISOLATION LEVE S

Isolation levels determine to what degree a transaction may be impacted by other transactions being performed in parallel

Isolation level	What it means
ISOLATION_DEFAULT	Use the default isolation level of the underlying data store.
ISOLATION_READ_UNCOMMITTED	Allows you to read changes that haven't yet been committed. May result in dirty reads, phantom reads, and nonrepeatable reads.
ISOLATION_READ_COMMITTED	Allows reads from concurrent transactions that have been committed. Dirty reads are prevented, but phantom and nonrepeatable reads may still occur.
ISOLATION_REPEATABLE_READ	Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonrepeatable reads are prevented, but phantom reads may still occur.
ISOLATION_SERIALIZABLE	This fully ACID-compliant isolation level ensures that dirty reads, nonrepeatable reads, and phantom reads are all prevented. This is the slowest of all isolation levels because it's typically accomplished by doing full table locks on the tables involved in the transaction.

rgupta.mtech@gmail.com

READ-ONLY

If a transaction performs only read operations against the underlying datastore, the data store maybe able to apply certain optimizations that take advantage of the read-only nature of the transaction.
By declaring a transaction as read- only, you give the underlying data store the opportunity to apply those optimizations as it sees fit

rgupta.mtech@gmail.com

TRANSACTION TIMEOUT

- Suppose that our transation become unexpectedly long running, because transaction may involve locks on the underlying data store, long running transactions can tie up database resources unnecessarily.
- Instead of waiting it out you can declare a transaction to automatically rollback after a certain number of seconds.
- Because the timeout clock begins ticking when a transaction start, it only makes sense to declare a transaction timeout on methods with propagation behaviours that may start a new transaction (PROPAGATION_REQUIRED,PROPAGATION_REQUIRES_NEW, and PROPAGATION_NESTED

rgupta.mtech@gmail.com

ROLLBACK RULES

- By default, transactions are rolled back only on runtime exceptions and not on checked exceptions. (This behaviour is consistent with rollback behaviour in EJBs)
- But you an declare that a transaction be rolled back on specific checked exceptions as well as run time exceptions.
- Likewise, you can declare that a transaction not roll back on specified exceptions, even if those exceptions are runtime exceptions.

rgupta.mtech@gmail.com

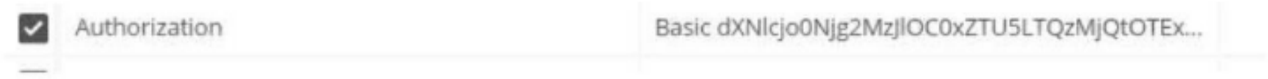
Module 9: Spring Security

rgupta.mtech@gmail.com

Configuration spring rest security

- Step 1: just add spring security dependencies to boot project, an default user/password is generated

```
Using generated security password: 468632e8-1e59-4324-911f-b493308f6408
```



- Step 2: configre username/password

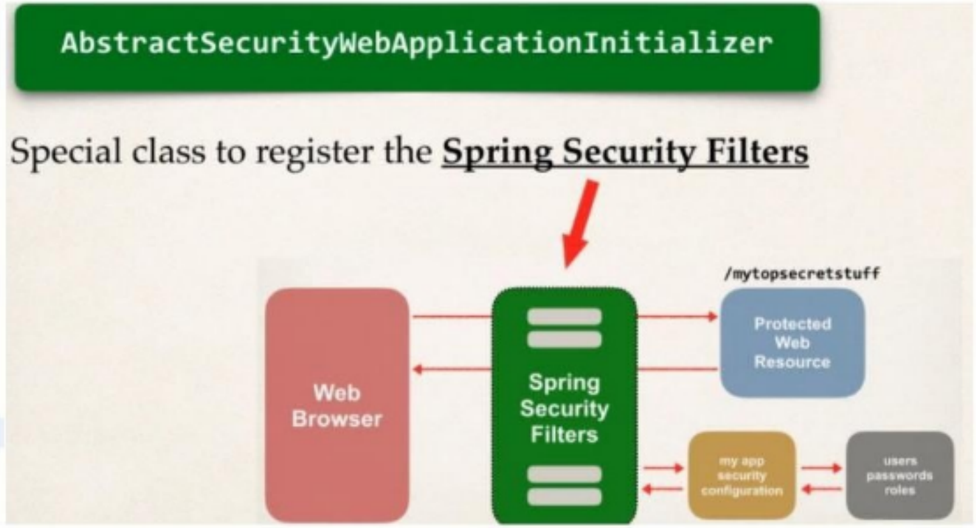
```
spring.security.user.name=admin
spring.security.user.password=admin
```

- Configuration of DelegatingFilterProxy:

  No need as spring boot do behind

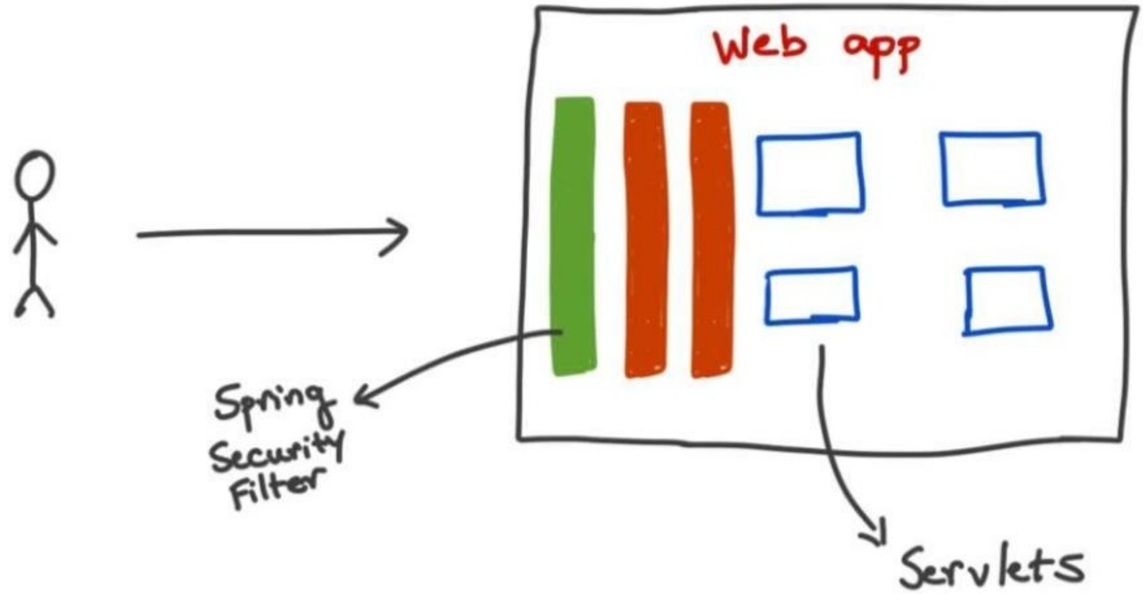
  the scenes

```
public class WebSecConfigInitilizer extends
        AbstractSecurityWebApplicationInitializer{

}
```



rgupta.mtech@gmail.com

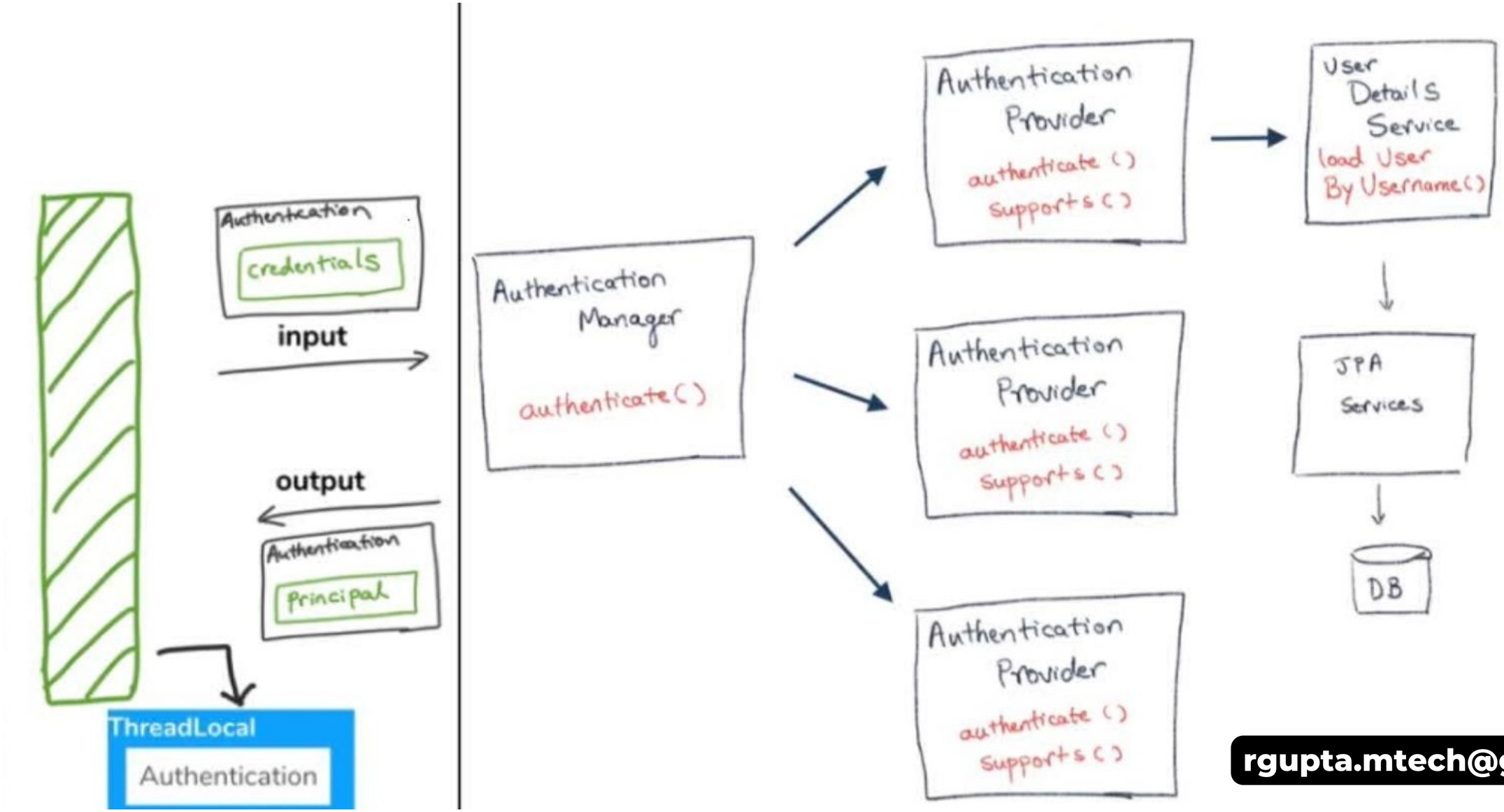
How Spring security works




```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

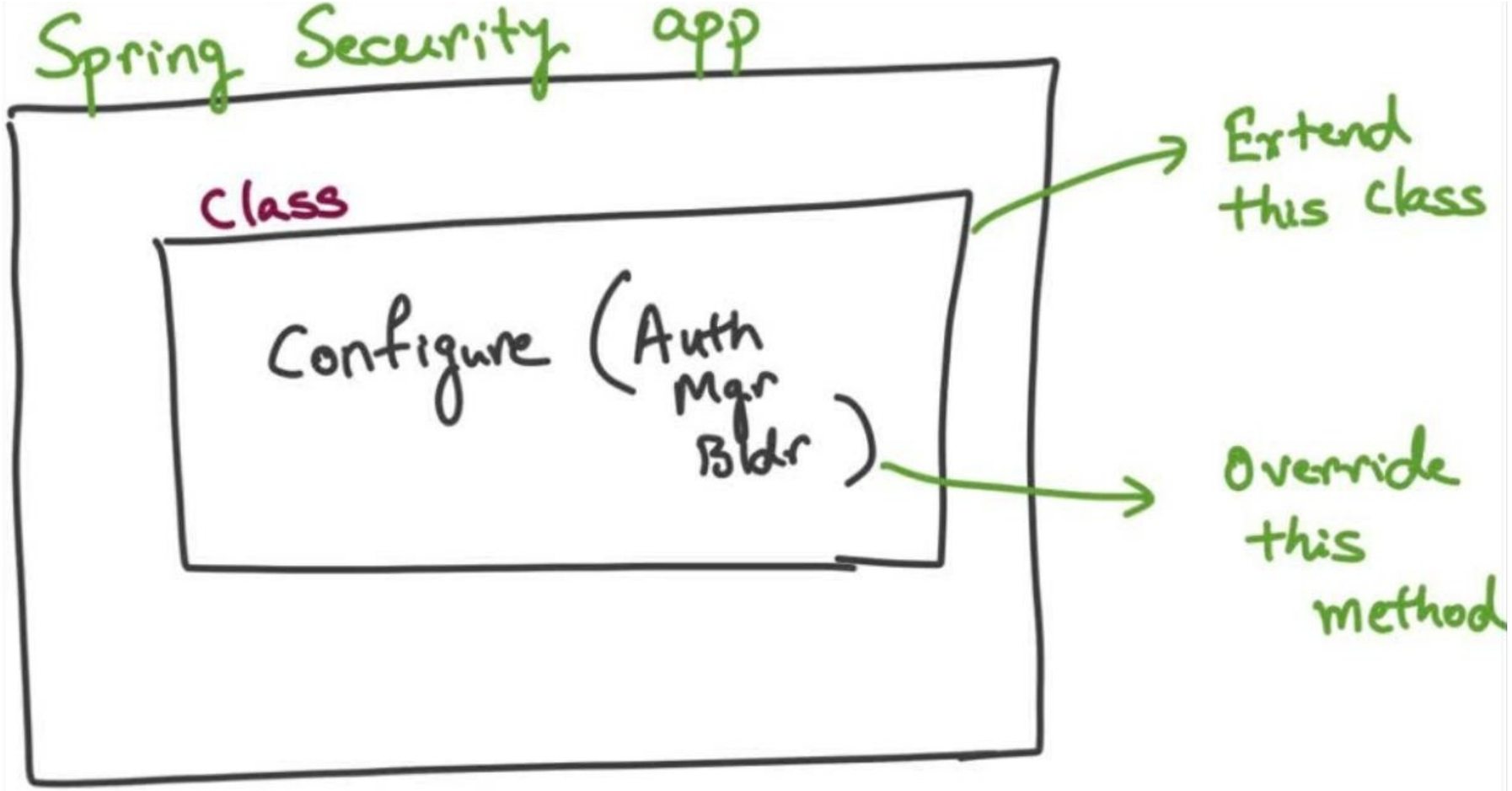
rgupta.mtech@gmail.com

How Spring security works



rgupta.mtech@gmail.com

How Spring security works



How to get hold of AuthenticationManagerBuilder?
=> WebSecurityConfigureAdeptor

rgupta.mtech@gmail.com

How Spring security works

1> Authentication filter creates an “Authentication Request” and passes it to Authentication manager

2 > Authentication Manager Delegates to one or more Authentication Provider

3> Authentication Provider Uses UserDetailsService to load the UserDetails and return an “Authenticated Principle”

4 > Authentication filter sets the authentication in the security context

rgupta.mtech@gmail.com

Configuration spring security

```java
@RestController
public class HelloController {

    @GetMapping("home")
    public String home() { return "home"; }

    @PreAuthorize("hasAuthority('ROLE_MGR')")
    @GetMapping("mgr")
    public String mgr() { return "mgr"; }
    @PreAuthorize("hasAuthority('ROLE_MGR') or hasAuthority('ROLE_CLERK')")
    @GetMapping("clerk")
    public String clerk() { return "clerk"; }
}
```

rgupta.mtech@gmail.com

Configuration spring security

```
@Component
@EnableWebSecurity(debug = true)
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecConfig2 extends WebSecurityConfigurerAdapter {
    @Autowired
    private UserDetailsService userDetailsService;
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.inMemoryAuthentication() InMemoryUserDetailsManagerConfigurer<AuthenticationManagerBuilder>
                .withUser( username: "raj").password("raj123").roles("MGR") UserDetailsManagerConfigurer<...>.UserDetailsBuilder
                .and() InMemoryUserDetailsManagerConfigurer<AuthenticationManagerBuilder>
                .withUser( username: "ekta").password("ekta123").roles("CLERK");
    }
    @Bean
    public PasswordEncoder encoding(){
        return NoOpPasswordEncoder.getInstance();
    }
}
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf() CsrfConfigurer<HttpSecurity>
            .disable() HttpSecurity
            .authorizeRequests().anyRequest().authenticated() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/mgr/**").hasAnyRole( ...roles: "MGR")
            .antMatchers( ...antPatterns: "/clerk/**").hasAnyRole( ...roles: "MGR", "CLERK")
            .and() HttpSecurity
            .httpBasic() HttpBasicConfigurer<HttpSecurity>
            .and() HttpSecurity
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

rgupta.mtech@gmail.com

Configuration security Hibernate config

```java
@Entity
@Table(name = "user_table")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(unique = true)
    private String username;

    private String password;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(name = "user_roles")
    private List<String> roles=new ArrayList<>();

    public UserEntity(String username, String password, List<String> roles) {
        this.username = username;
        this.password = password;
        this.roles = roles;
    }
}
```

```java
@Repository
public interface UserRepo extends JpaRepository<UserEntity, Integer> {
    public UserEntity findByUsername(String userName);
}
```

rgupta.mtech@gmail.com

Configuration security Hibernate config

```java
public interface UserService {
    public UserEntity findByUsername(String userName);
    public void addUser(UserEntity userEntity);
}
```

```java
@Service
@Transactional
public class UserServiceImpl implements UserService {
    @Autowired
    private UserRepo userRepo;

    @Override
    public UserEntity findByUsername(String userName) { return userRepo.findByUsername(userName); }

    @Override
    public void addUser(UserEntity userEntity) { userRepo.save(userEntity); }
}
```

rgupta.mtech@gmail.com

Configuration user Detail service

```java
@Service
public class DetailService implements UserDetailsService {
    //called by spring sec
    @Autowired
    private UserService userService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        UserEntity userEntity=userService.findByUsername(username);
        if(userEntity==null){
            throw new UsernameNotFoundException("user is invalid");
        }
        return  new SecUser(userEntity);
    }
}
```

rgupta.mtech@gmail.com

Converting my userEntity to the user that spring security understand

```java
//userEntity to user that spring sec understand
public class SecUser implements UserDetails {

    private UserEntity userEntity;

    public SecUser(UserEntity userEntity) { this.userEntity = userEntity; }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<String> roles=userEntity.getRoles();
        //i have to convert List<String> to Array of String String []
        String rolesArray[]=roles.toArray(new String[roles.size()]);

        return AuthorityUtils.createAuthorityList(rolesArray);
    }

    @Override
    public String getPassword() { return userEntity.getPassword(); }

    @Override
    public String getUsername() { return userEntity.getUsername(); }

    @Override
    public boolean isAccountNonExpired() { return true; }
```

rgupta.mtech@gmail.com

Configuration security Hibernate config

```java
@Service
public class DetailService implements UserDetailsService {
    //called by spring sec
    @Autowired
    private UserService userService;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        UserEntity userEntity=userService.findByUsername(username);
        if(userEntity==null){
            throw new UsernameNotFoundException("user is invalid");
        }
        return  new SecUser(userEntity);
    }
}
```

```java
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecConfig2 extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService);
    }
```

rgupta.mtech@gmail.com

WebSecurityConfigureAdaptor is Deprecated

```
public class SecConfig2 {

    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider authenticationProvider
                =new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService);
        authenticationProvider.setPasswordEncoder(encoding());
        return authenticationProvider;
    }
    @Bean
    public PasswordEncoder encoding(){
        return NoOpPasswordEncoder.getInstance();
    }
```

```
public SecurityFilterChain filterChain(HttpSecurity http)throws Exception{
    return  http.csrf() CsrfConfigurer<HttpSecurity>
            .disable() HttpSecurity
            .authorizeHttpRequests().anyRequest().authenticated() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
            .antMatchers( ...antPatterns: "/mgr/**").hasAnyRole( ...roles: "MGR")
            .antMatchers( ...antPatterns: "/clerk/**").hasAnyRole( ...roles: "MGR", "CLERK")
            .and() HttpSecurity
            .httpBasic() HttpBasicConfigurer<HttpSecurity>
            .and() HttpSecurity
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementConfigurer<HttpSecurity>
            .and() HttpSecurity
            .build();
}
```

rgupta.mtech@gmail.com

Method Level Security

step 1: Enable method level security

```
@EnableWebSecurity(debug = true)
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableMethodSecurity
public class SecConfig2 {
```

step 1: Apply method level security annotations

```
@RestController
public class HelloController {

    @GetMapping("home")
    public String home() { return "home"; }

    @PreAuthorize("hasAuthority('ROLE_MGR')")
    @GetMapping("mgr")
    public String mgr() { return "mgr"; }
    @PreAuthorize("hasAuthority('ROLE_MGR') or hasAuthority('ROLE_CLERK')")
    @GetMapping("clerk")
    public String clerk() { return "clerk"; }
}
```

step 1: Remove url seurity from security configuration

```
public SecurityFilterChain filterChain(HttpSecurity http)throws Exception{
    return  http.csrf() CsrfConfigurer<HttpSecurity>
            .disable() HttpSecurity
            .authorizeHttpRequests().anyRequest().authenticated() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
            .and() HttpSecurity
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and() HttpSecurity
            .build();
}
```

rgupta.mtech@gmail.com

JWT based security

JWT ?

It ships information that can be verified and trusted with a digital signature

Why JWT?

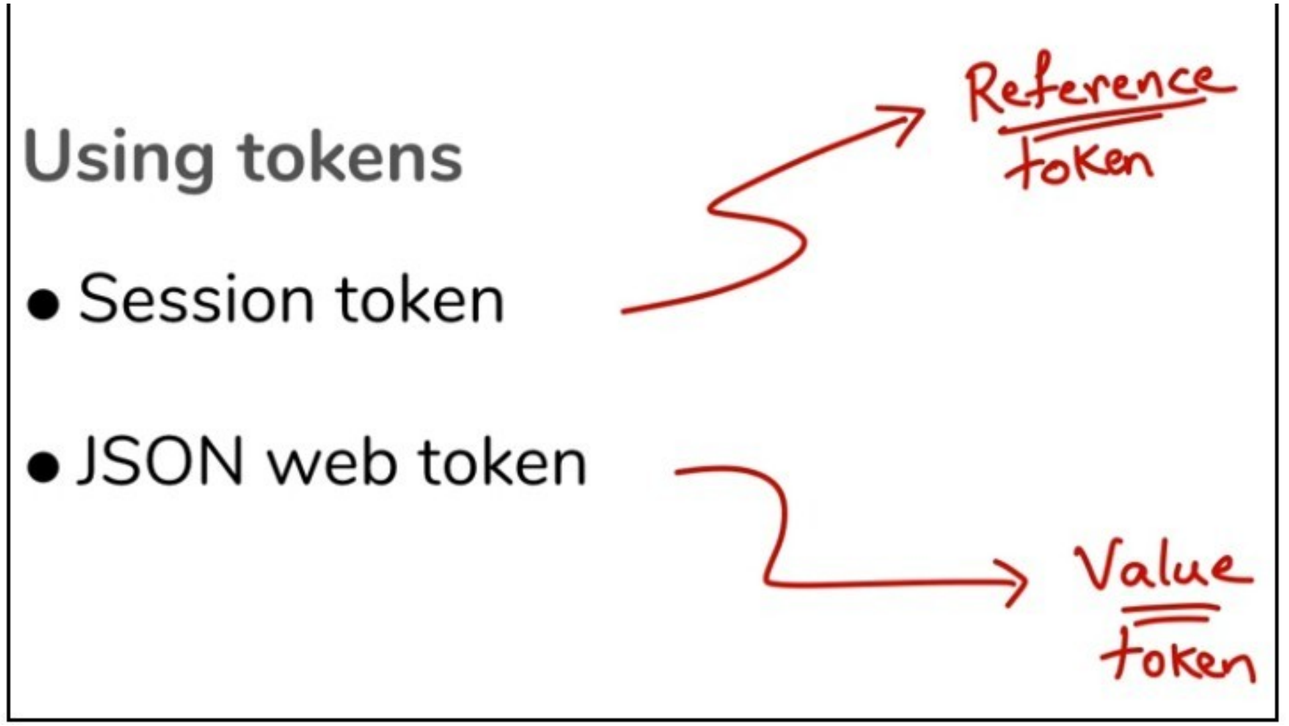
Stateless

JWT allow the server to verify the information contained in the JWT without necessarily storing state on the server.

http://secure.indas.on.ca

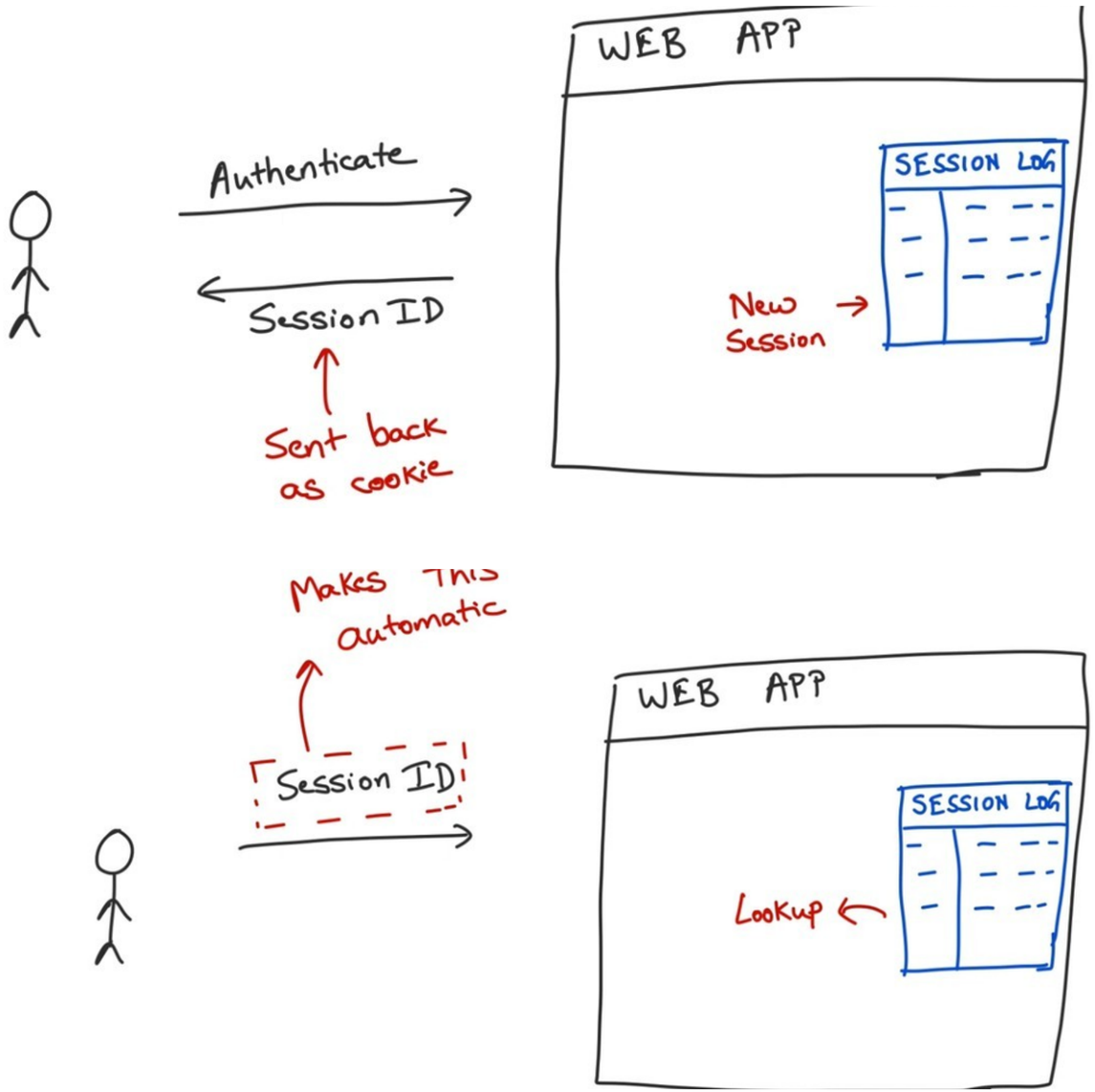
rgupta.mtech@gmail.com

Authorization strategies



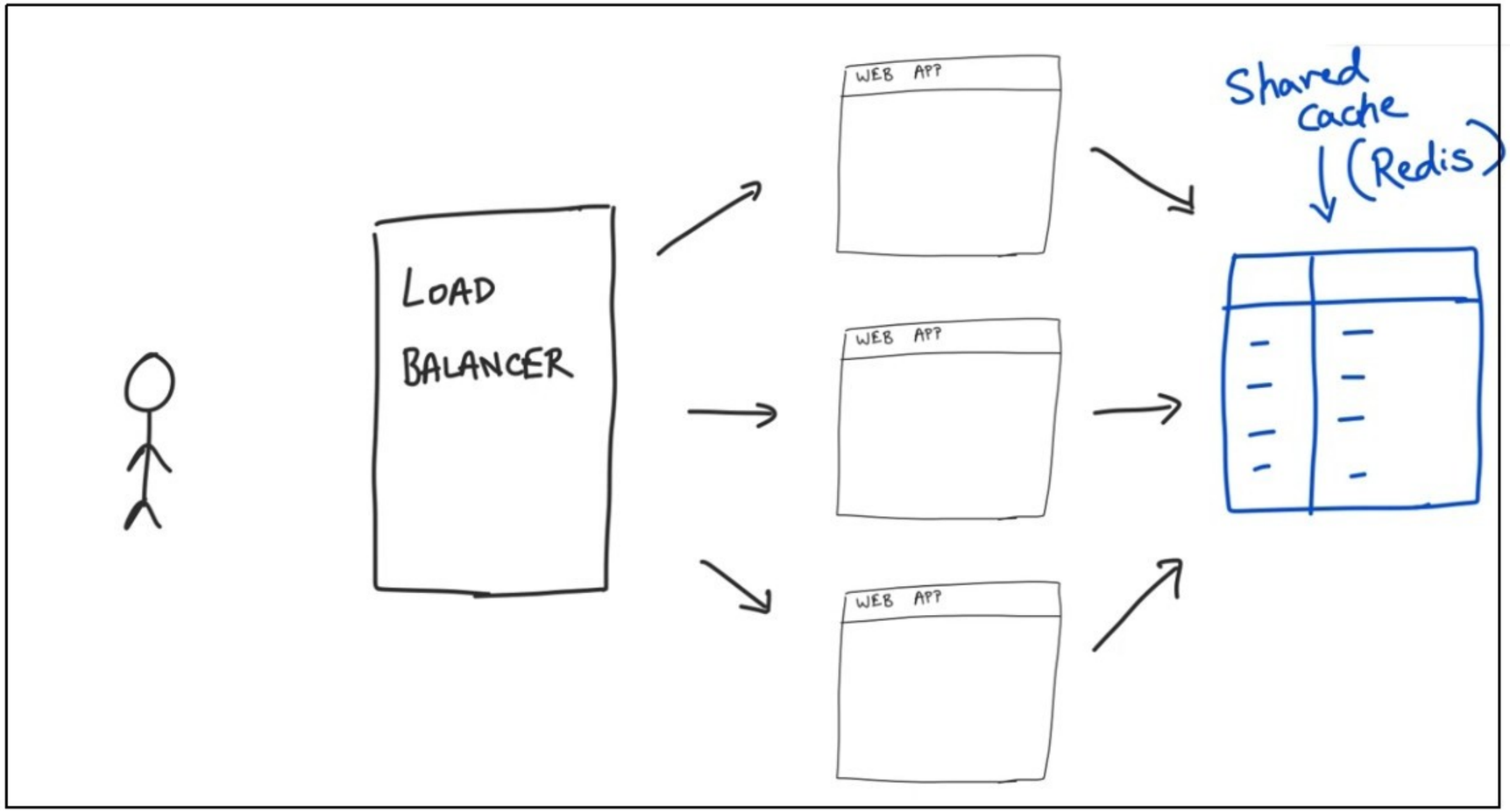
rgupta.mtech@gmail.com

Session based authorization



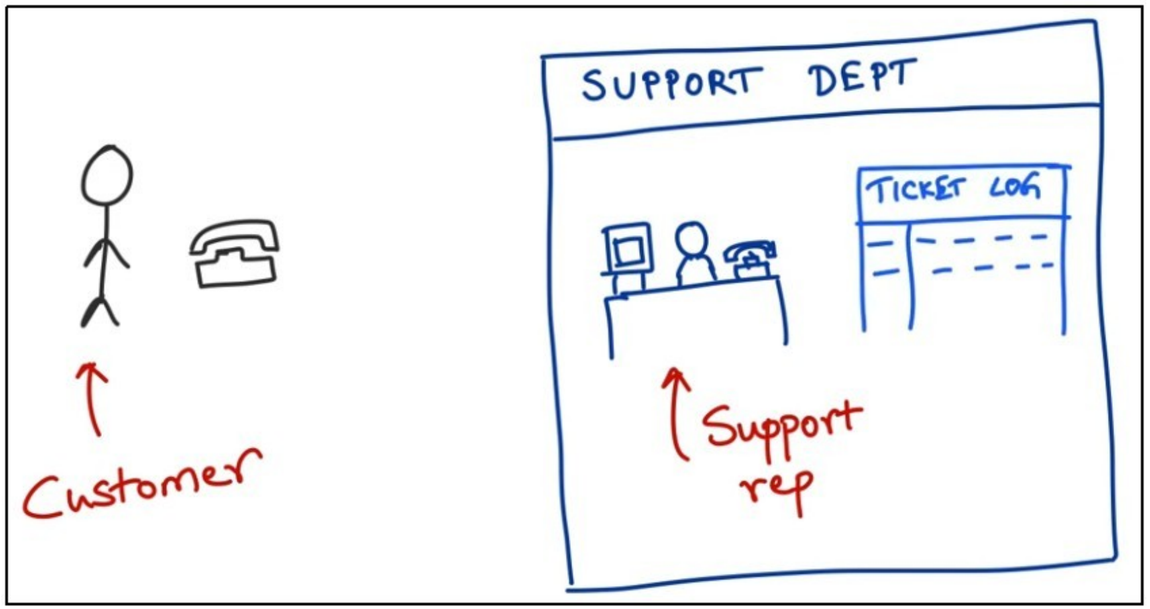
rgupta.mtech@gmail.com

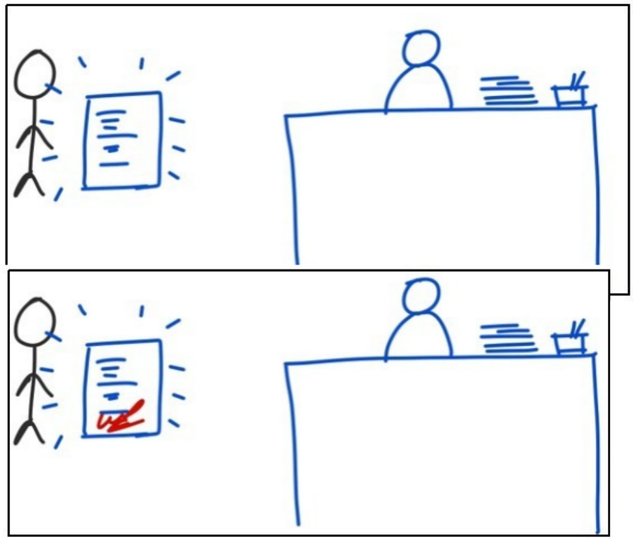
Session based authorization



rgupta.mtech@gmail.com

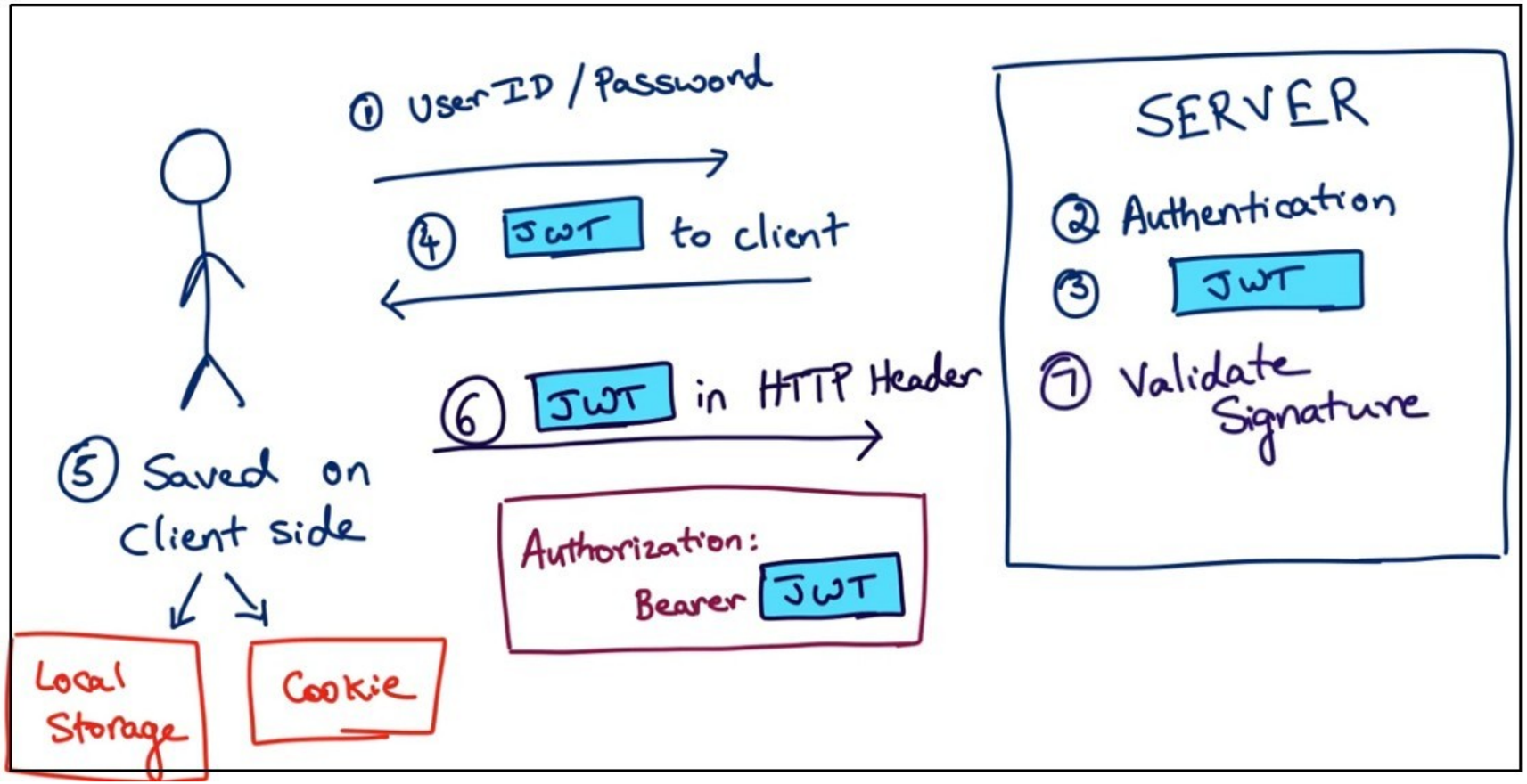
Session based vs JWT





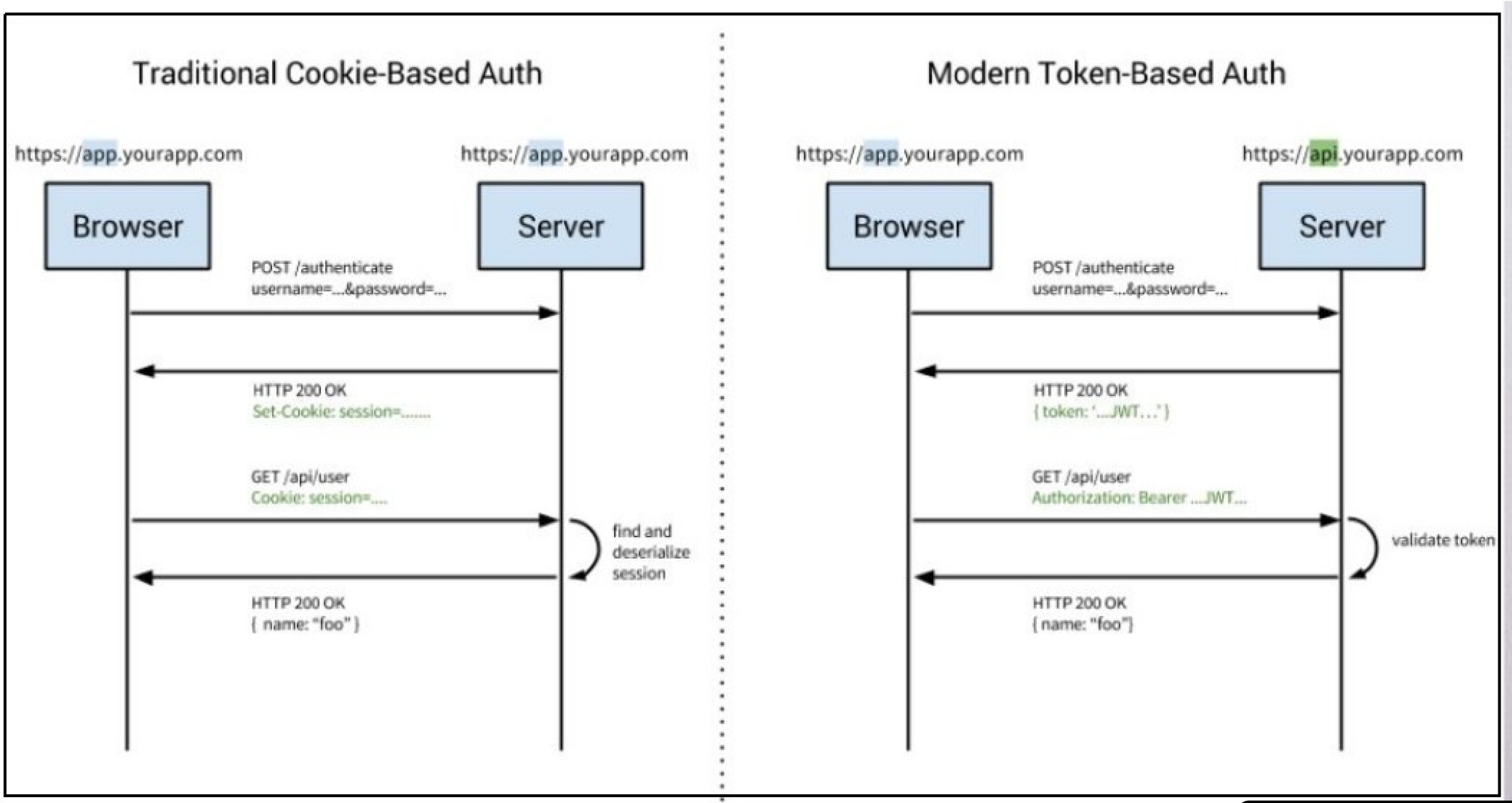
rgupta.mtech@gmail.com

JWT Token Flow

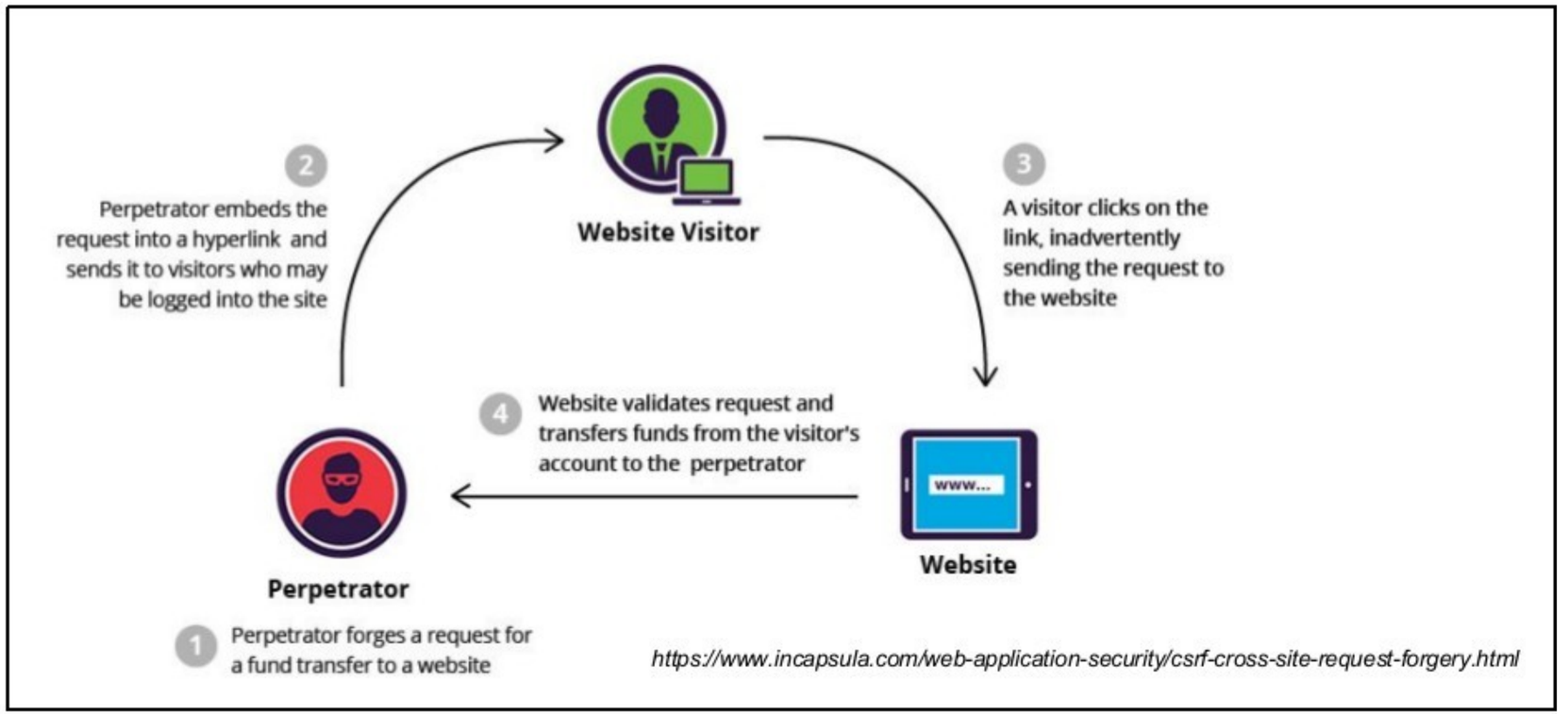


rgupta.mtech@gmail.com

Session based vs JWT Token Flow



No need to protect against CSRF



rgupta.mtech@gmail.com

JWTUtil

Step to configure JWT security

1. Create JWTUtil class that will create and validate jwt token
2. Define an rest end point that allow user to get jwt token after authentictation, by pass security for this end point
3. If username/password is correct issue jwt token to the user
4. customized spring security filter chain configuration and define AuthenticationManager
5. Define OncePerRequestFilter that validate the token and put user in securitycontextholder
6. Configure OncePerRequestFilter before UsernamePasswordAuthenticationFilter
7. Run application

rgupta.mtech@gmail.com

Step 1: Create JWTUtil class that will create and validate jwt token

```java
@Component
public class JwtService {
    public static final String
            SECRET = "5367566B59703373367639792F423F4528482B4D6251655468576D5A71347437";
    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }
    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }
    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }
    private Claims extractAllClaims(String token) {
        return Jwts.parserBuilder().setSigningKey(getSignKey()).build().parseClaimsJws(token).getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }
```

rgupta.mtech@gmail.com

Step 1: Create JWTUtil class that will create and validate jwt token

```
    public String generateToken(String userName) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, userName);
    }

    private String createToken(Map<String, Object> claims, String userName) {
        return Jwts.builder().setClaims(claims).setSubject(userName)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 30))
                .signWith(getSignKey(), SignatureAlgorithm.HS256).compact();
    }

    private Key getSignKey() {
        byte[] keyBytes = Decoders.BASE64.decode(SECRET);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

rgupta.mtech@gmail.com

Step 2/3: Define an rest end point that allow user to get jwt token after authentictation, by pass security for this end point

```java
@RestController
public class HelloController {

    @Autowired
    private JwtService jwtService;

    @Autowired
    private AuthenticationManager authenticationManager;

    //3. craete a endpoint so that user can send his u/p and get token
    @PostMapping(path = "authenticate")
    public String authenticateAndGetToken(@RequestBody AuthRequest authRequest) {
        Authentication authentication
                =authenticationManager.
                authenticate(new UsernamePasswordAuthenticationToken(
                        authRequest.getUsername(),
                        authRequest.getPassword()
                ));

        if(authentication.isAuthenticated()){
            return jwtService.generateToken(authRequest.getUsername());
        }else {
            throw  new UsernameNotFoundException("user is invalid");
        }
```

rgupta.mtech@gmail.com

Step 4: customized spring security filter chain configuration and define AuthenticationManager

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)throws  Exception{
   return  http.csrf().disable() HttpSecurity
           .authorizeHttpRequests() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
           .requestMatchers( ...patterns: "/home","/authenticate").permitAll()
           .and() HttpSecurity
           .authorizeHttpRequests().requestMatchers( ...patterns: "/**") AuthorizeHttpRequestsConfigurer<...>.AuthorizedUrl
           .authenticated() AuthorizeHttpRequestsConfigurer<...>.AuthorizationManagerRequestMatcherRegistry
           .and() HttpSecurity
           .httpBasic() HttpBasicConfigurer<HttpSecurity>
           .and() HttpSecurity
           .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) SessionManagementConfigurer<HttpSecurity>
           .and() HttpSecurity
           .authenticationProvider(getAuthentication())
           .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)
           .build();
}

//5. get the AM
@Bean
public AuthenticationManager authenticationManager(AuthenticationConfiguration config)throws Exception {
    return config.getAuthenticationManager();
}
```

rgupta.mtech@gmail.com

Step 5: Configure OncePerRequestFilter before UsernamePasswordAuthenticationFilter

```java
@Service
public class JwtAuthFilter extends OncePerRequestFilter {
    @Autowired
    private JwtService jwtService;
    @Autowired
    private UserDetailsService userDetailsService;
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        String authHeader= request.getHeader( name: "Authorization");
        String token=null;
        String username=null;
        if(authHeader!=null && authHeader.startsWith("Bearer ")){
        token=authHeader.substring( beginIndex: 7);
        username=jwtService.extractUsername(token);
        }
        if(username!=null && SecurityContextHolder.getContext().getAuthentication()==null){
            UserDetails userDetails=userDetailsService.loadUserByUsername(username);
            //username is correct , and we are going to get UNAuthToeken and put that in SecurityContextHolder ....
            if(jwtService.validateToken(token, userDetails)){

                UsernamePasswordAuthenticationToken authToken=
                     new  UsernamePasswordAuthenticationToken(userDetails,  credentials: null, userDetails.getAuthorities());
                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);

                }
            }
            filterChain.doFilter(request, response);
        }
    }
```

rgupta.mtech@gmail.com

Module 11: Spring Reactive Streams

rgupta.mtech@gmail.com

Why Reactive Programming ?

Evolution of Programming

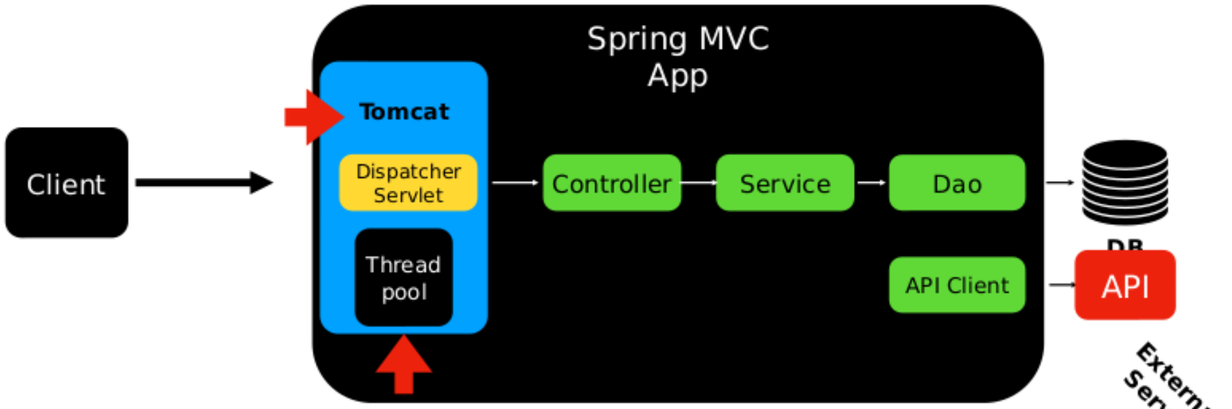
Past (10 -15) years ago	Current
• Monolith Applications	• Microservices Applications
• Deployed in Application Server	• Deployed in Cloud Environments
• Does not embrace distributed systems	• Embrace Distributed Systems

Expectations of the Application

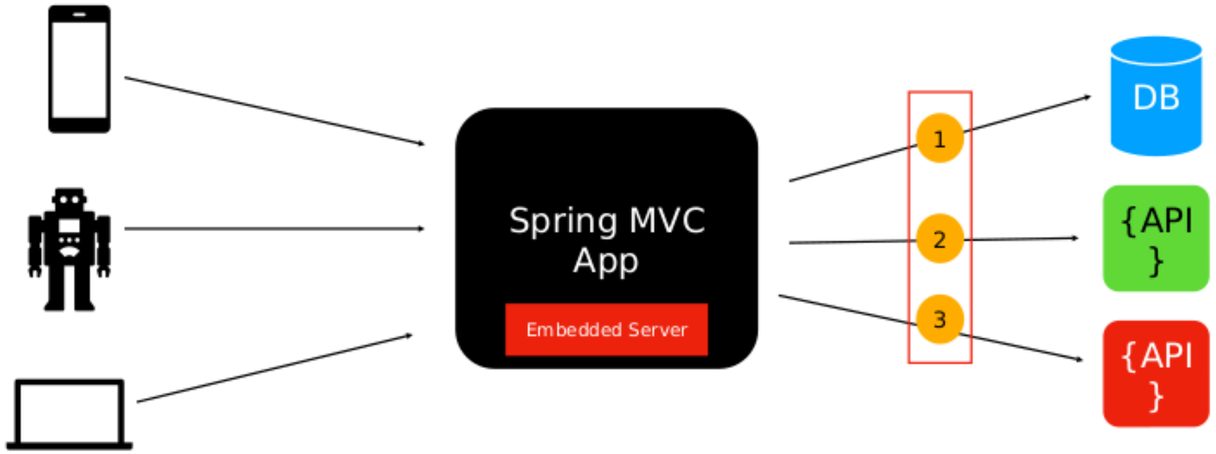
Response times are expected in milliseconds
No Downtime is expected
Scale up automatically based on the load

rgupta.mtech@gmail.com

Restful API using Spring Boot/MVC



- Concurrency is **Thread Per Request** model
- This style of building APIs are called **Blocking APIs**
- Wont scale for today's application needs



Latency = Summation of (DB + API + API) response times

rgupta.mtech@gmail.com

Spring MVC Limitations

Thread pool size of Embedded tomcat in Spring MVC's is 200

- Can we increase the thread pool size based on the need ?
- Yes, only to a certain limit.
- Let's say you have a use case to support 10000 concurrent users.
- Can we create a thread pool of size 10000 Threads ? NO

Thread is an expensive resource

- It can easily take up to 1MB of heap space
- More threads means more memory consumption by the thread itself
- Less heap space for actually processing the reques

Lets explore the asynchrony options in Java

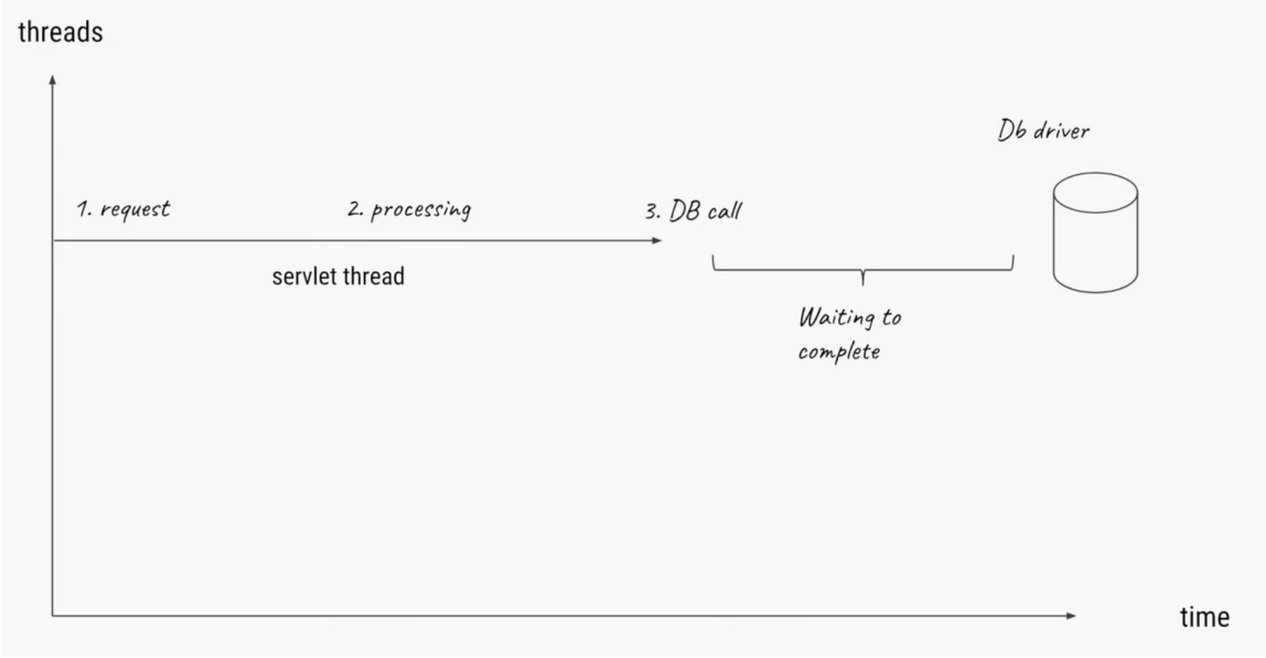
- Callbacks
- Futures

Reactive programming to the rescue

rgupta.mtech@gmail.com

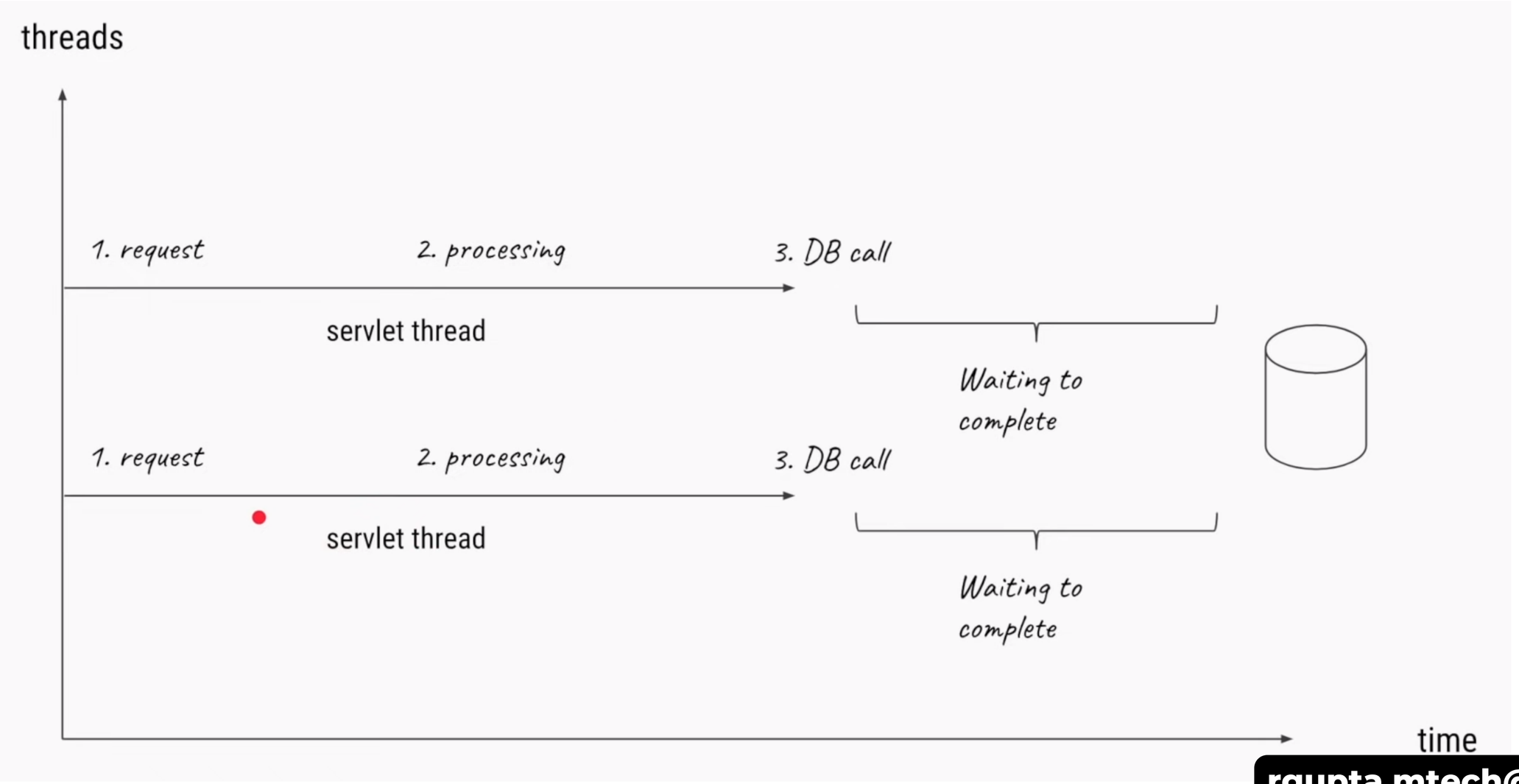
Traditional Spring Boot application

```java
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public User getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
```



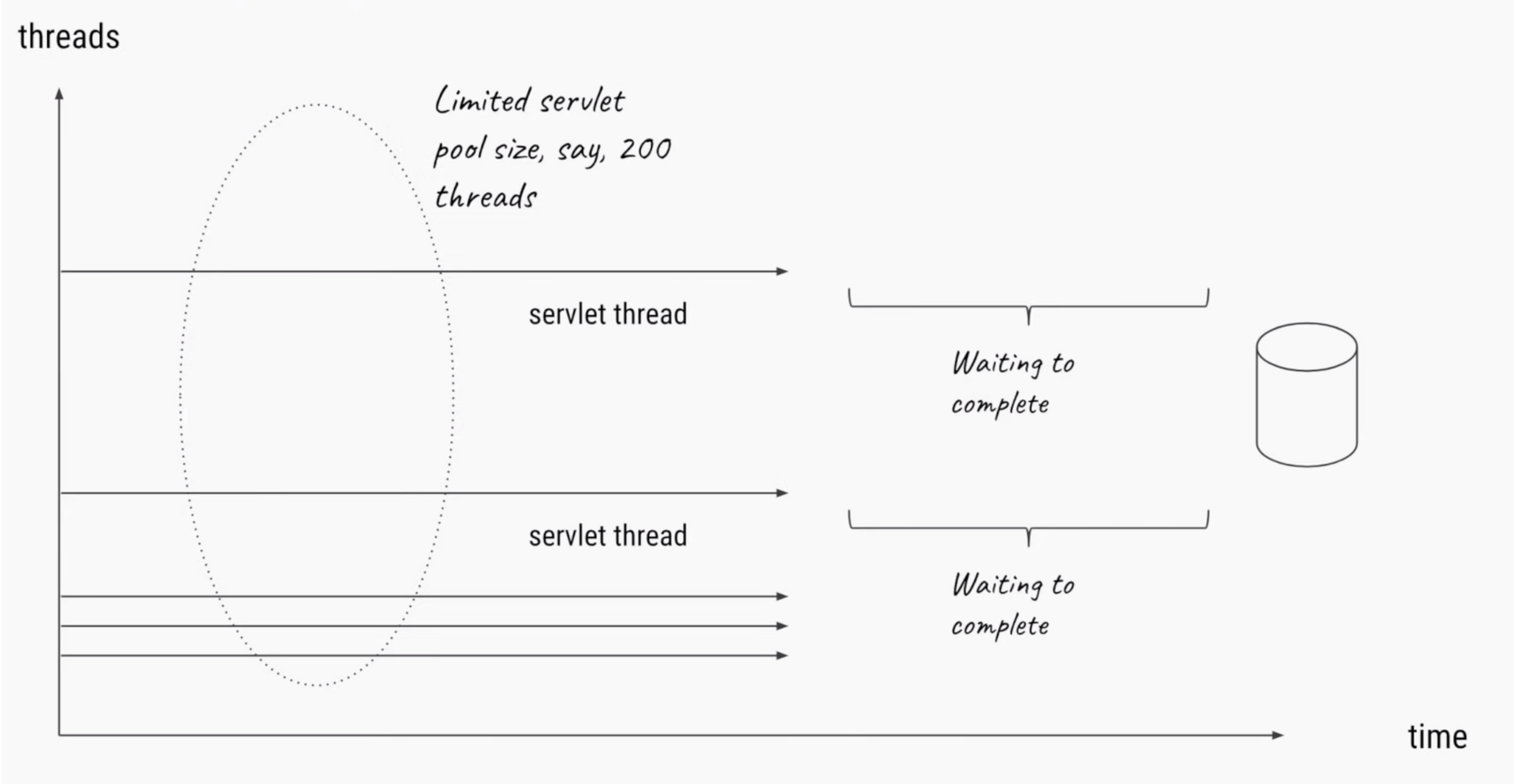
rgupta.mtech@gmail.com

Traditional Spring Boot application



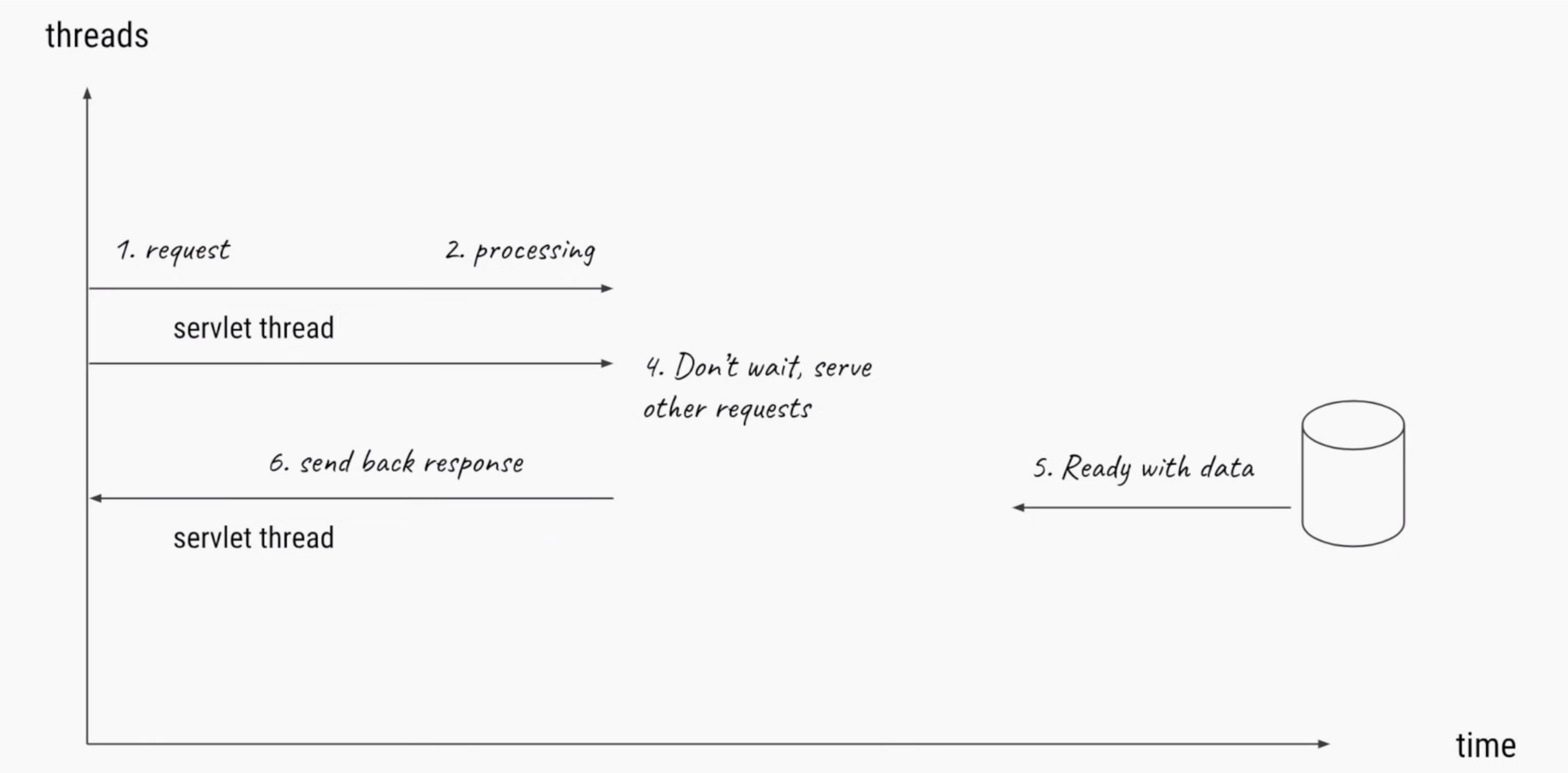
rgupta.mtech@gmail.com

Traditional Spring Boot application



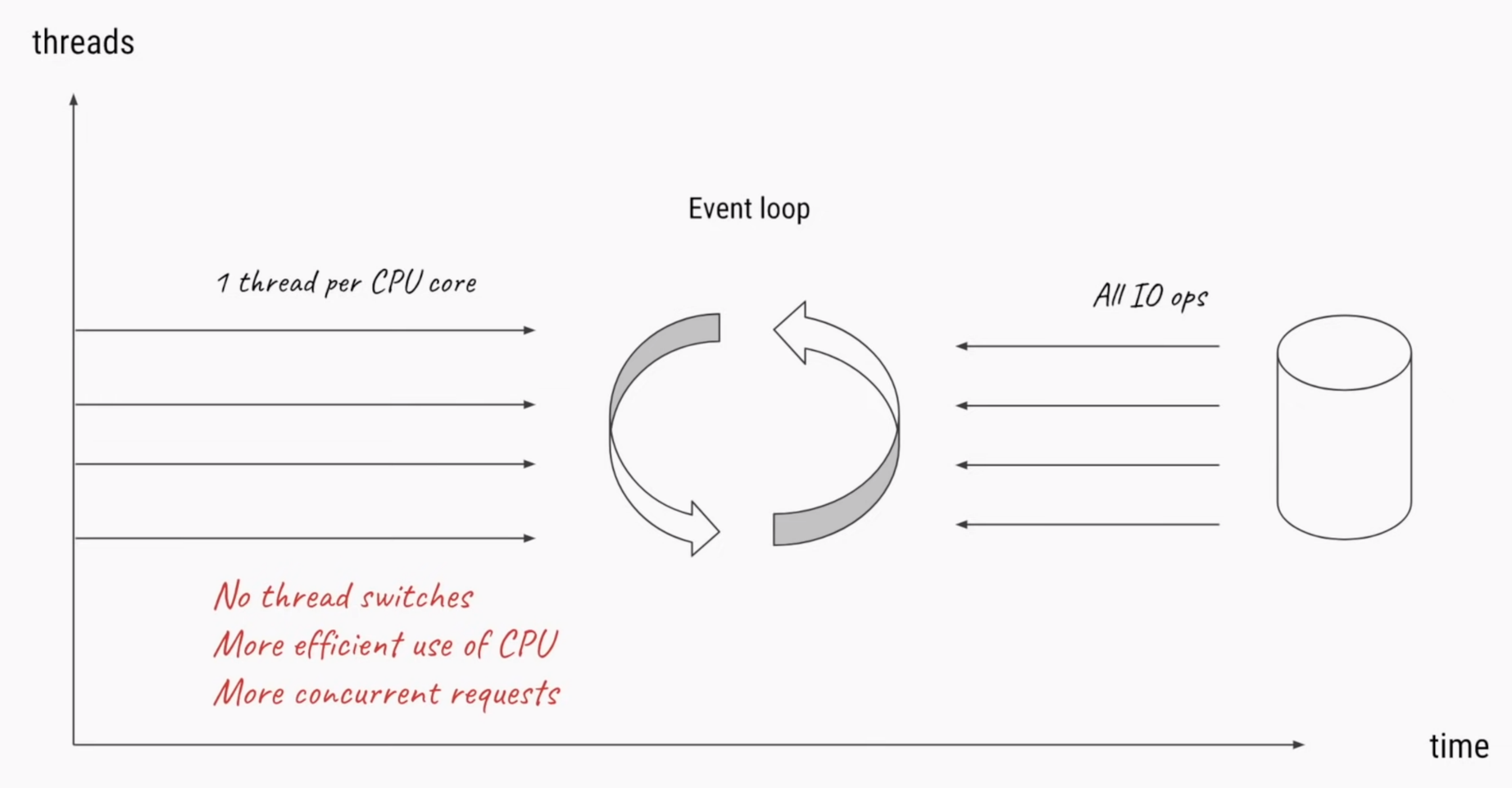
rgupta.mtech@gmail.com

What we want



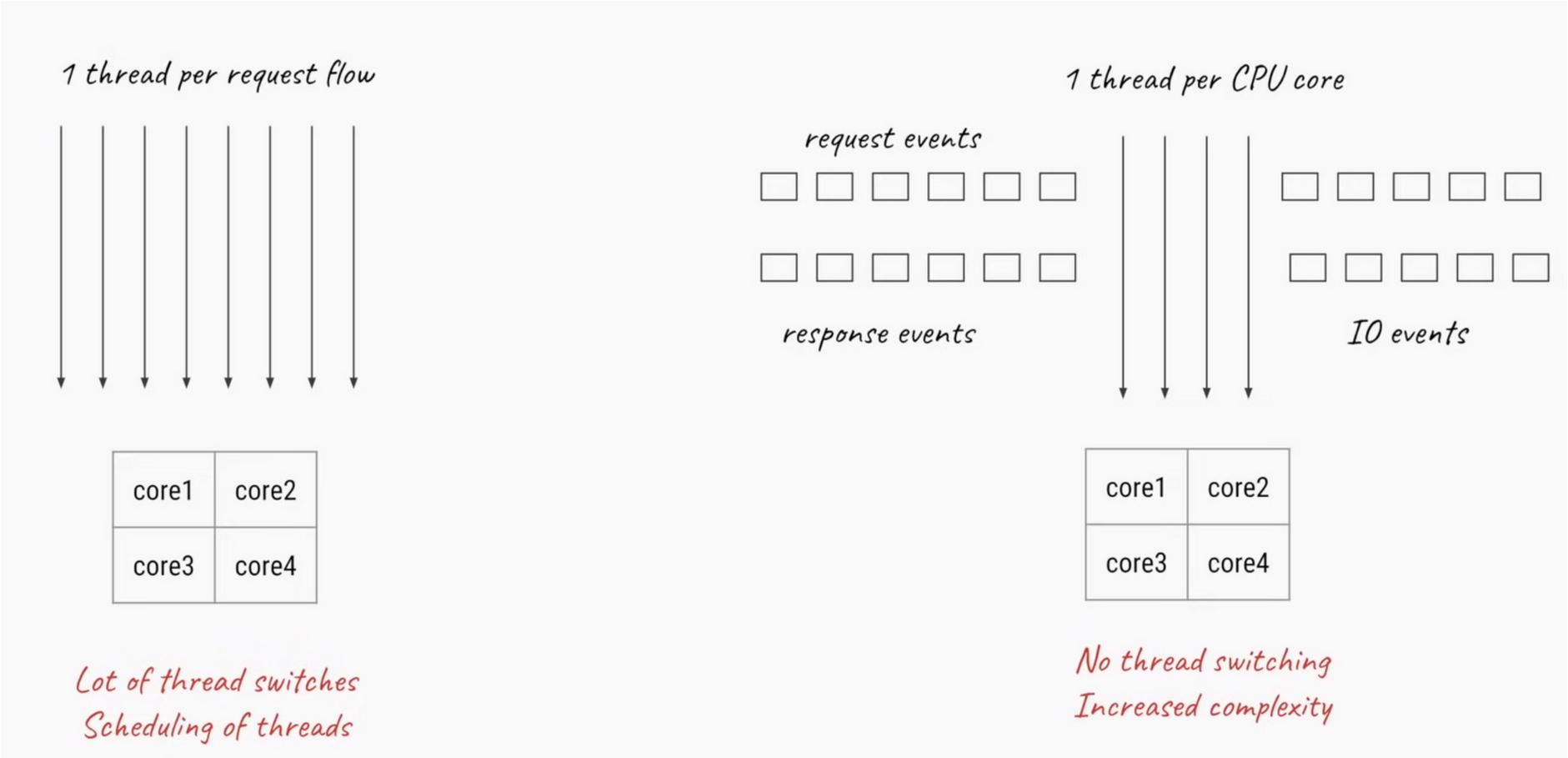
rgupta.mtech@gmail.com

Event Loop concept



rgupta.mtech@gmail.com

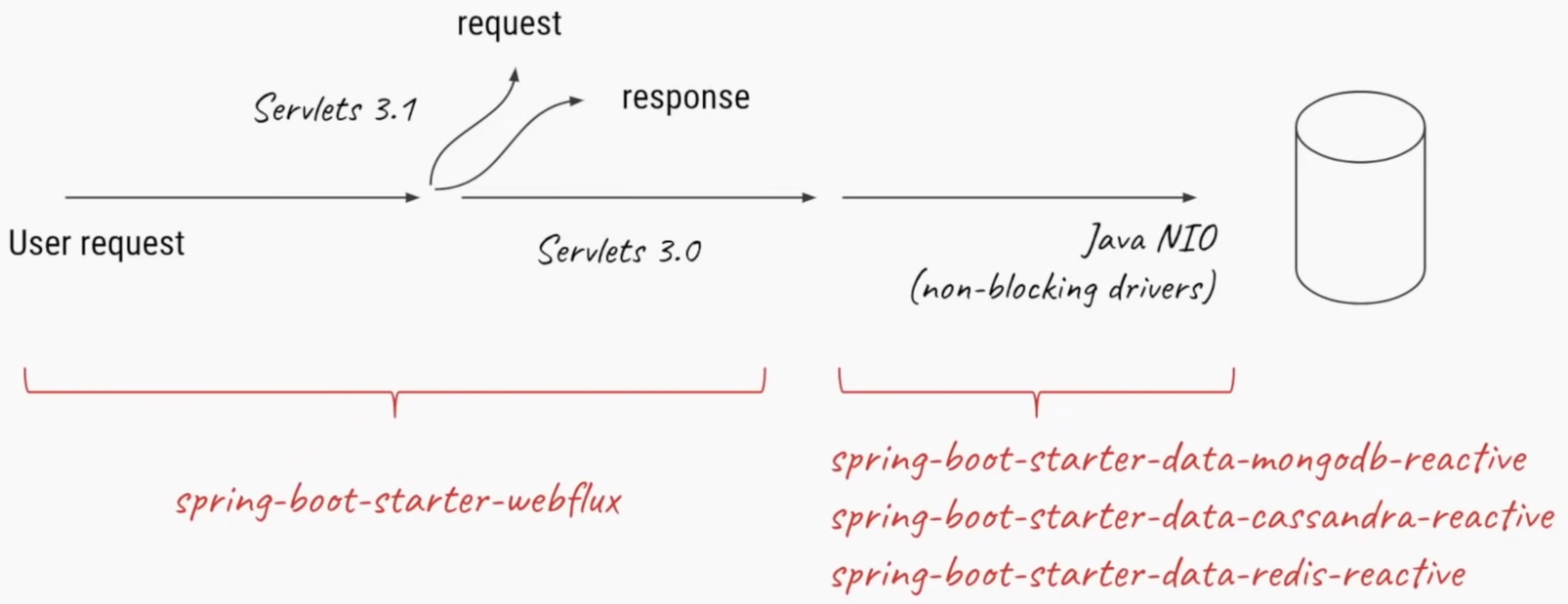
Traditional vs Event loop



rgupta.mtech@gmail.com

End to end Reactive

- Servlets 3.0 and 3.1 (Async and NIO)
- Java NIO (File, Network IO)
- MongoDB, Cassandra, Redis (more to come)



rgupta.mtech@gmail.com

End to end Reactive

```java
@GetMapping("/get/user/{id}")
public User getUser(@PathVariable String id) {
    return userRepository.findUser(id);
}
```

Blocking call

```java
@GetMapping("/get/user/{id}")
public Future<User> getUser(@PathVariable String id) {
    return userRepository.findUser(id);
}
```

```java
@GetMapping("/get/user/{id}")
public CompletableFuture<User> getUser(@PathVariable String id) {
    return userRepository.findUser(id);
}
```

```java
@GetMapping("/get/user/{id}")
public Mono<User> getUser(@PathVariable String id) {
    return userRepository.findUser(id);
}
```

Mono = 0..1 elements

Flux = 0..N elements

rgupta.mtech@gmail.com

Reactive Controller

```java
@RestController
public class PricingController {

    @Autowired
    private PricesRepository pricesRepository;

    @GetMapping("/get/prices/{id}")
    public Mono<Price> getPrice(@PathVariable String id) {
        return pricesRepository.findById(id);
    }

    @GetMapping("/get/prices/all")
    public Flux<Price> getAllPrice() {
        return pricesRepository.findAll();
    }

    @GetMapping(value = "/get/prices/live", produces = "text/event-stream")
    public Flux<Price> getLivePrice() {
        return pricesRepository.findAll();
    }
}
```

Reactive DB driver

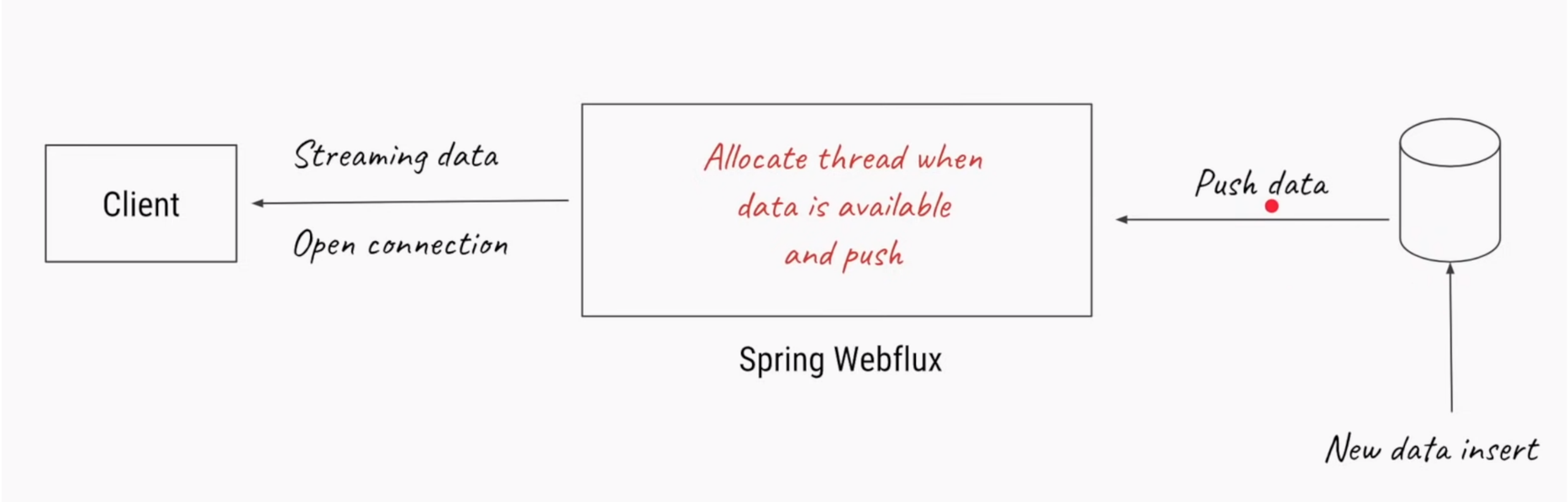
Single value

List of values

Live list of values

rgupta.mtech@gmail.com

Continus data response



rgupta.mtech@gmail.com

Continus data Request

```java
@RestController
public class PricingRequestsController {

    @Autowired
    private PricesRepository pricesRepository; Reactive DB driver

    @PostMapping("/save/price")
    public void save(@RequestBody Mono<Price> price) {
        price.subscribe(pricesRepository::save);
    }

    @PostMapping("/save/price/all")
    public void saveAll(@RequestBody Flux<Price> prices) {
        pricesRepository.saveAll(prices);
    }

    @PostMapping(value = "/save/price", consumes = "application/stream+json")
    public void getPrice(@RequestBody Flux<Price> prices) {
        pricesRepository.saveAll(prices);
    }
```

Single value

List of values

Live list of values

rgupta.mtech@gmail.com

Advantage and consideration

- Scalability
  - IO bound operations (microservices)
  - Efficient CPU utilization
  - Data locality & Less-context switches
- Streaming use-cases (Live source of data)
- Backpressure

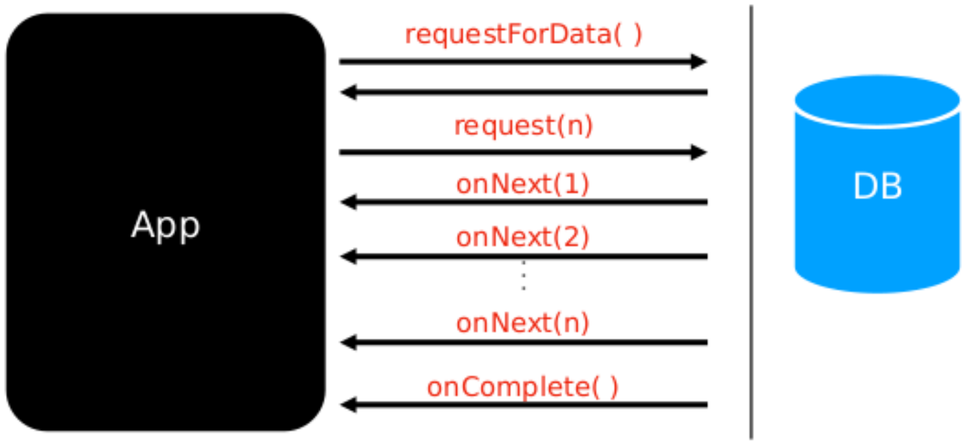
- Different programming model
- End-to-end reactive required
- Not too useful for CPU bound flows
- Hard to debug (stack trace)
- Hard to write tests

rgupta.mtech@gmail.com

What is Reactive Programming ?

Reactive Programming is a new programming paradigm

- **Asynchronous and non blocking**
- **Data flows as an Event/Message driven stream**
- **Functional Style Code**
- **BackPressure on Data Streams**

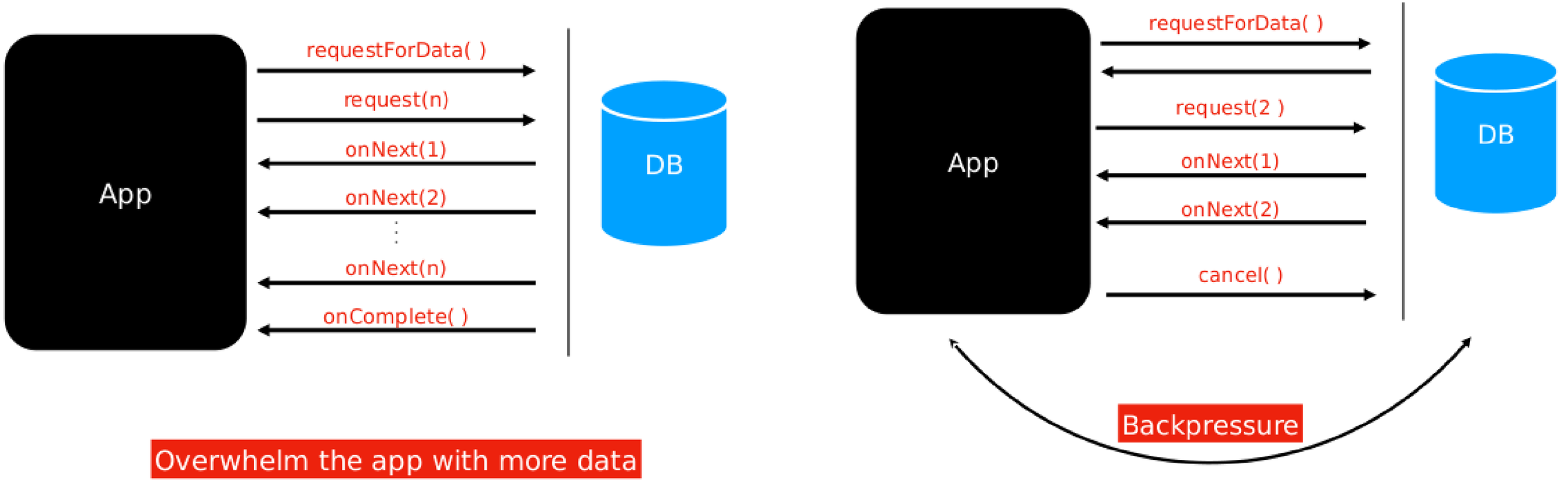


- This is not a blocking call anymore
- Calling thread is released to do useful work

Push Based data streams model

rgupta.mtech@gmail.com

Backpressure

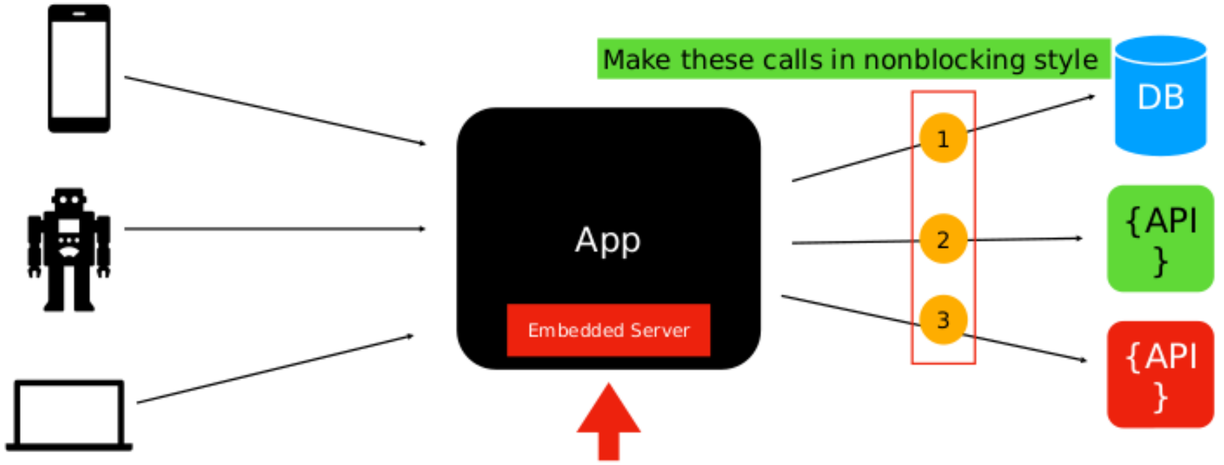


Push-based data flow model

rgupta.mtech@gmail.com

When to use Reactive Programming

Use Reactive Programming when there is need to build and support app that can handle high load



- Handle request using non blocking style
  - Netty is a non blocking Server uses Event Loop Model
- Using Project Reactor for writing non blocking code
- Spring WebFlux uses the Netty and Project Reactor for building non blocking or reactive APIs

Reactive App Architecture

rgupta.mtech@gmail.com

Reactive Streams

Reactive Streams are the foundation for Reactive programming

<u>Reactive Streams Specification</u>

- Publisher
- Subscriber
- Subscription
- Processor

rgupta.mtech@gmail.com

Publisher

```
public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);
}
```
1

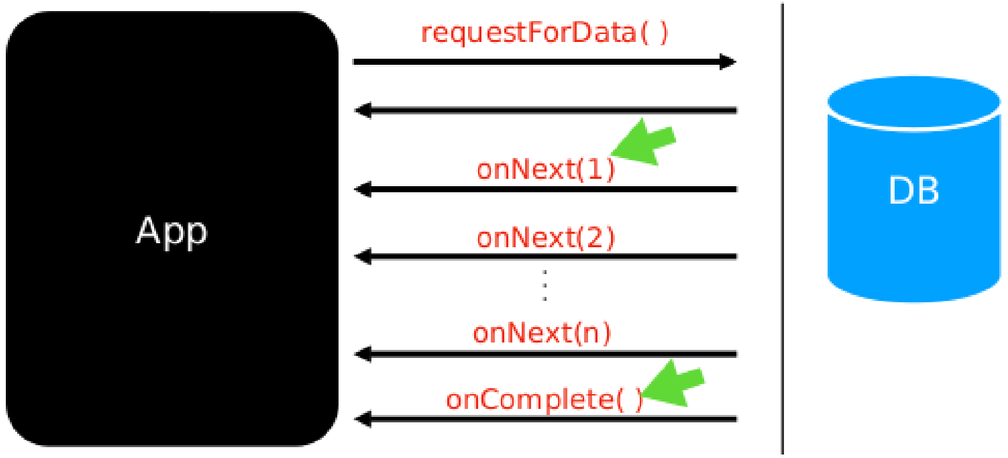
- Publisher represents the DataSource
  - Database
  - RemoteService etc.,

rgupta.mtech@gmail.com

Subscriber

```
public interface Subscriber<T> {

    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

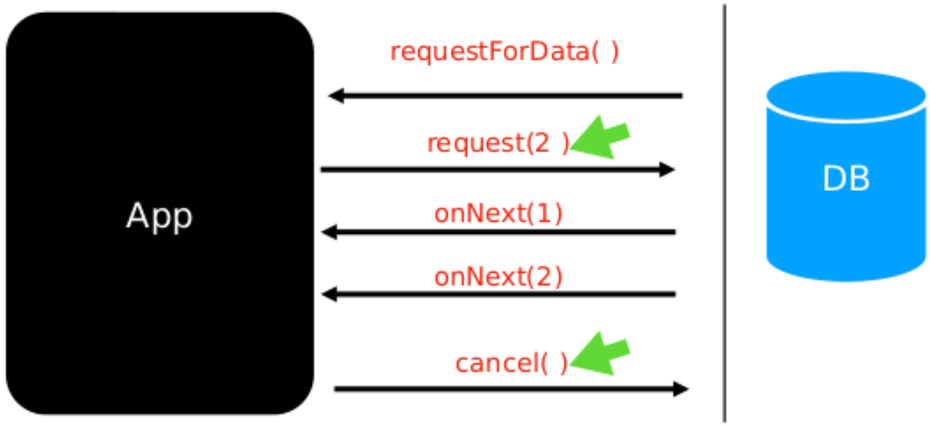
1 2 3 4



rgupta.mtech@gmail.com

Subscription

```
public interface Subscription {

    public void request(long n);

    public void cancel();
}
```



Subscription is the one which connects the app and datasource

rgupta.mtech@gmail.com

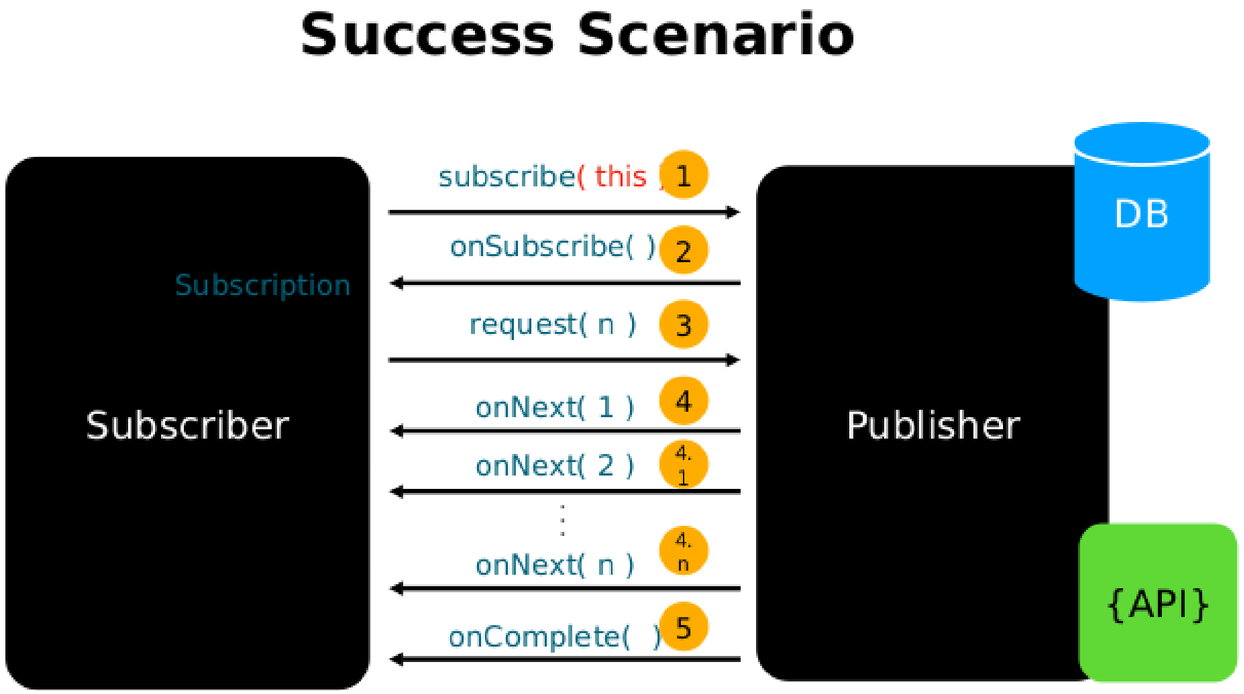
Processor

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {

}
```

- Processor extends Subscriber and Publisher
  - Processor can behave as a Subscriber and Publisher
  - Not really used this on a day to day basis

rgupta.mtech@gmail.com

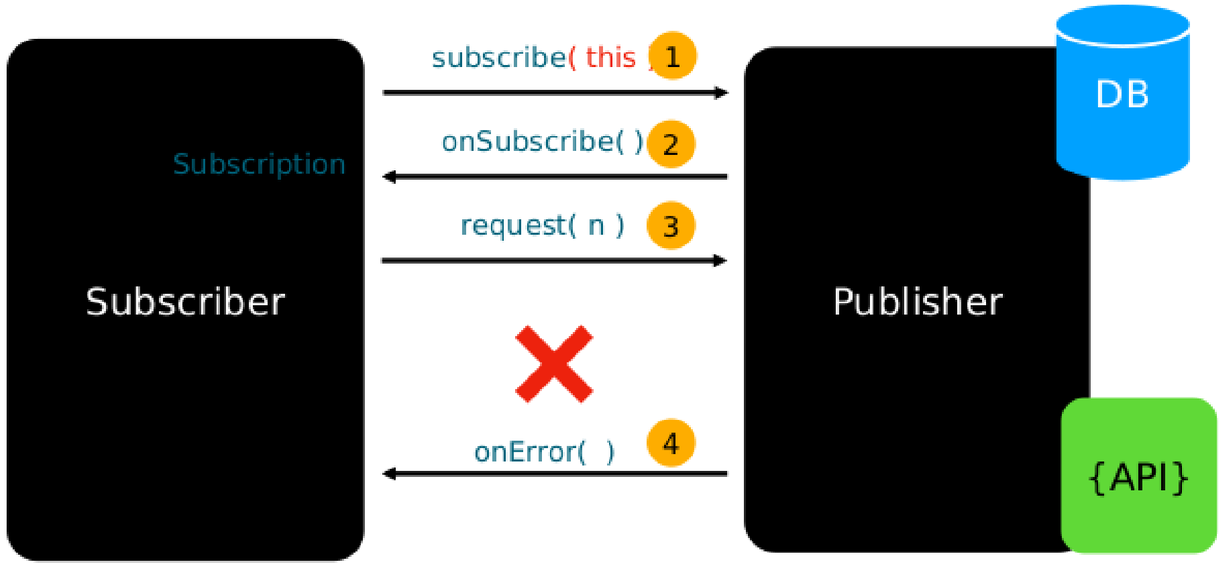
Reactive Streams - How it works together ?



rgupta.mtech@gmail.com

Reactive Streams - How it works together ?

Error/Exception Scenario



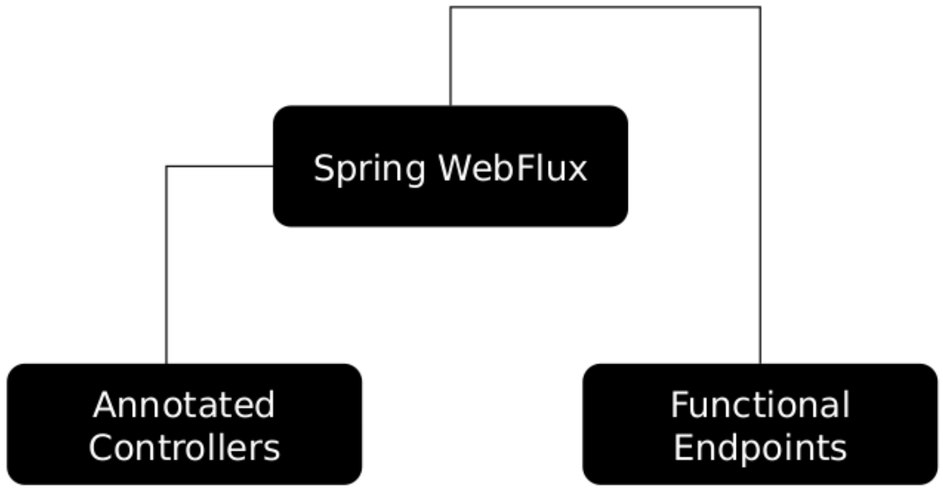
- Exceptions are treated like the data
- The Reactive Stream is dead when an exception is thrown

rgupta.mtech@gmail.com

NonBlocking or Reactive RestFul API

- A Non-Blocking or Reactive RestFul API has the behavior of providing end to end non-blocking communication between the client and service
- Non-Blocking or Reactive == Not Blocking the thread
- Thread involved in handling the httprequest and httpresponse is not blocked at all
- Spring WebFlux is a module that's going to help us in achieving the Non-Blocking or Reactive behavior

Spring WebFlux

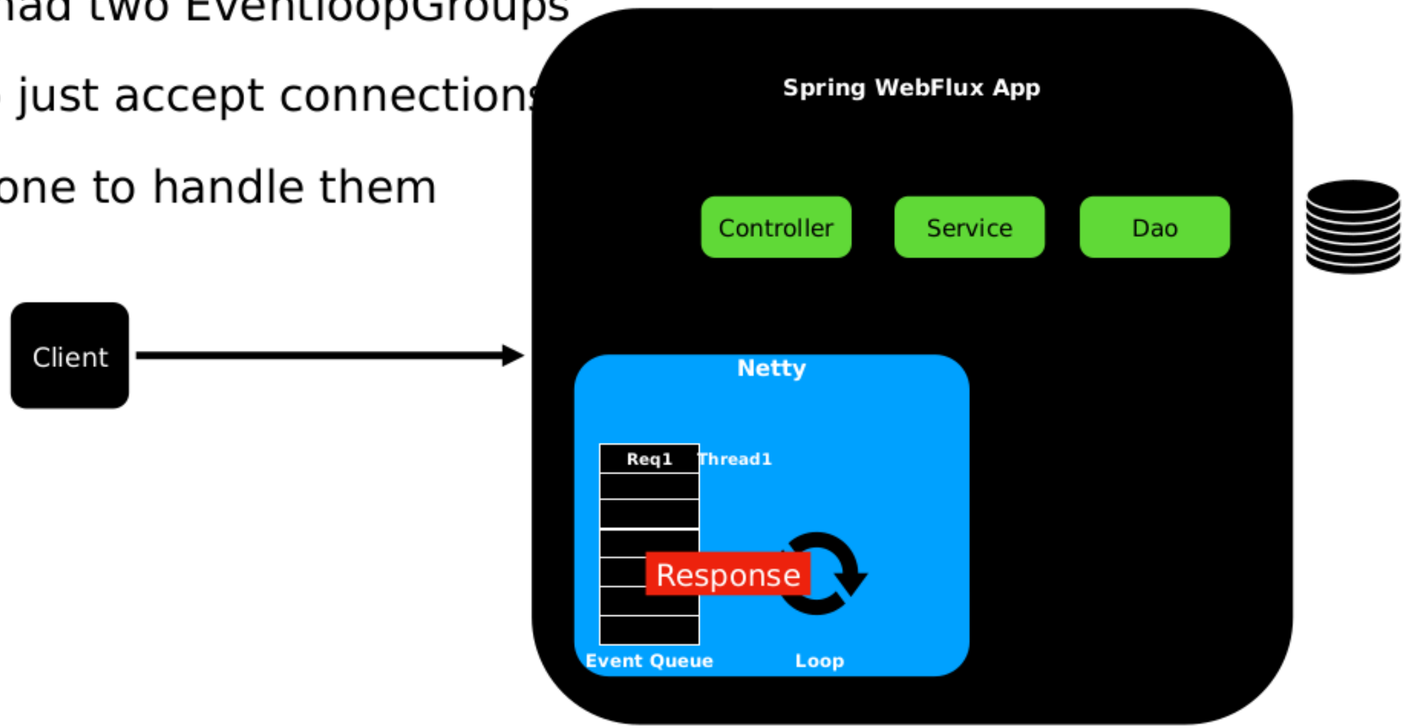


rgupta.mtech@gmail.com

How Netty Works with Spring WebFlux ?

How Netty handles the request ?

- Netty had two EventloopGroups
- One to just accept connections
- Other one to handle them



rgupta.mtech@gmail.com

Module 10: Spring REST Projects

rgupta.mtech@gmail.com

Spring Boot REST Projects

- Spring REST Bank Application
- Spring REST BlogPost Application
- Spring REST EventManagement Application
- Spring REST food delivery Application
- Spring REST ProductStore Application

rgupta.mtech@gmail.com