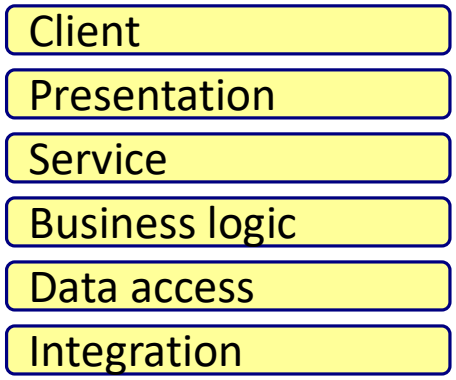
Lesson 2

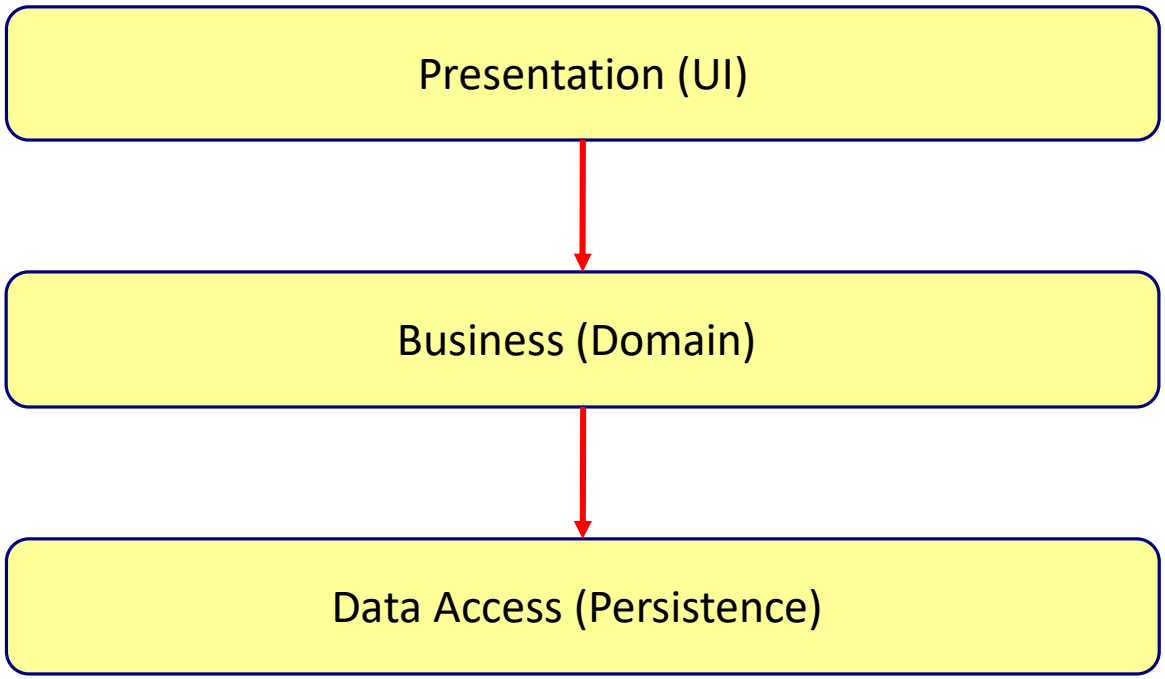
LAYERING & SPRING BOOT

Layering

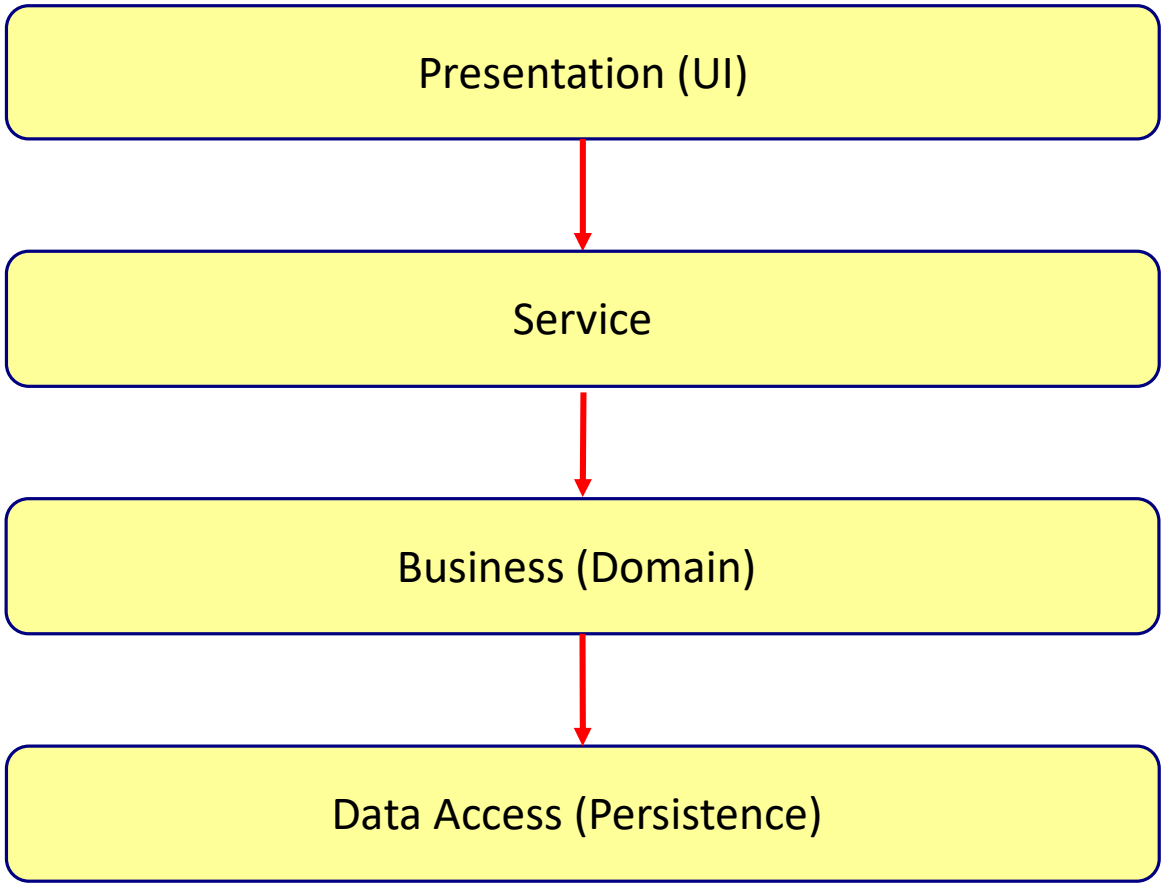
- Separation of concern
- Layers are independent
- Layers can be distributed
- Layers use different techniques



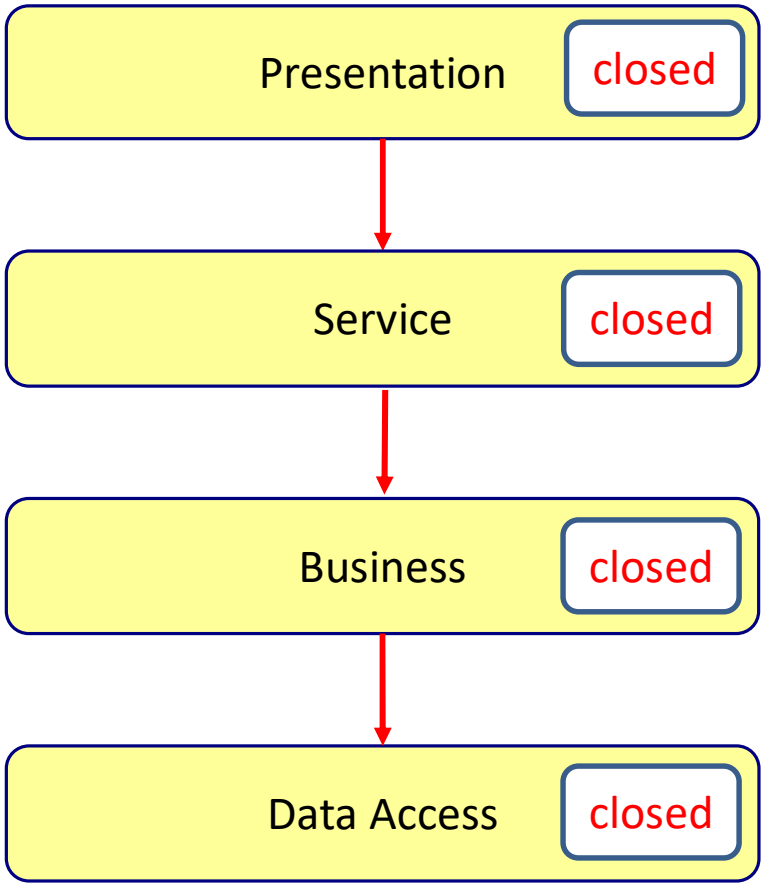
3 layered architecture

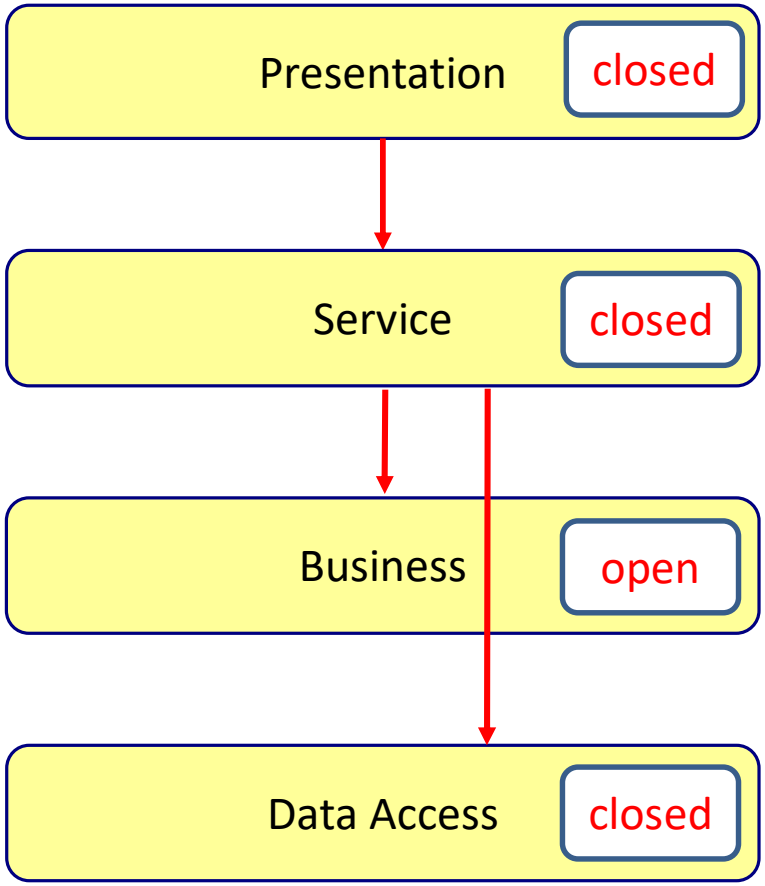


4 layered architecture

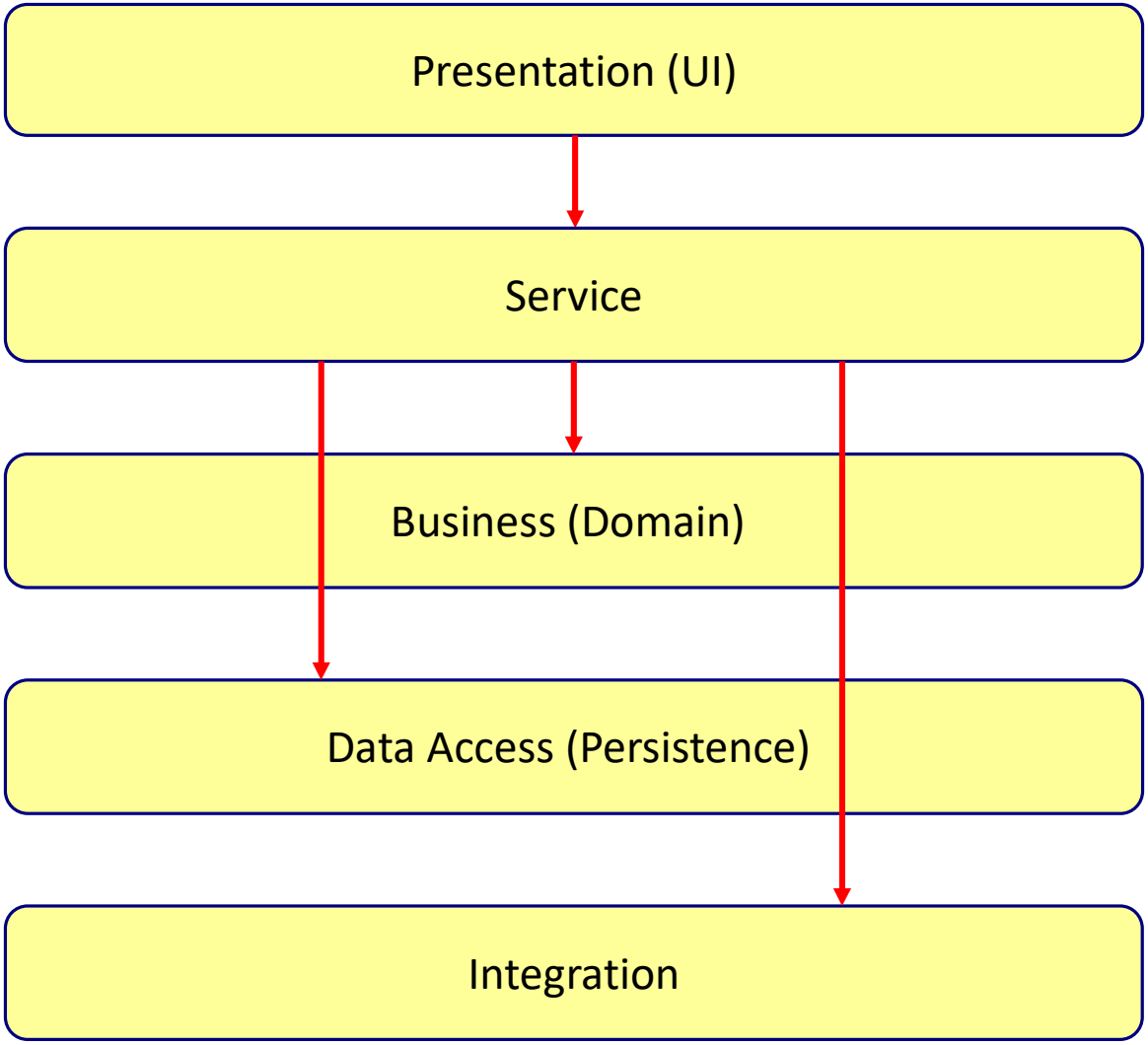


Open and closed layers

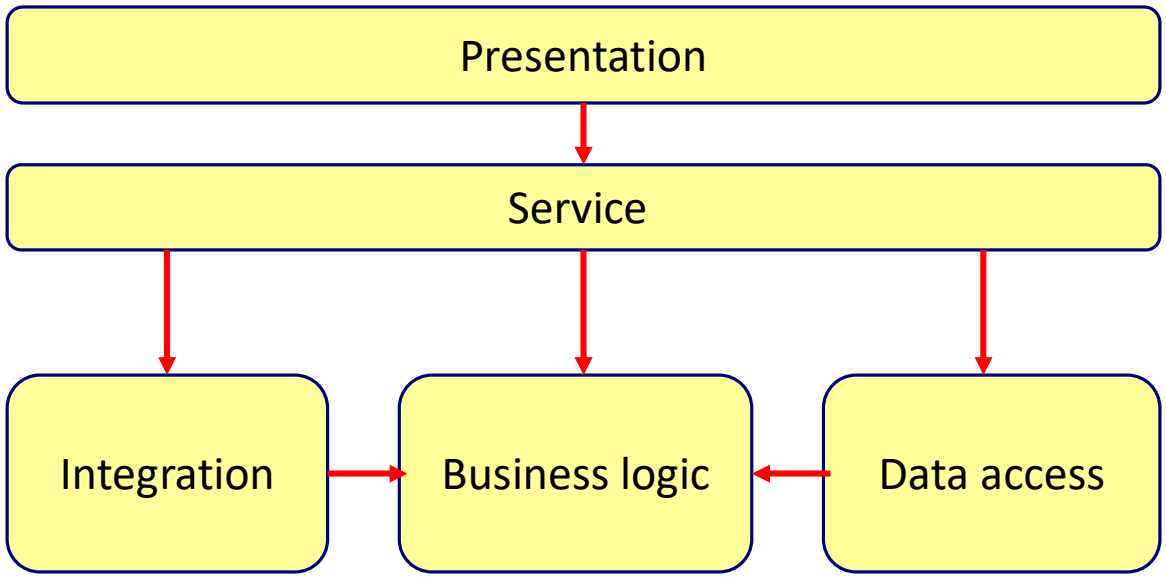




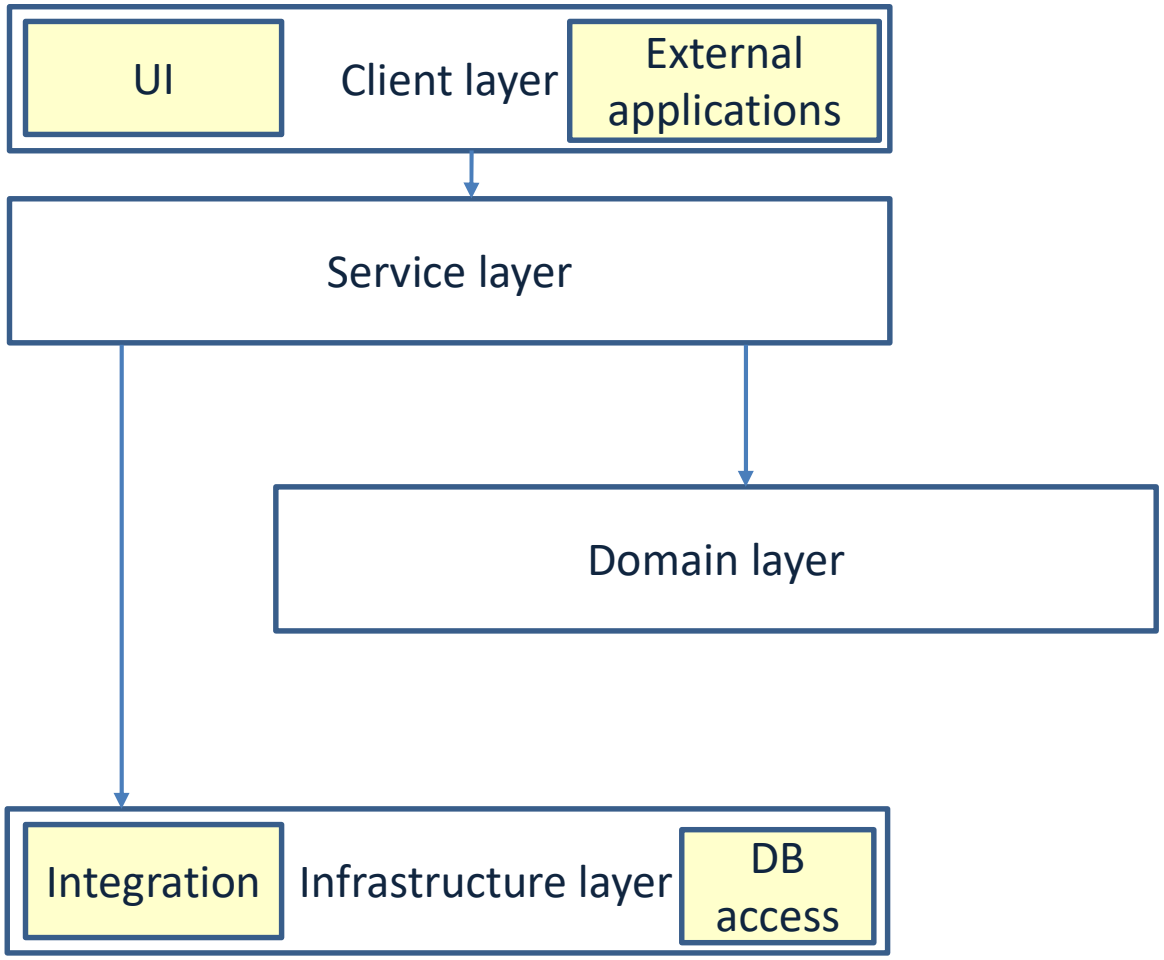
5 layered architecture



Layered architecture



Layered architecture

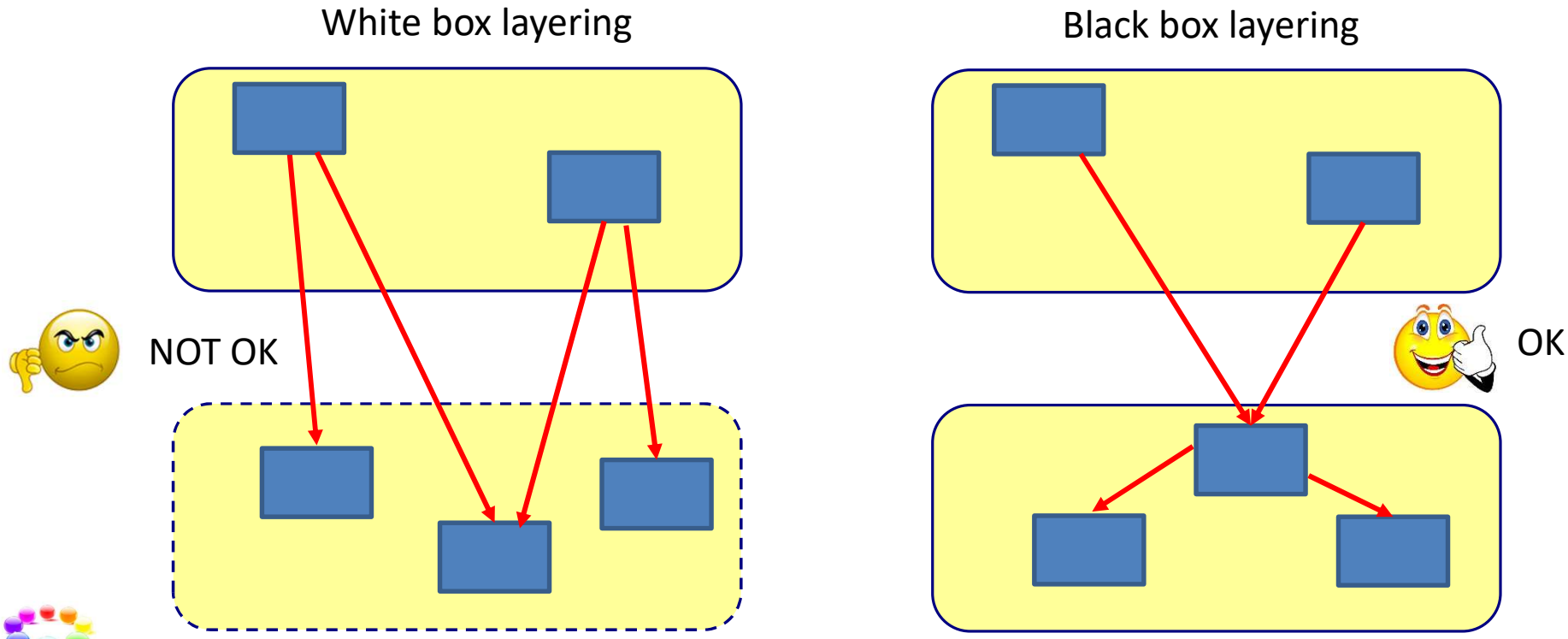


Layering

- Benefits
  - Layers can be distributed
  - Separation if concern
    - Different skills required in each layer
    - Easy to modify
    - Easy to test
- Drawbacks
  - Development effort can increase
  - Performance can become an issue

Layering anti patterns

- Too much layers
- No logic in layers
- No encapsulation of layers



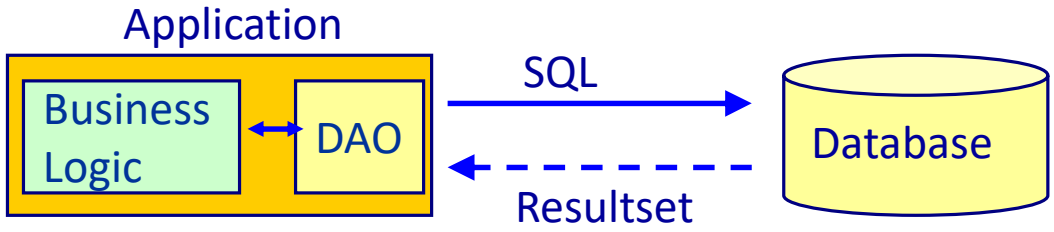
Main point

- An enterprise back-end system is typically divided in different layers. *Life is found in layers*.

ARCHITECTURE PATTERNS

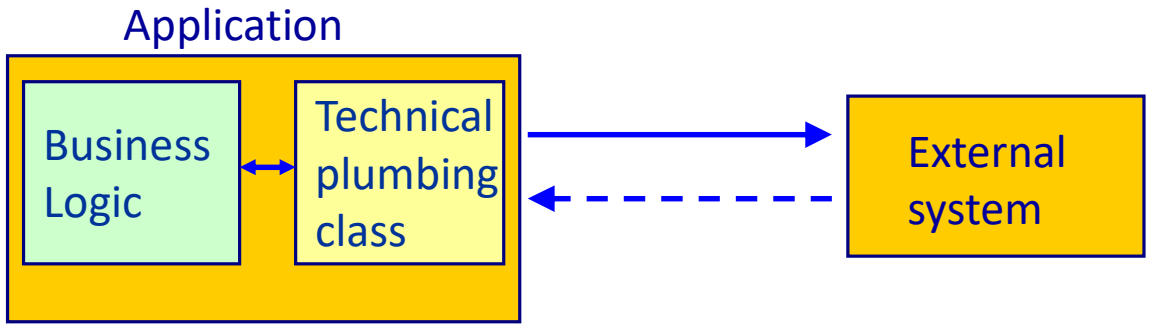
Data Access Object (DAO)

- Object that knows how to access the database
- Contains all database related logic
- Also called repository

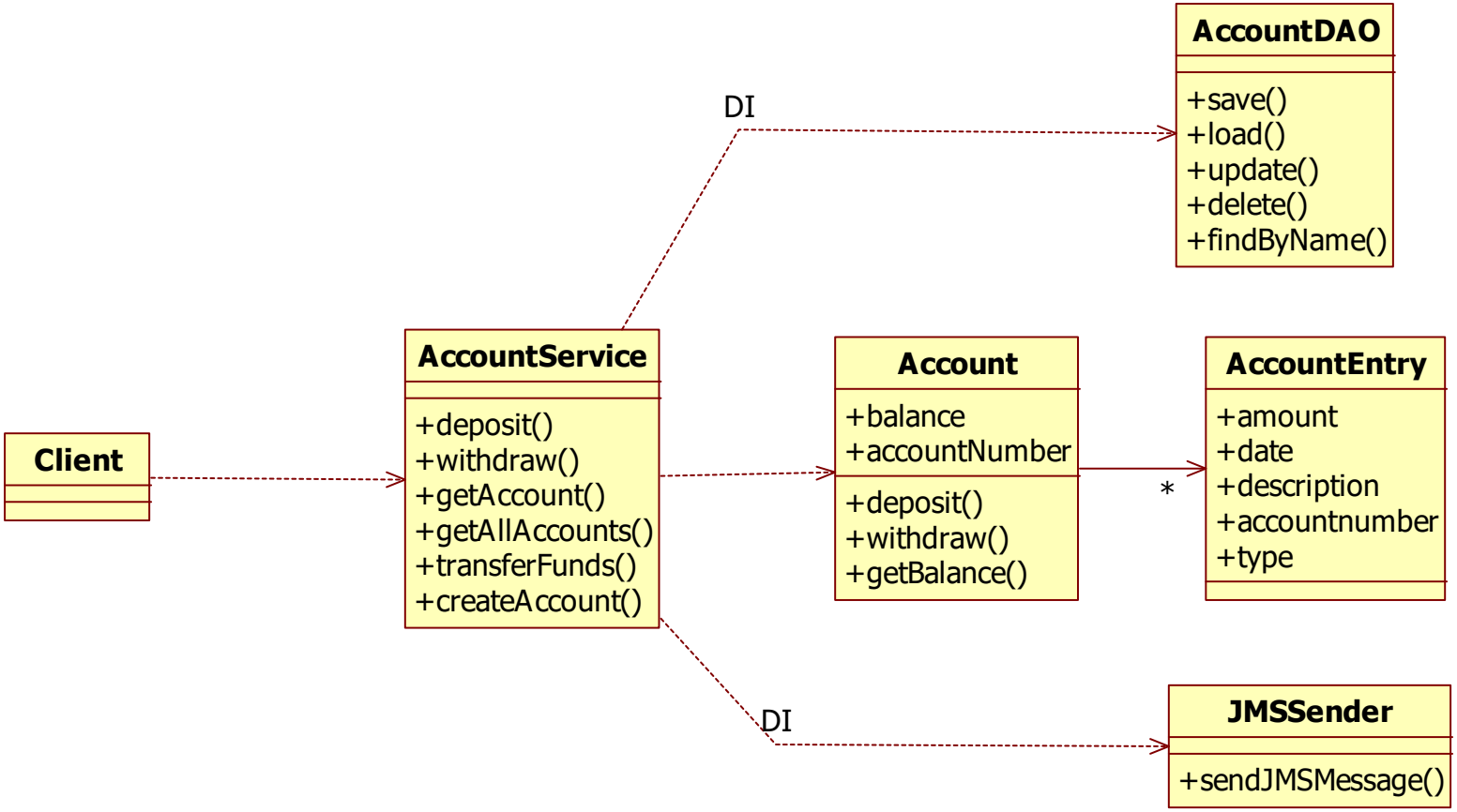


Technical plumbing classes

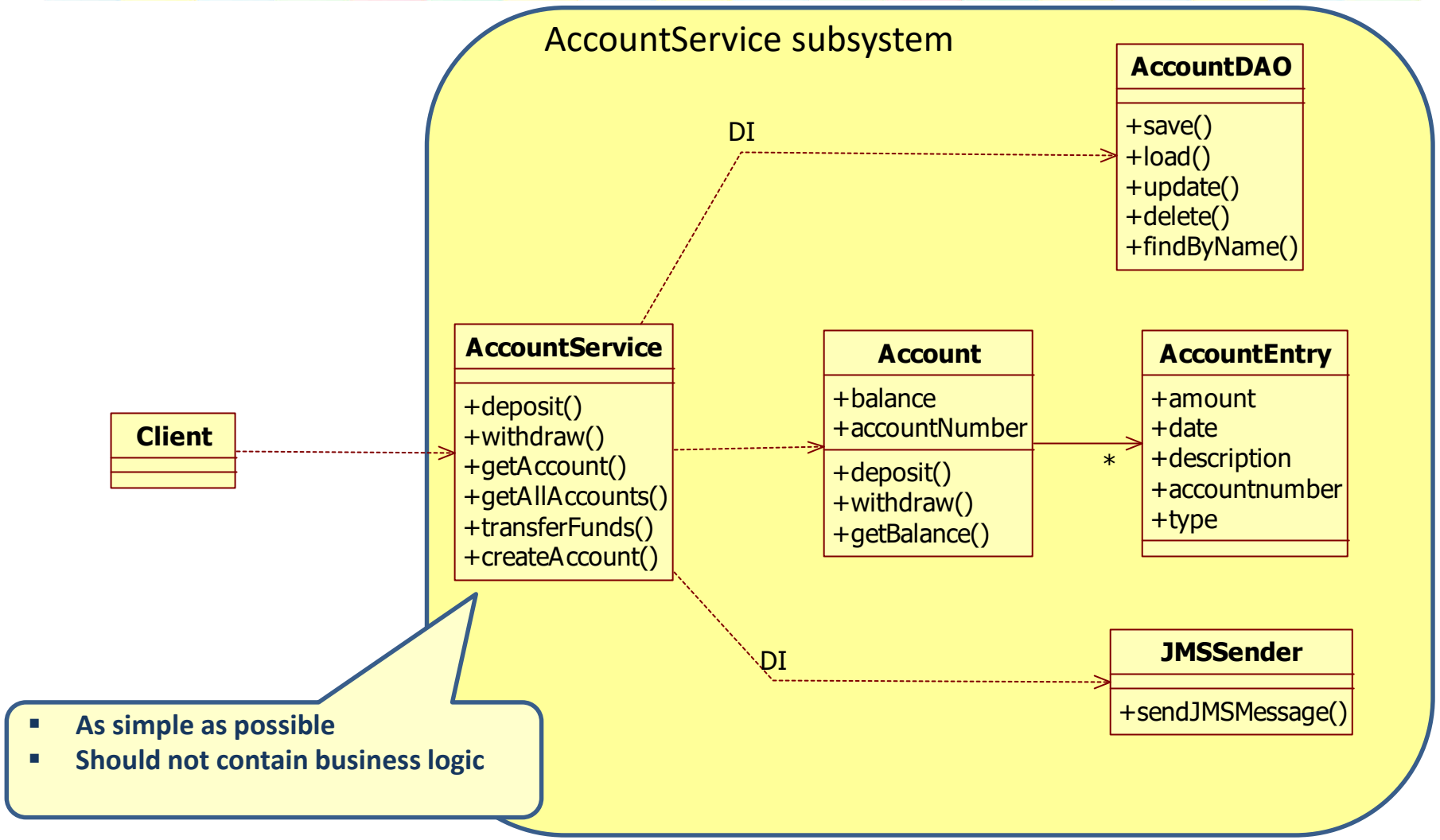
- Single responsibility
  - Web service
  - Remote calls
  - Messaging
  - Email
  - Logging



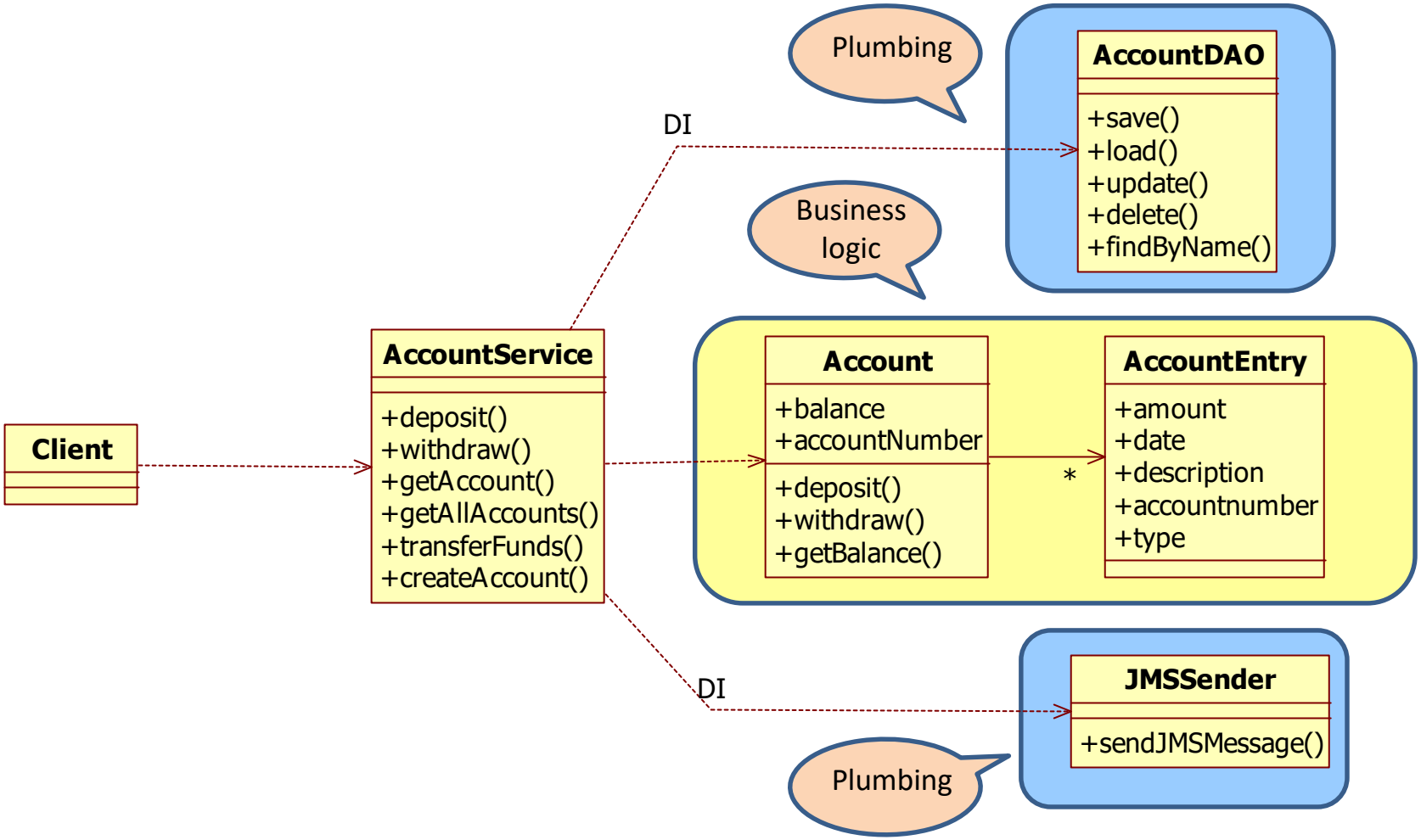
Service Object



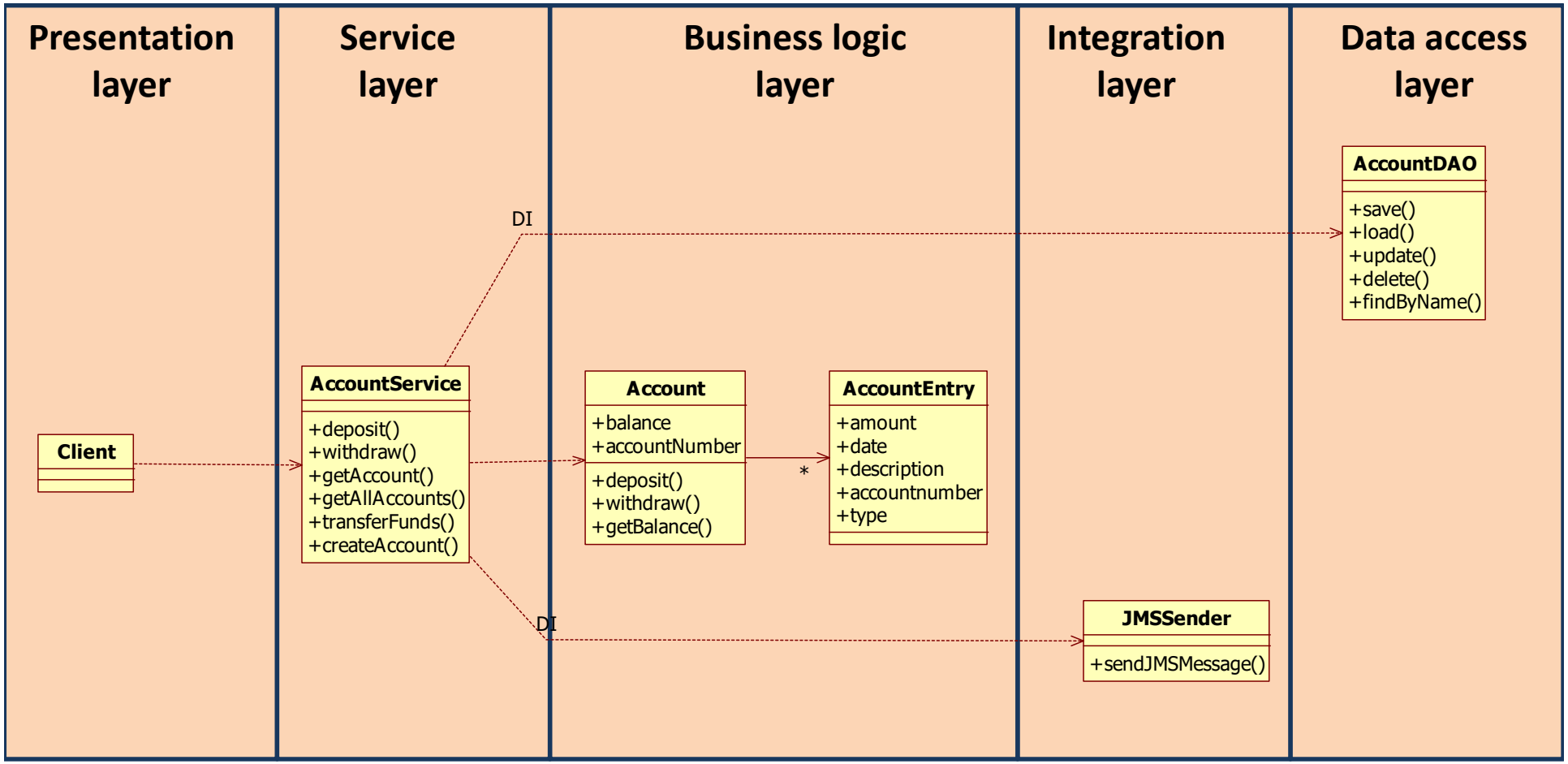
Entry of a complex subsystem



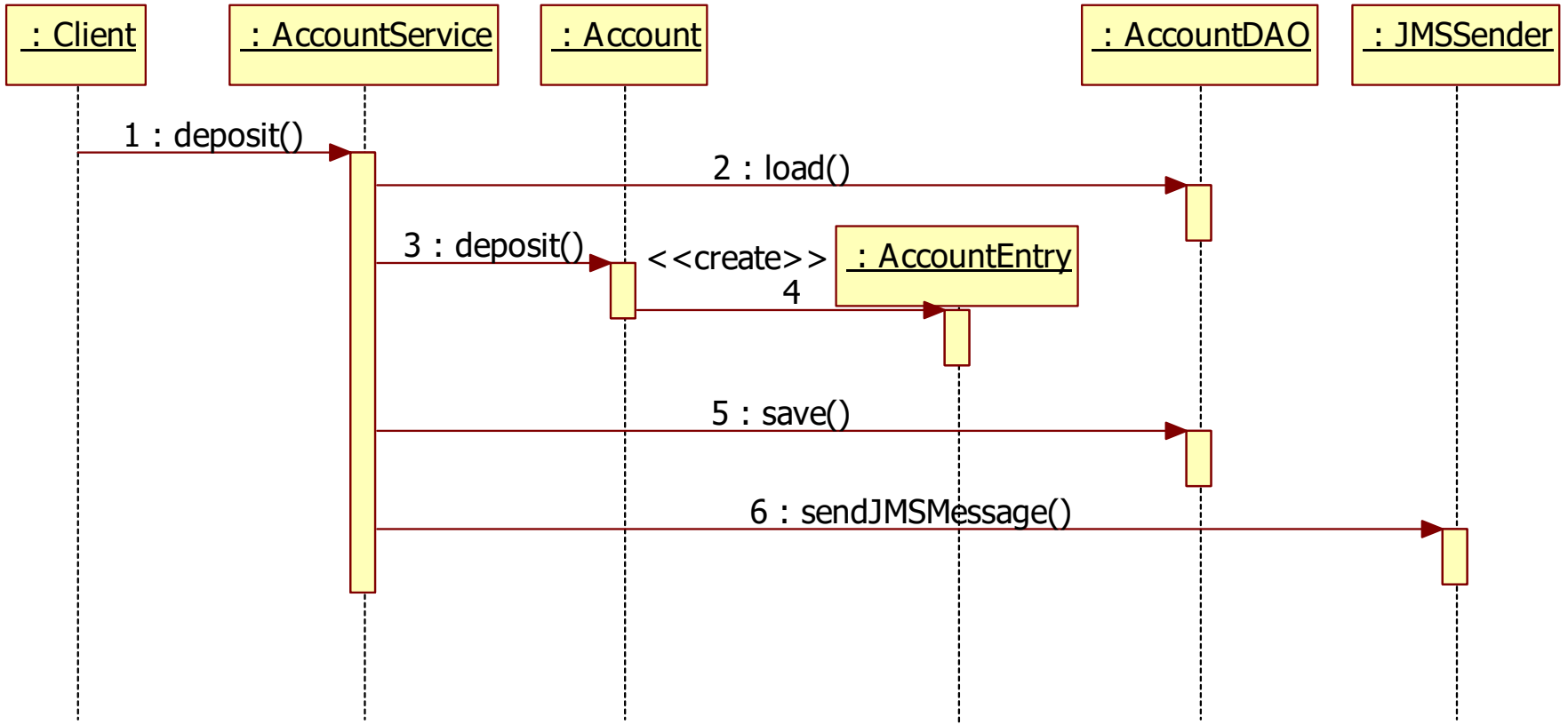
Separation of concern



Application layers

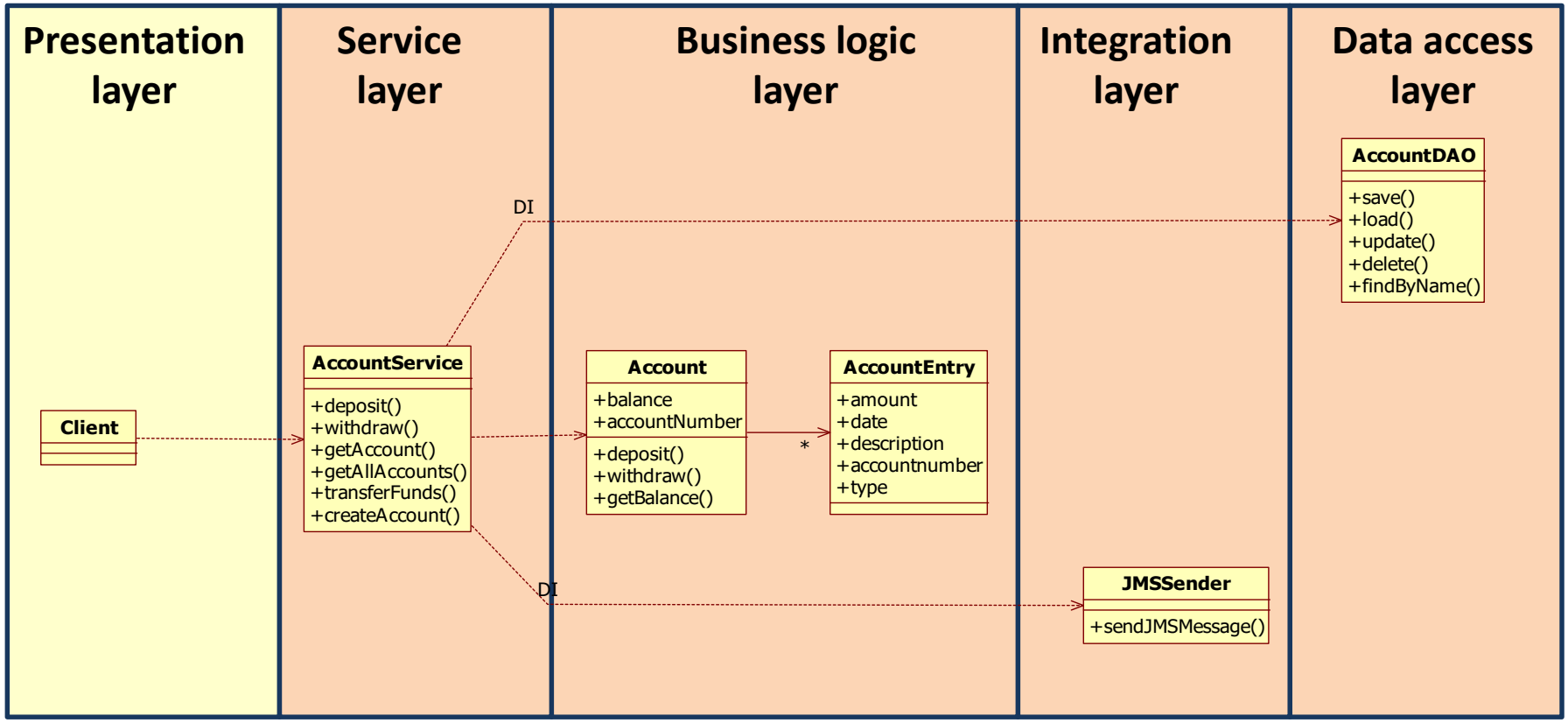


Service object



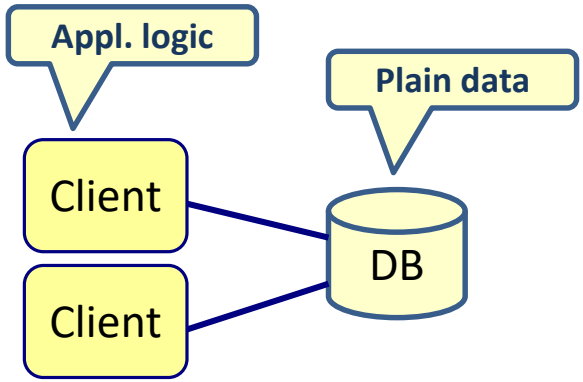
Main point

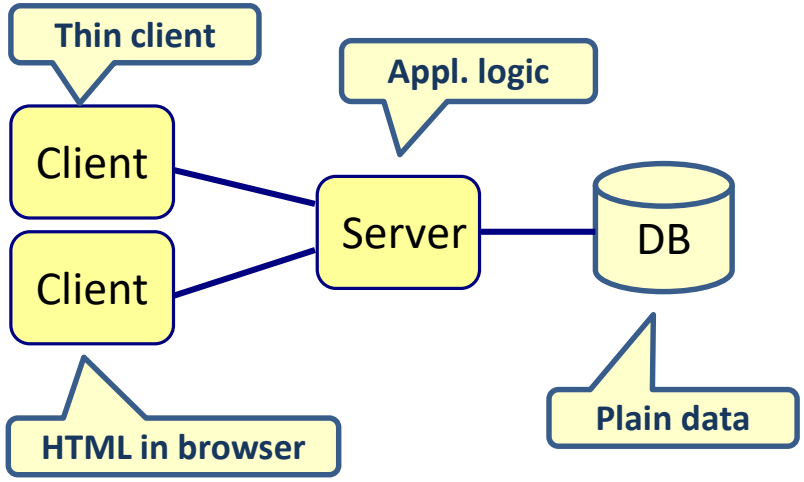
- The domain classes are never aware of technical “plumbing” classes. This gives many different advantages.
- By diving deep into pure consciousness, one gains support of all the laws of nature without needing to know or to be aware of all different details of your life and your world.

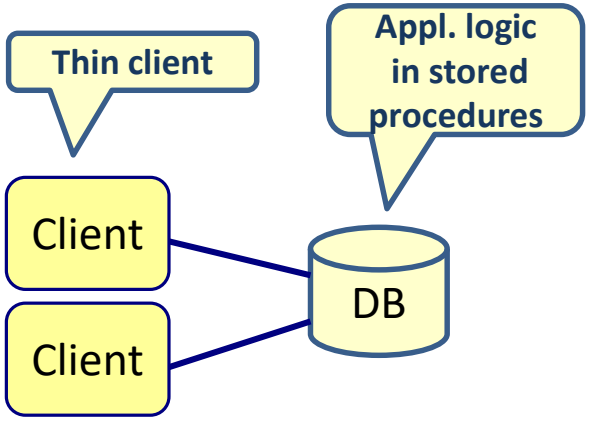


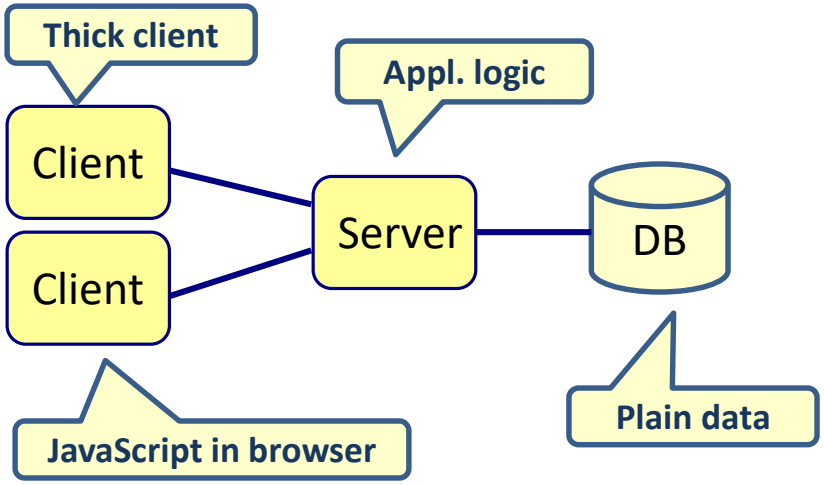
PRESENTATION LAYER

Client-server architectures

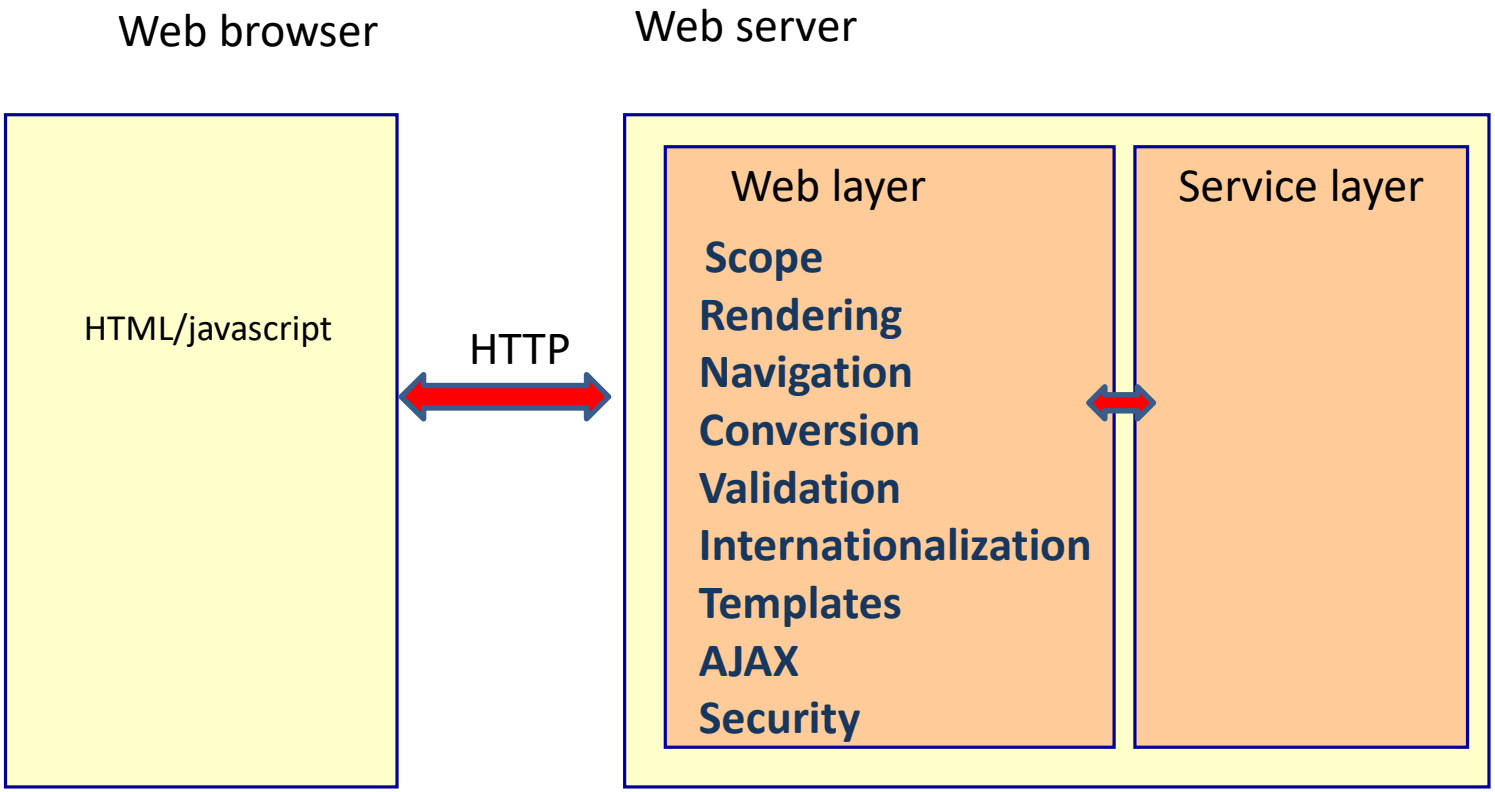






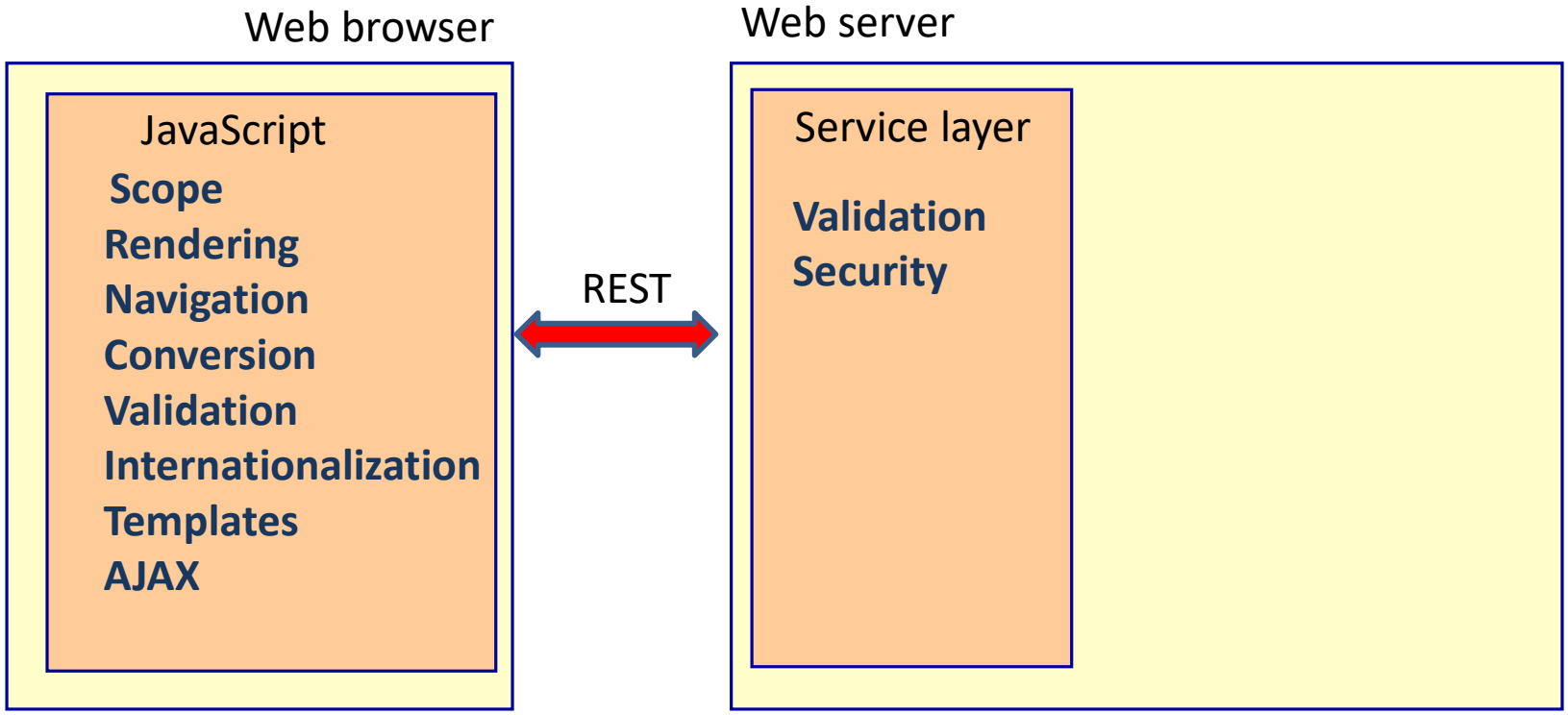


Server side web framework



- Every action is executed on the server

Client side web framework



- You only go to the server if you need to.

Server centric versus client centric

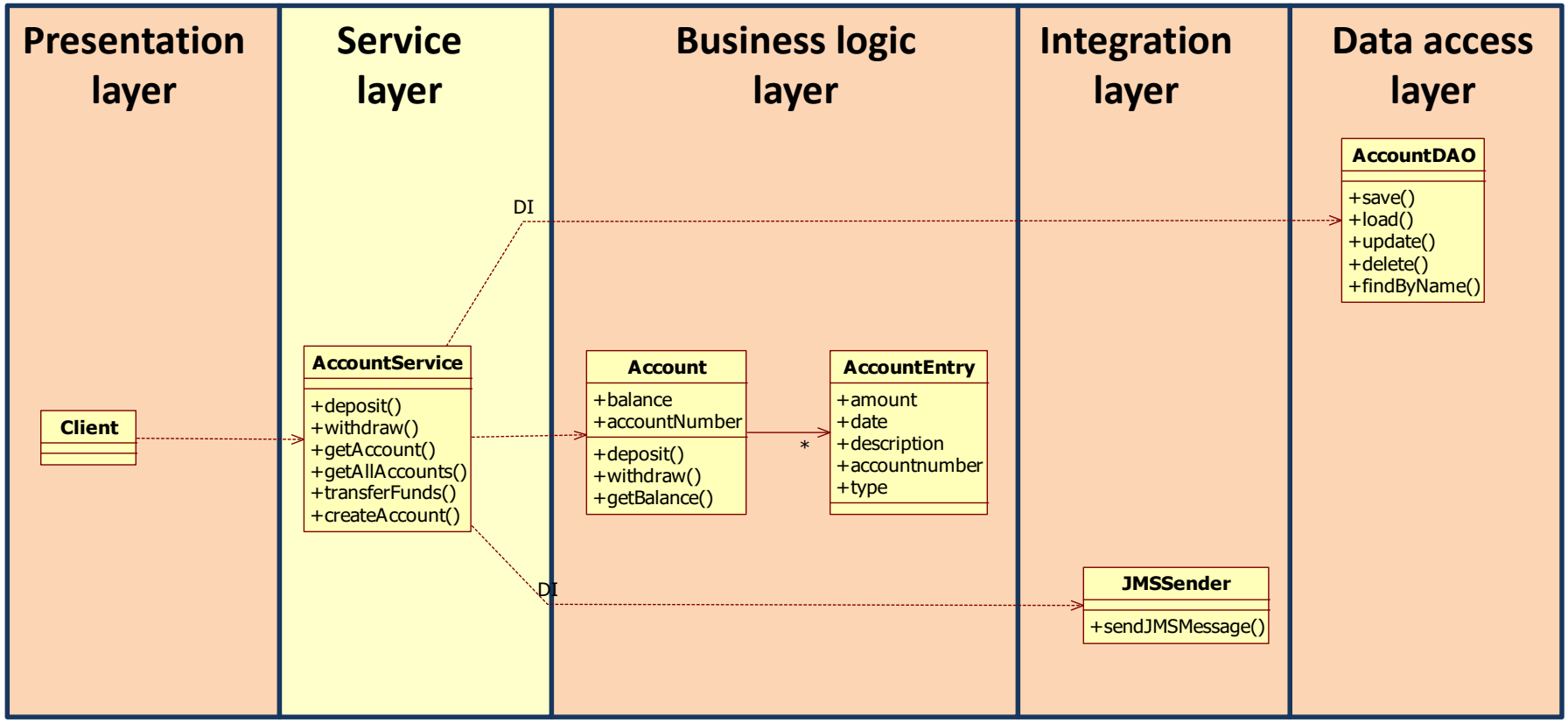
- Remove a stock from the watch list:
  - Sever centric: send a request to the server and execute on the server
  - Client centric: execute within the browser

Server centric versus client centric

- Server centric
  - Servlets/JSP
  - JSF
  - Spring MVC
- Client centric
  - Angular
  - React
  - Vue

Client centric

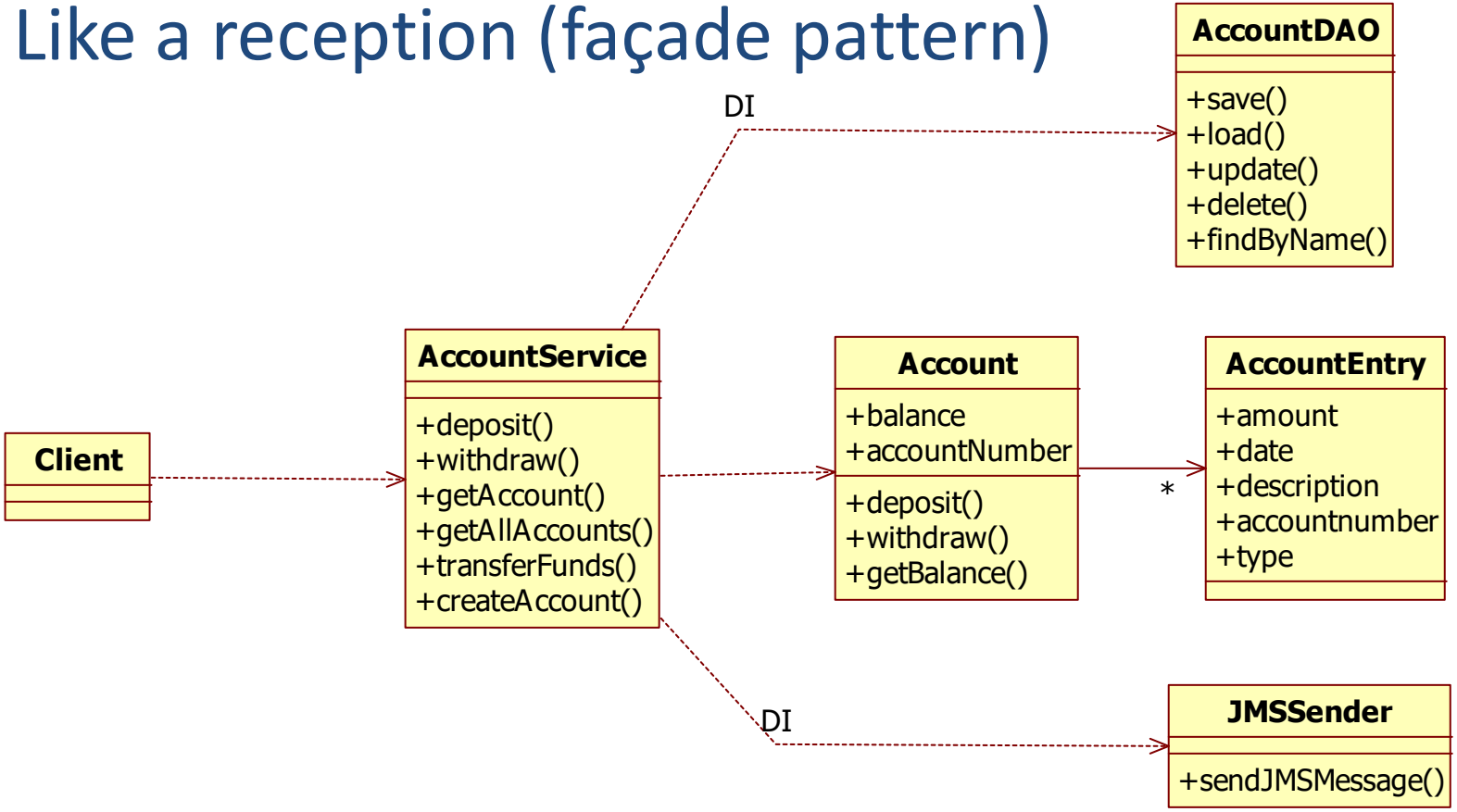
- Advantages
  - Javascript runs in the browser. It only goes to the server when it needs data.
    - Less network traffic between client and server, which makes the application more scalable
      - Less burden on the server
    - Faster response times in the client because we don't need to go to the server all the time
    - Separation between front-end and back-end allows different programmers/teams to work on the front-end or back-end, independent from each other
    - Separation between front-end and back-end support using different front-end channels
- Disadvantage
  - The frameworks and techniques change very fast

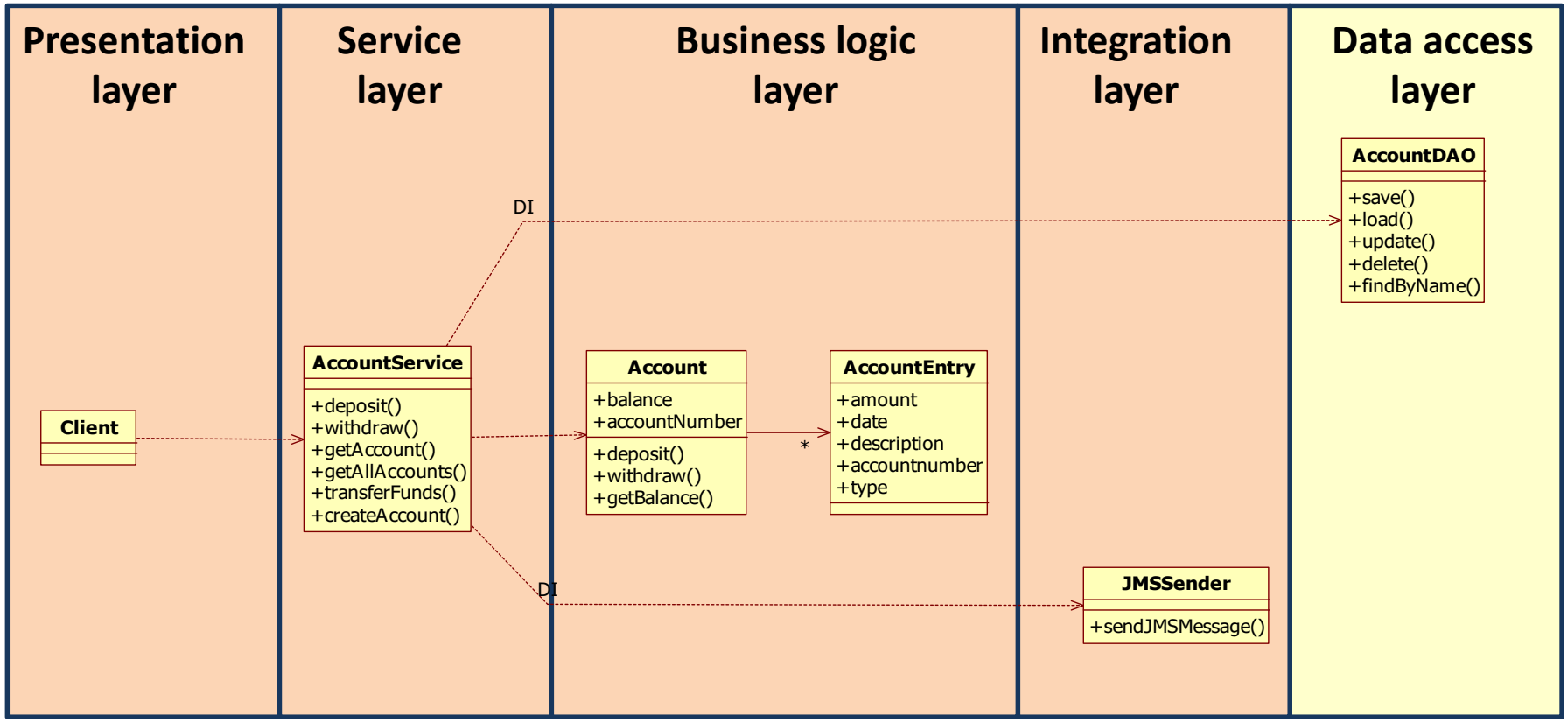


SERVICE LAYER

Service class

- Like a reception (façade pattern)

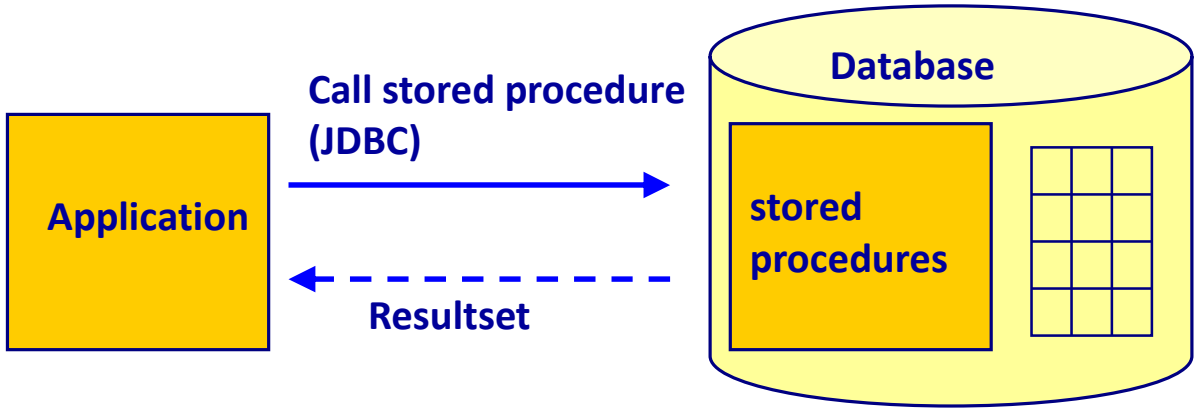




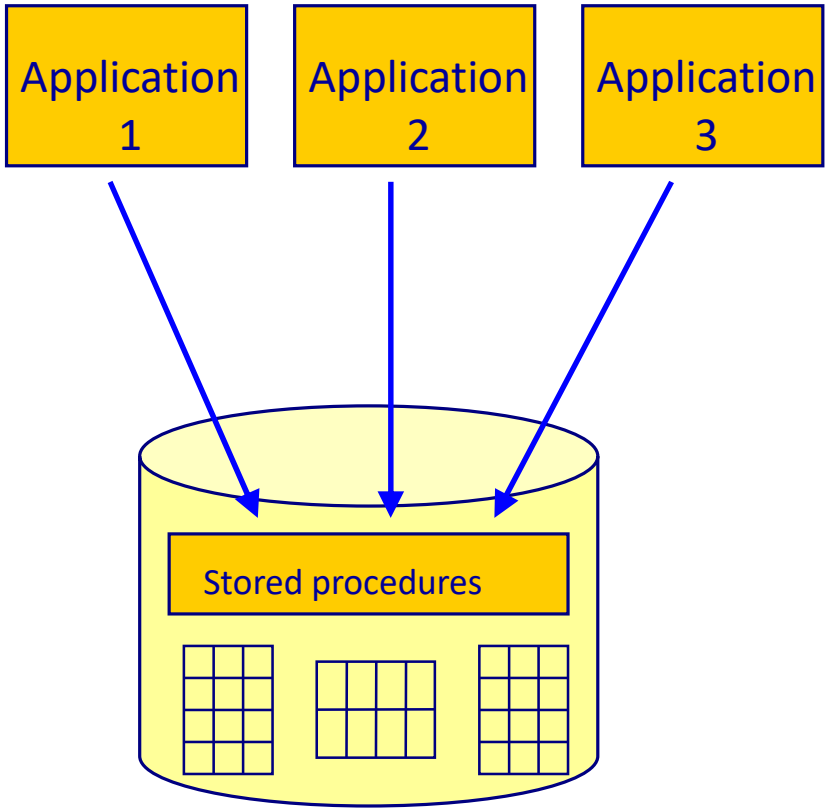
DATA ACCESS LAYER

Stored procedures

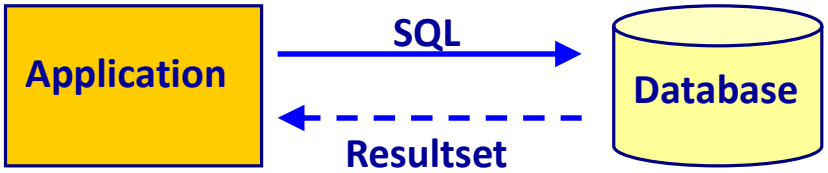
- Logic that runs in the database
- Fast
- Difficult to maintain when number of stored procedures grows
  - Every schema change leads to changes to the stored procedures
  - Lot of duplications, not much reuse
- PL/SQL
- Java Stored Procedures



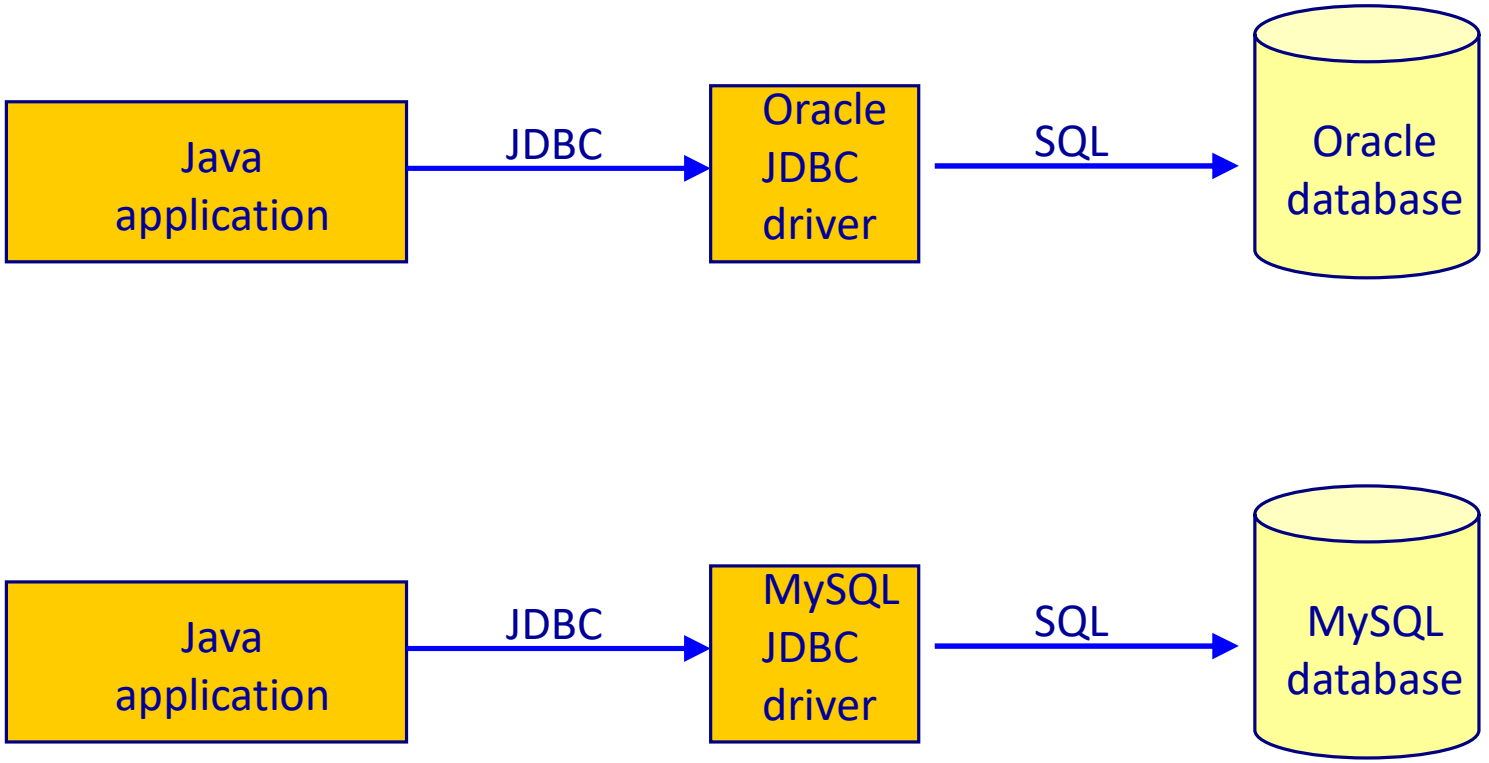
Layer of indirection



SQL based approach: JDBC

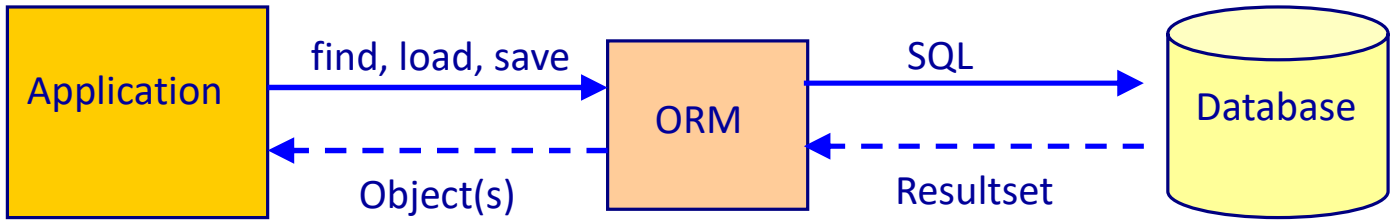


JDBC



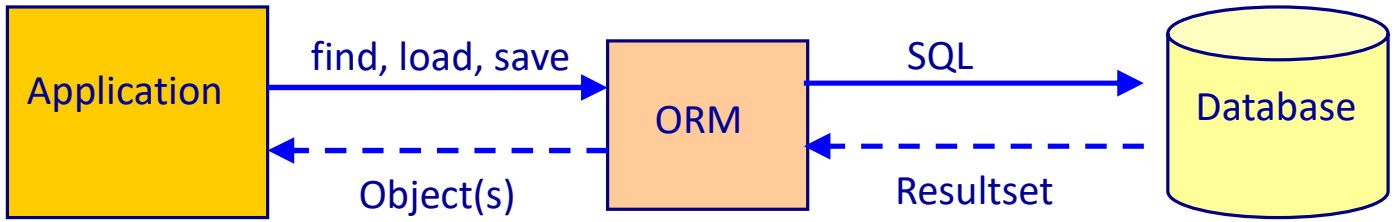
Object Relational Mapping (ORM)

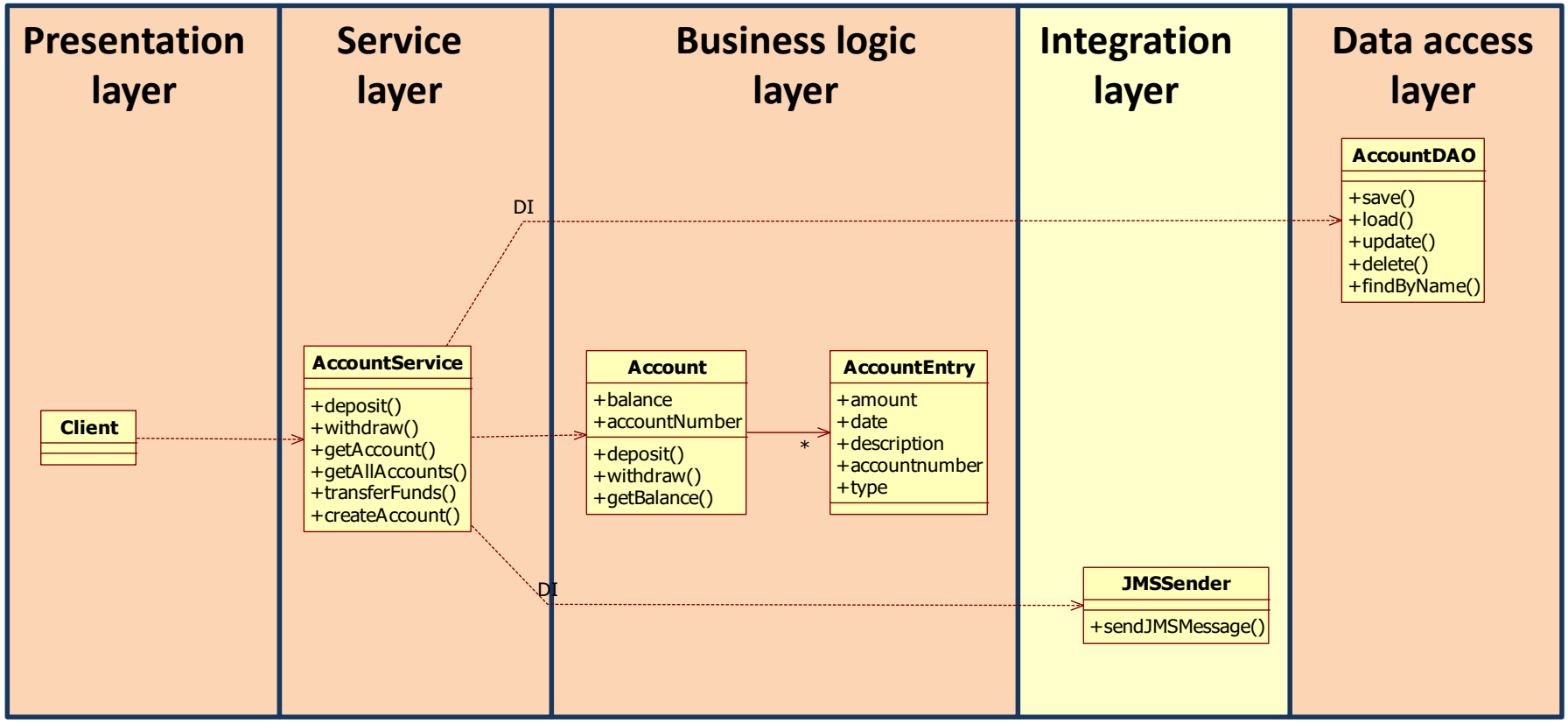
- Object Relational Mapping lets the programmer focus on the Object Model
  - Supports Domain Driven Development (DDD)
  - Programmer can just work with objects
  - Once an object has been retrieved any related objects are automatically loaded as needed
  - Changes to objects can automatically be stored in the database



Advantages of ORM

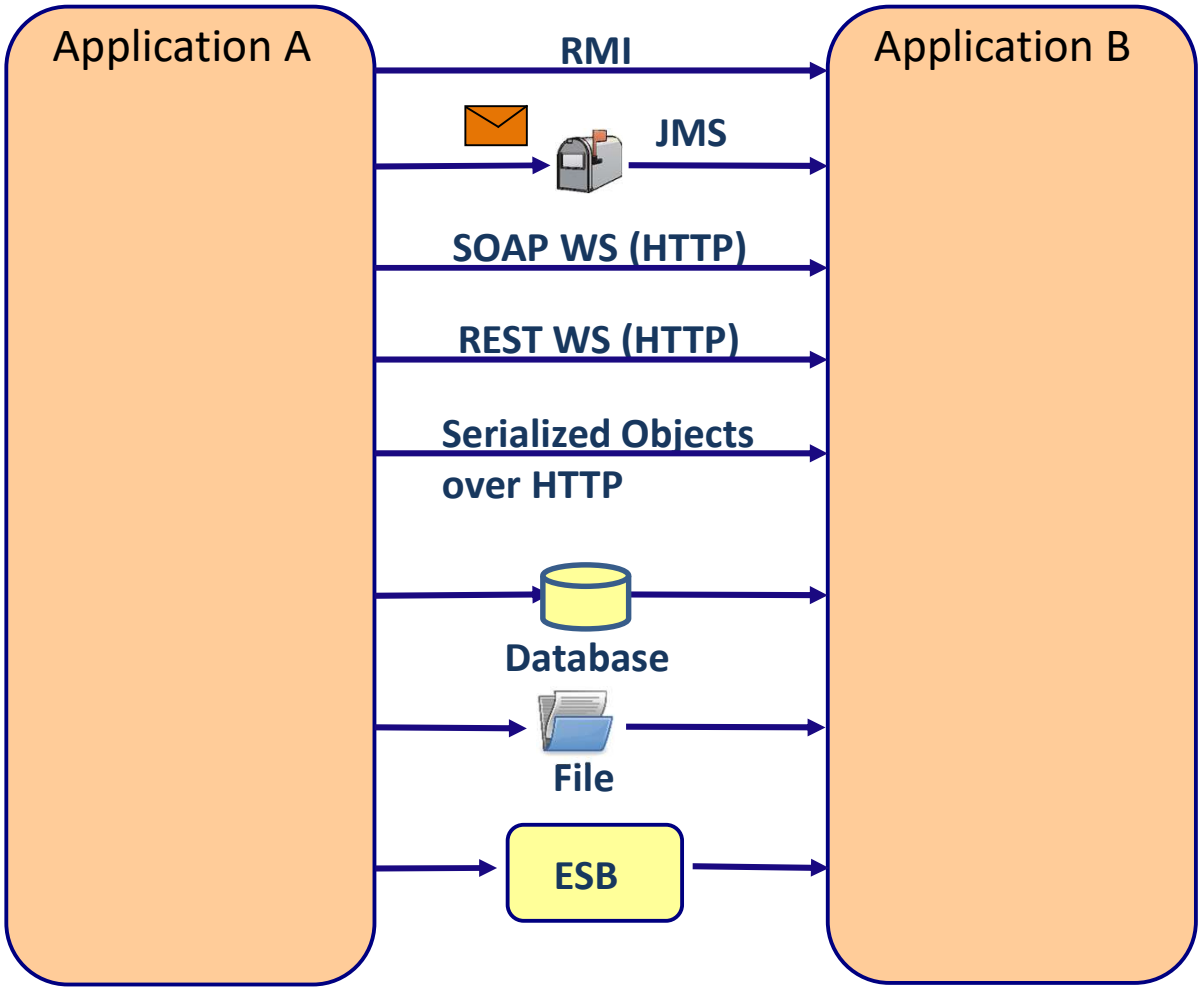
Advantage	Details
Productivity	•Fewer lines of persistency code
Maintainability	•Fewer lines of persistency code •Mapping is defined in one place
Performance	•Caching •Higher productivity gives more time for optimization ✓Projects under time pressure often don't have time for optimization •The developers of the ORM put a lot of effort in optimizing the ORM





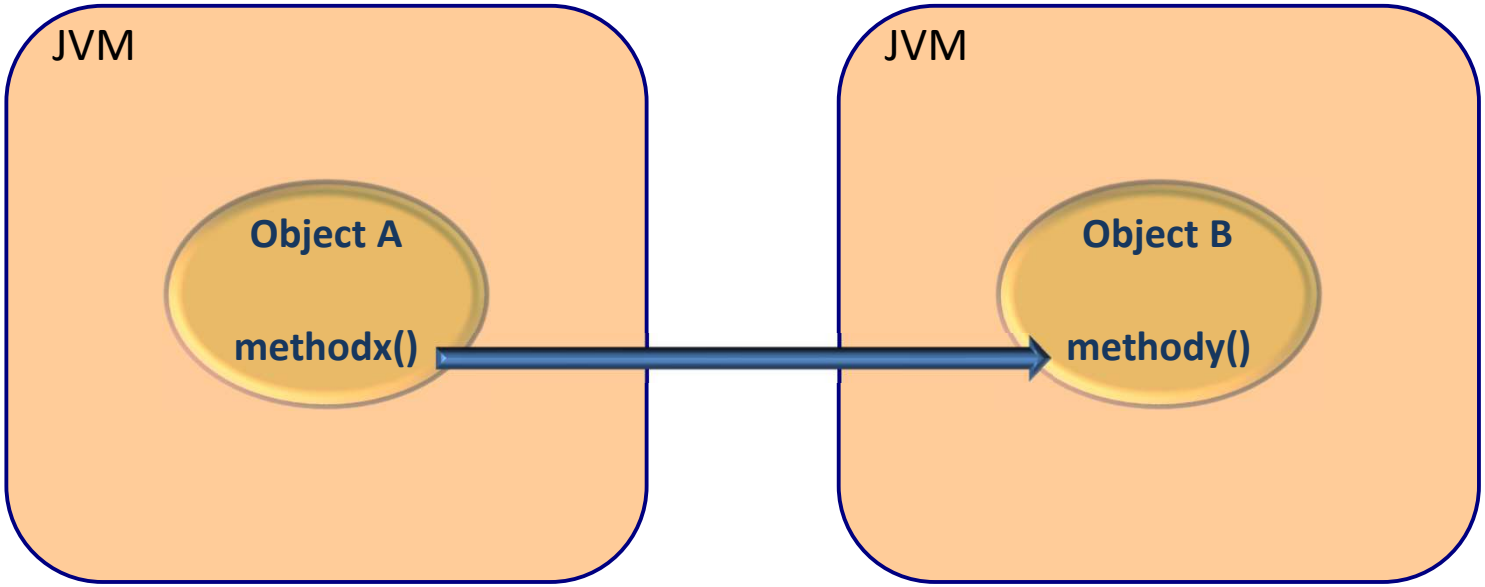
INTEGRATION LAYER

Integration possibilities



RMI

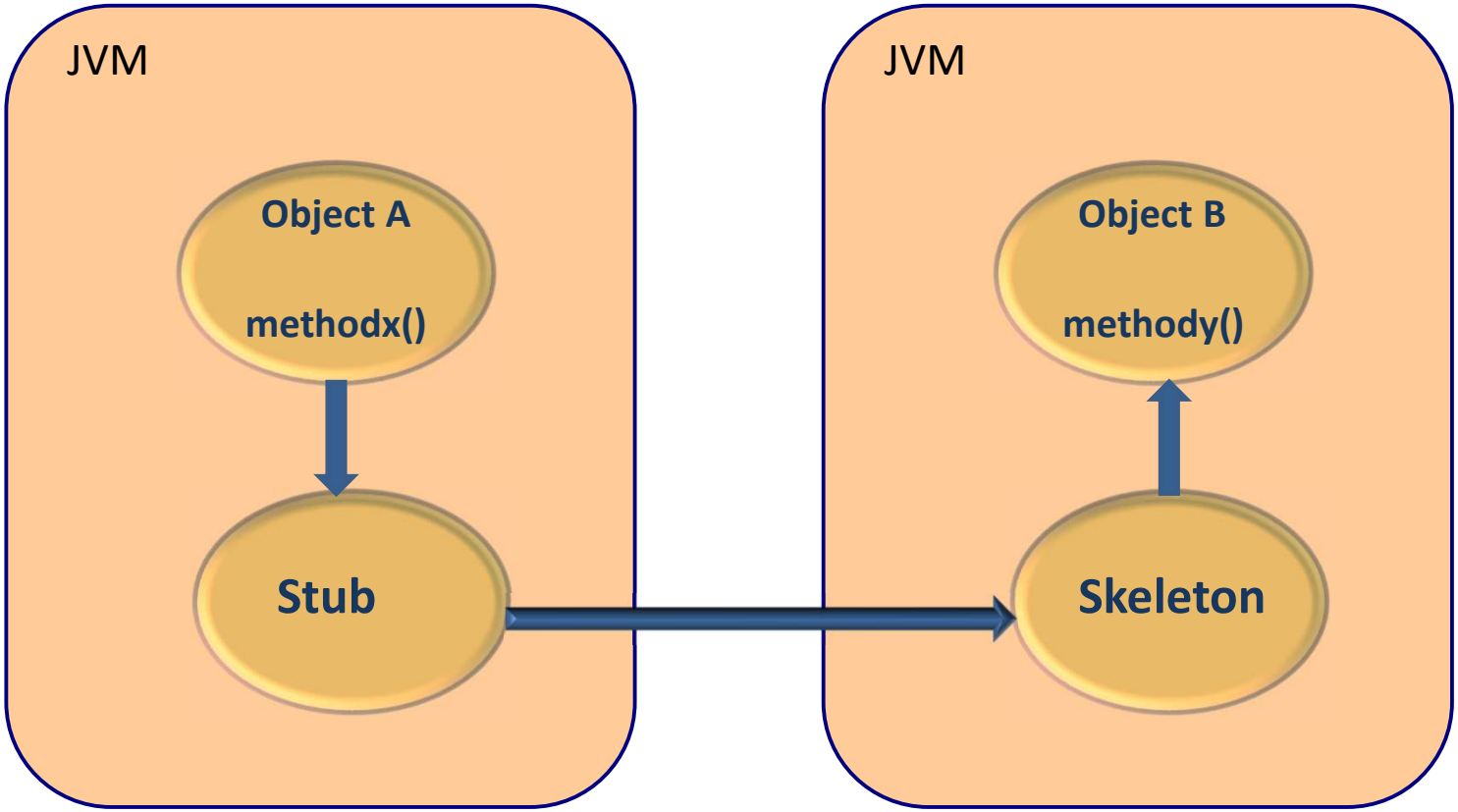
- An object calls a method of another object that lives in a different virtual machine.



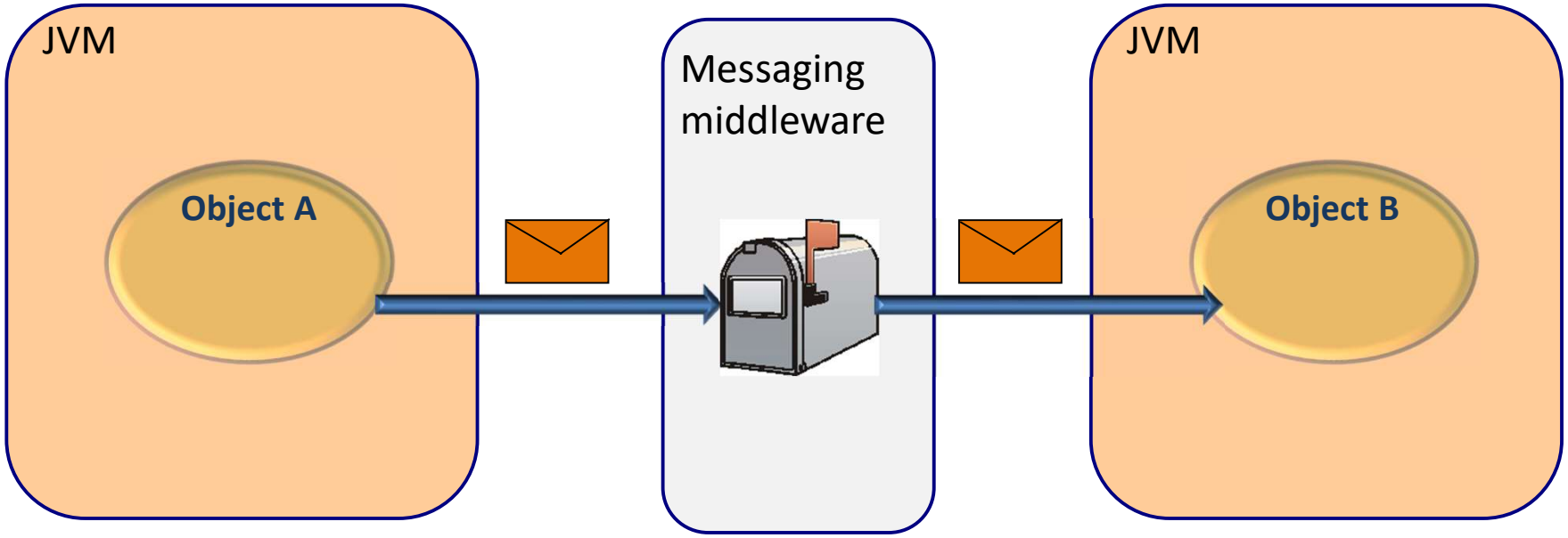
Characteristics of RMI

- Synchronous
  - The calling object has to wait until the remote method call returns
- Call by value
  - If the remote method needs other objects as parameters, these parameter objects will be serialized and will be sent to the remote object.
  - All associated object will also be serialized.

Stub and skeleton

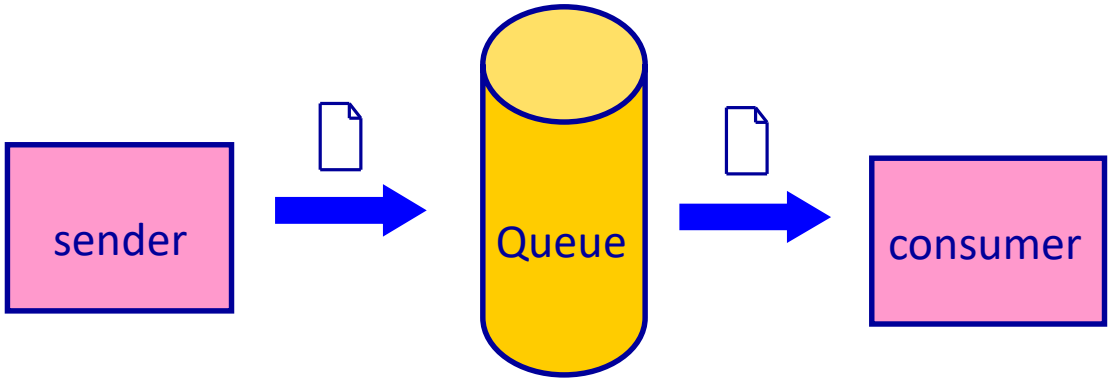


Java Message Service (JMS)



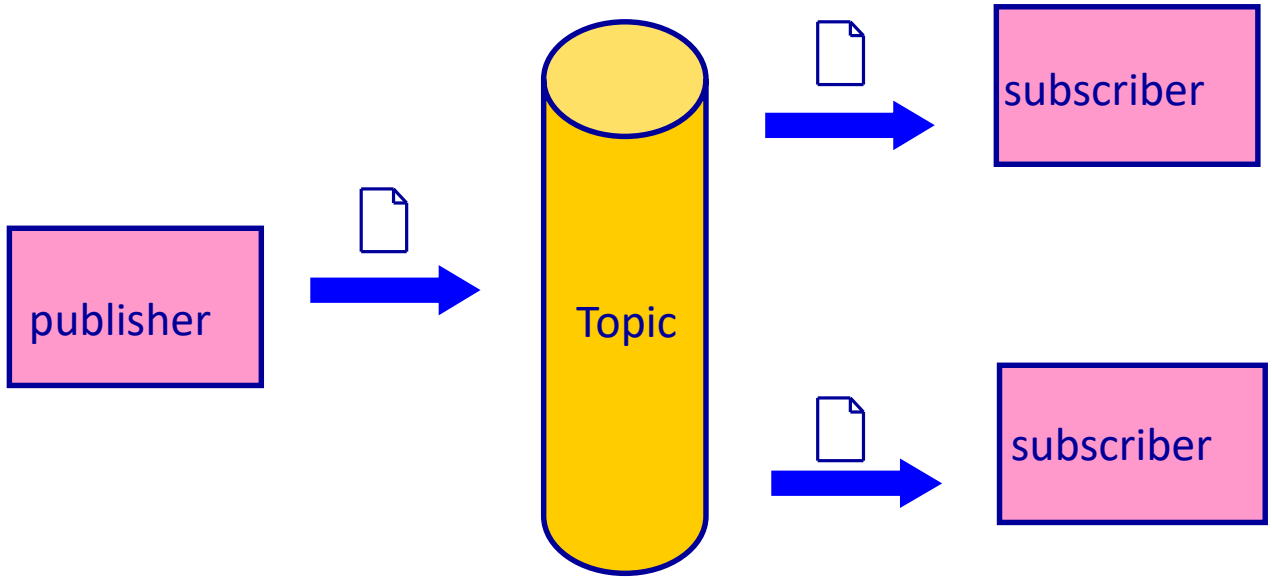
Point-To-Point (PTP)

- A dedicated consumer per Queue message

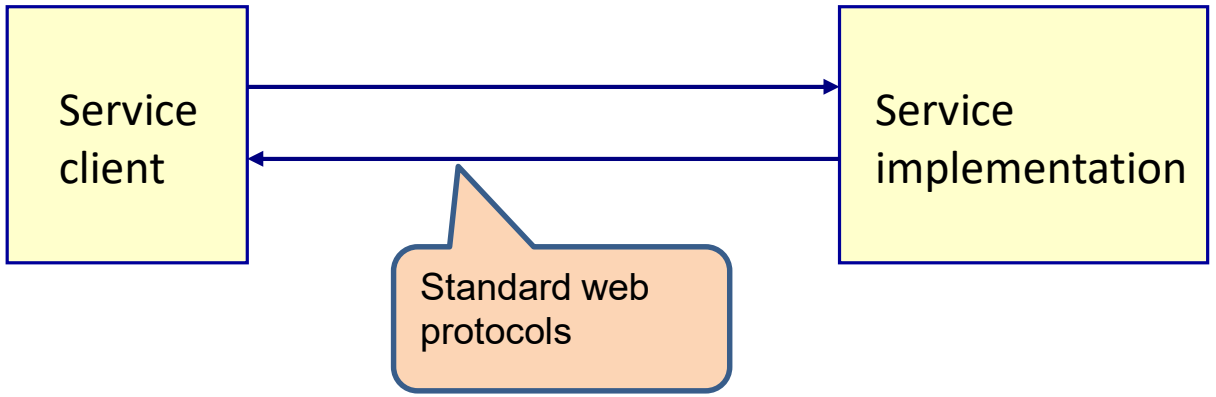


Publish-Subscribe (Pub-Sub)

- A message channel can have more than one '*consumer*'
  - Ideal for broadcasting



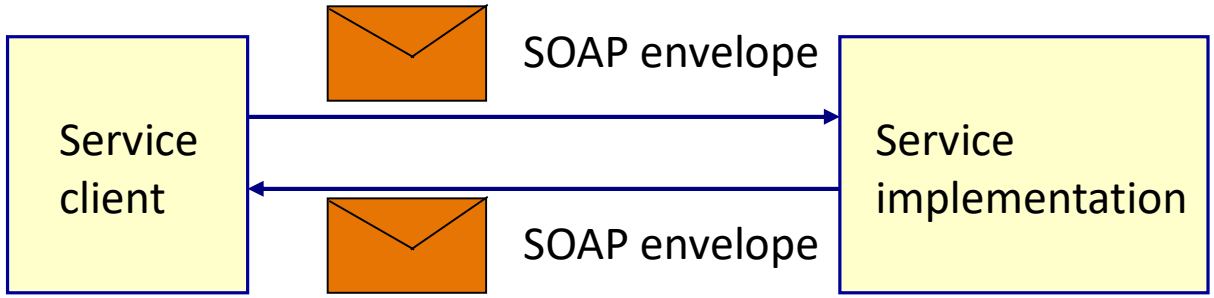
What is a Web Service?



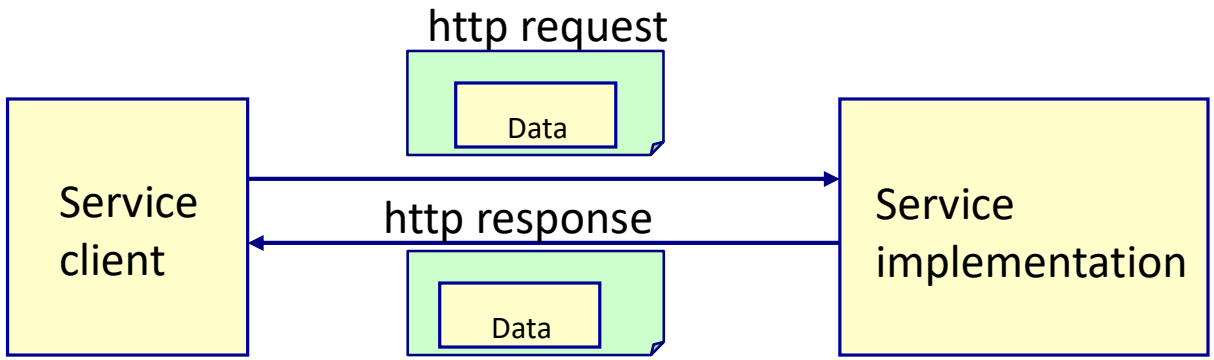
- A web service offers functionality that can be called by other clients using standard web protocols (SOAP, XML, HTTP)

Types of Web Services

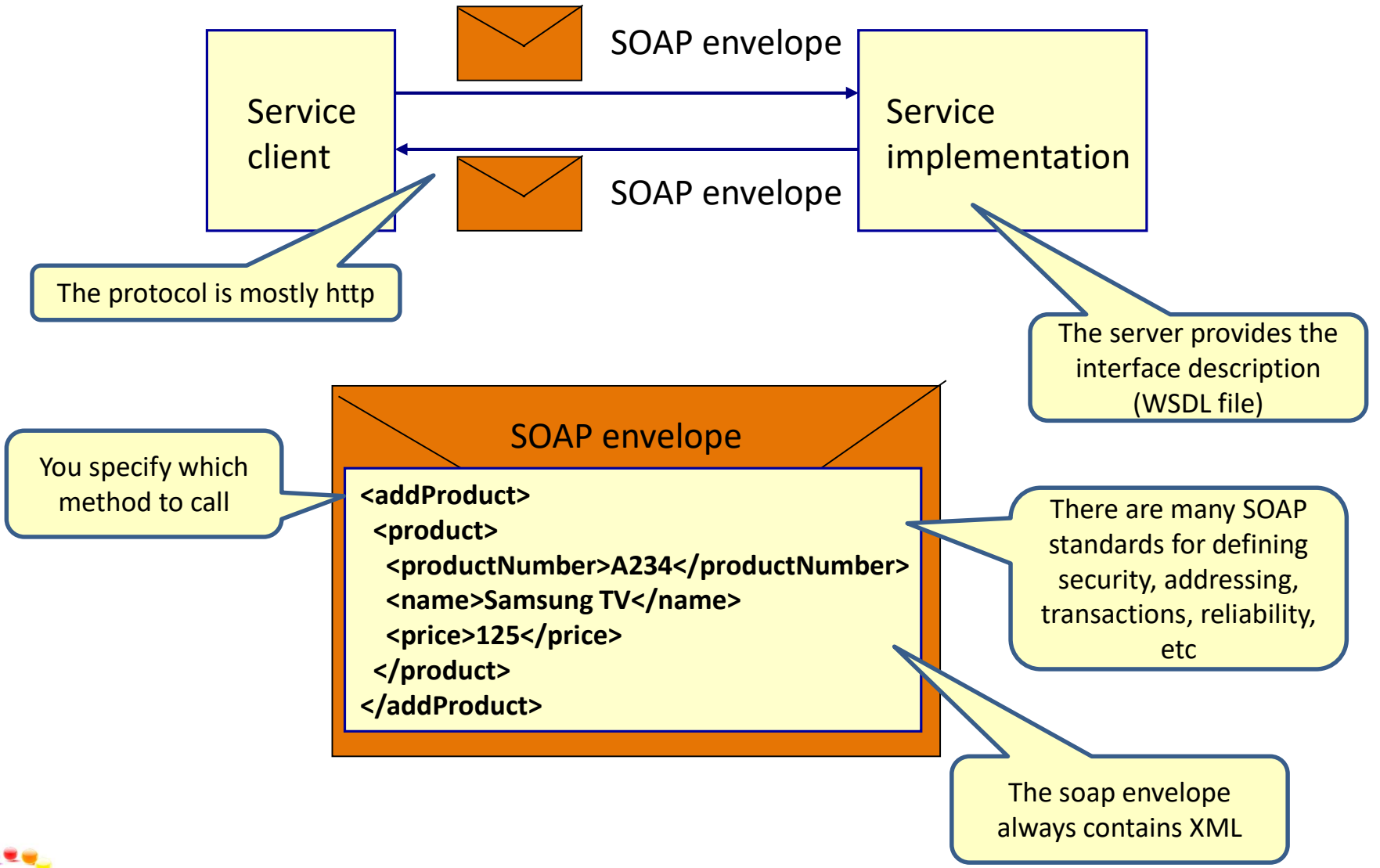
- SOAP



- REST

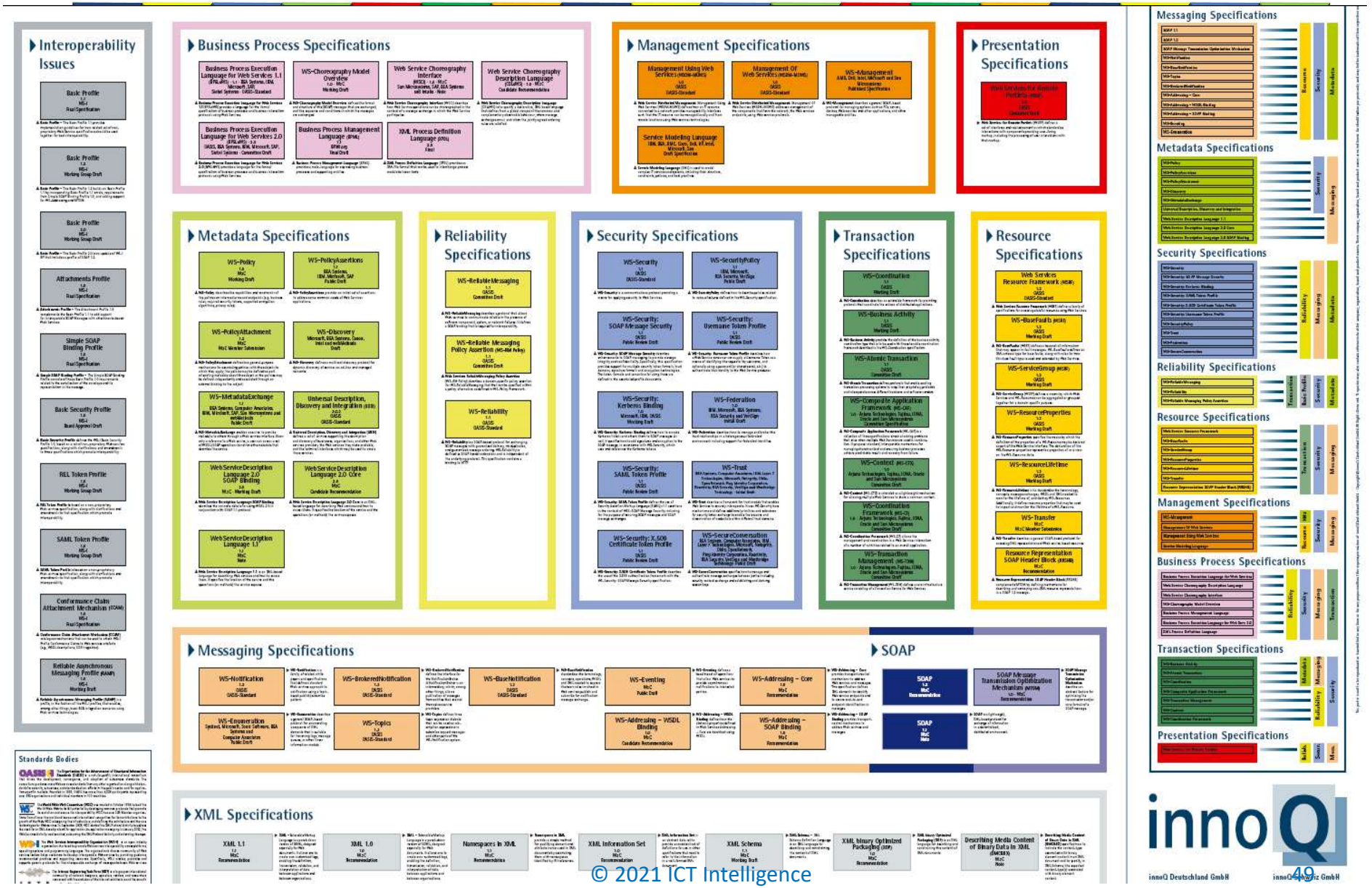


Simple Object Access Protocol (SOAP)

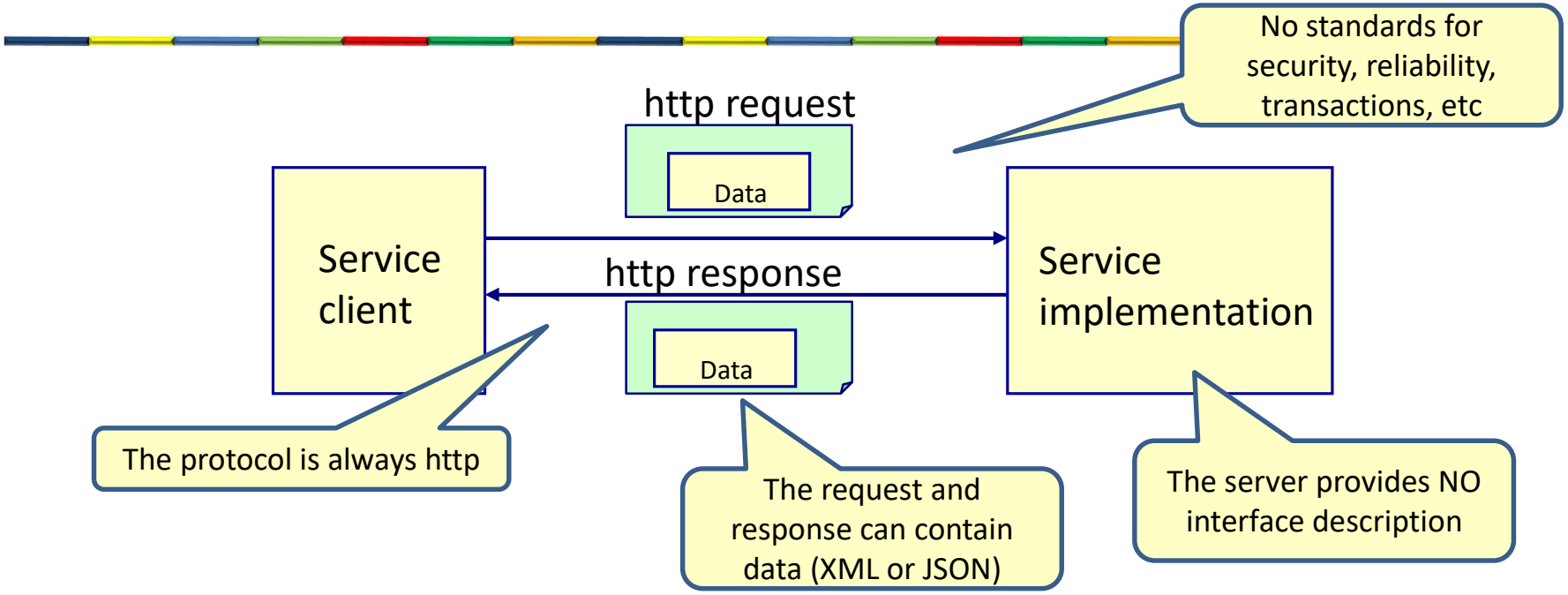


SOAP example

SOAP standards

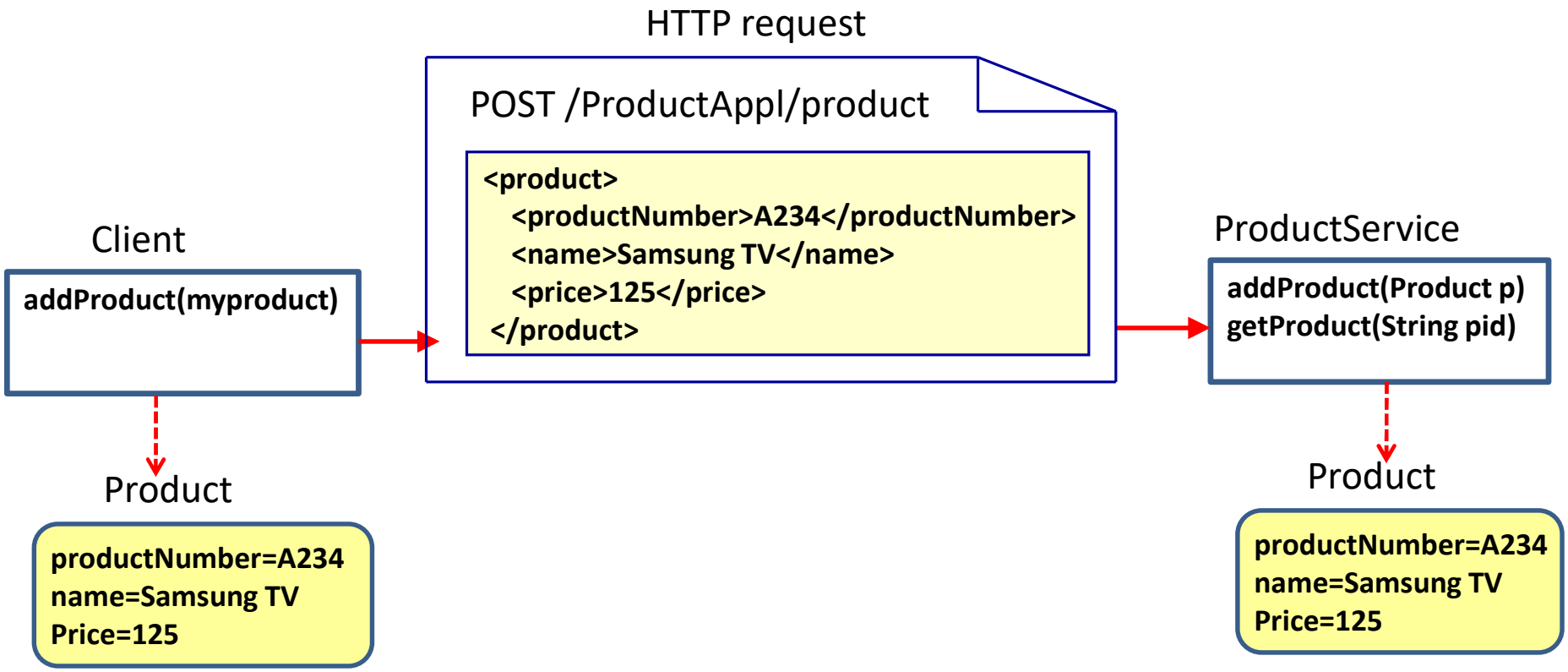


RESTful Web Services

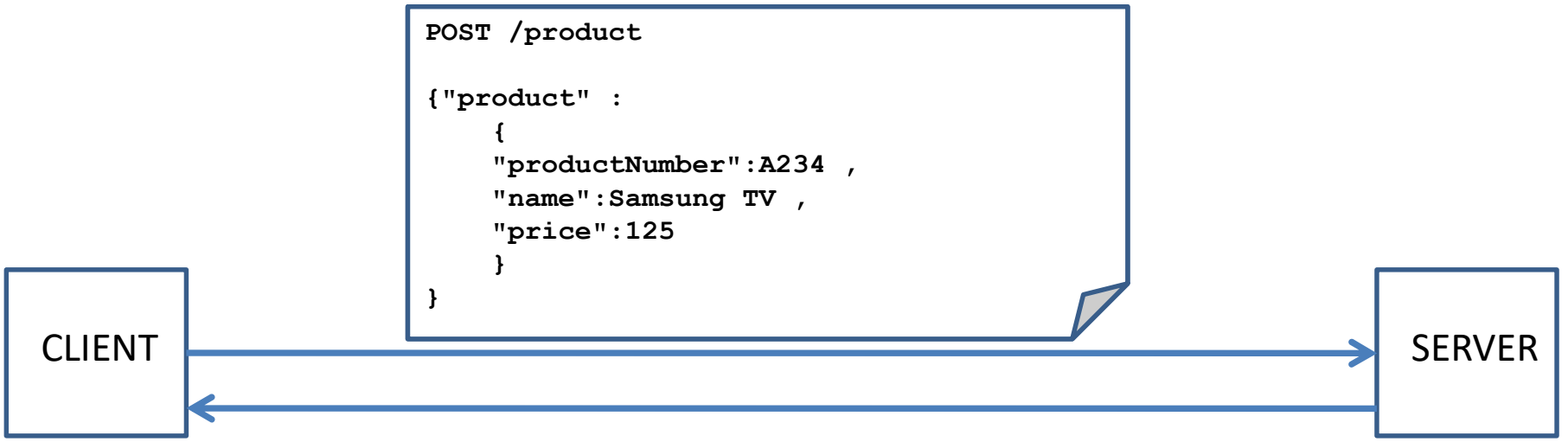


- Data in HTTP messages
  - GET message for retrieving data
  - POST message for creating data
  - PUT message for updating data
  - DELETE message for deleting data

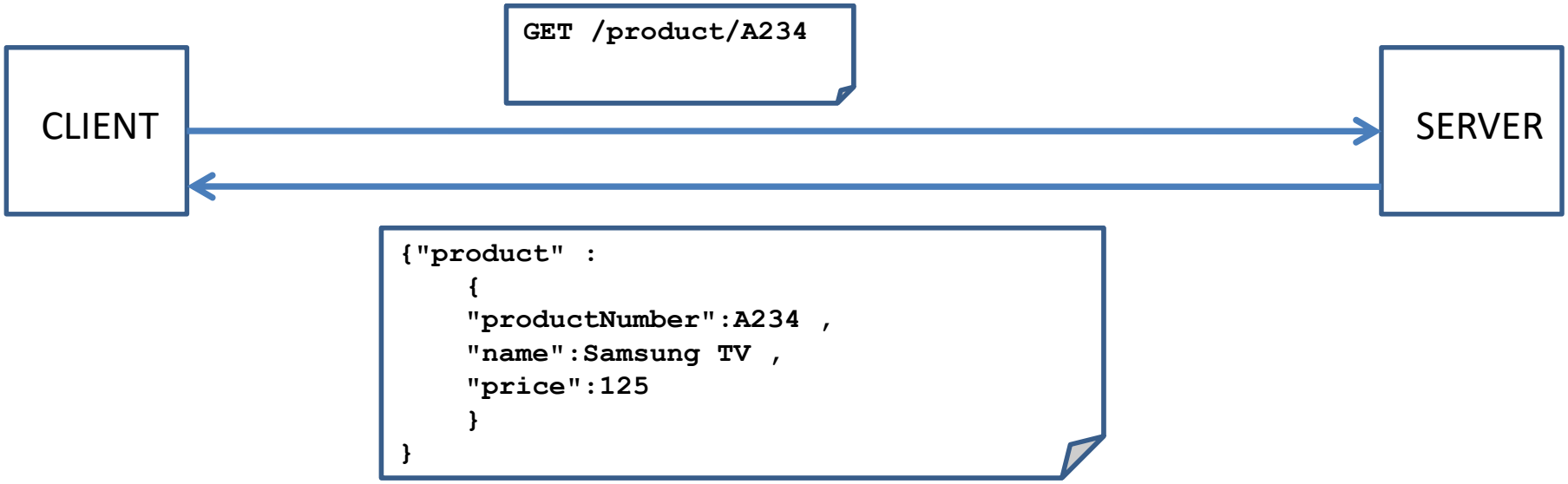
REST example



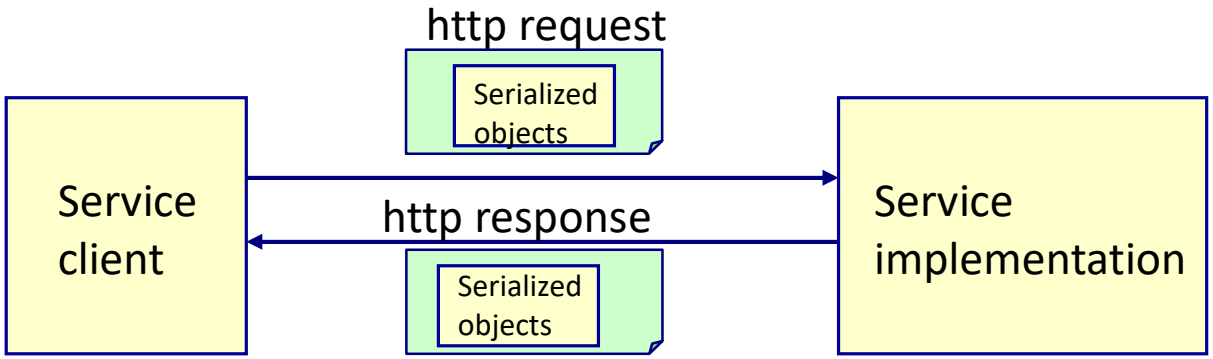
POST method using JSON



GET method using JSON

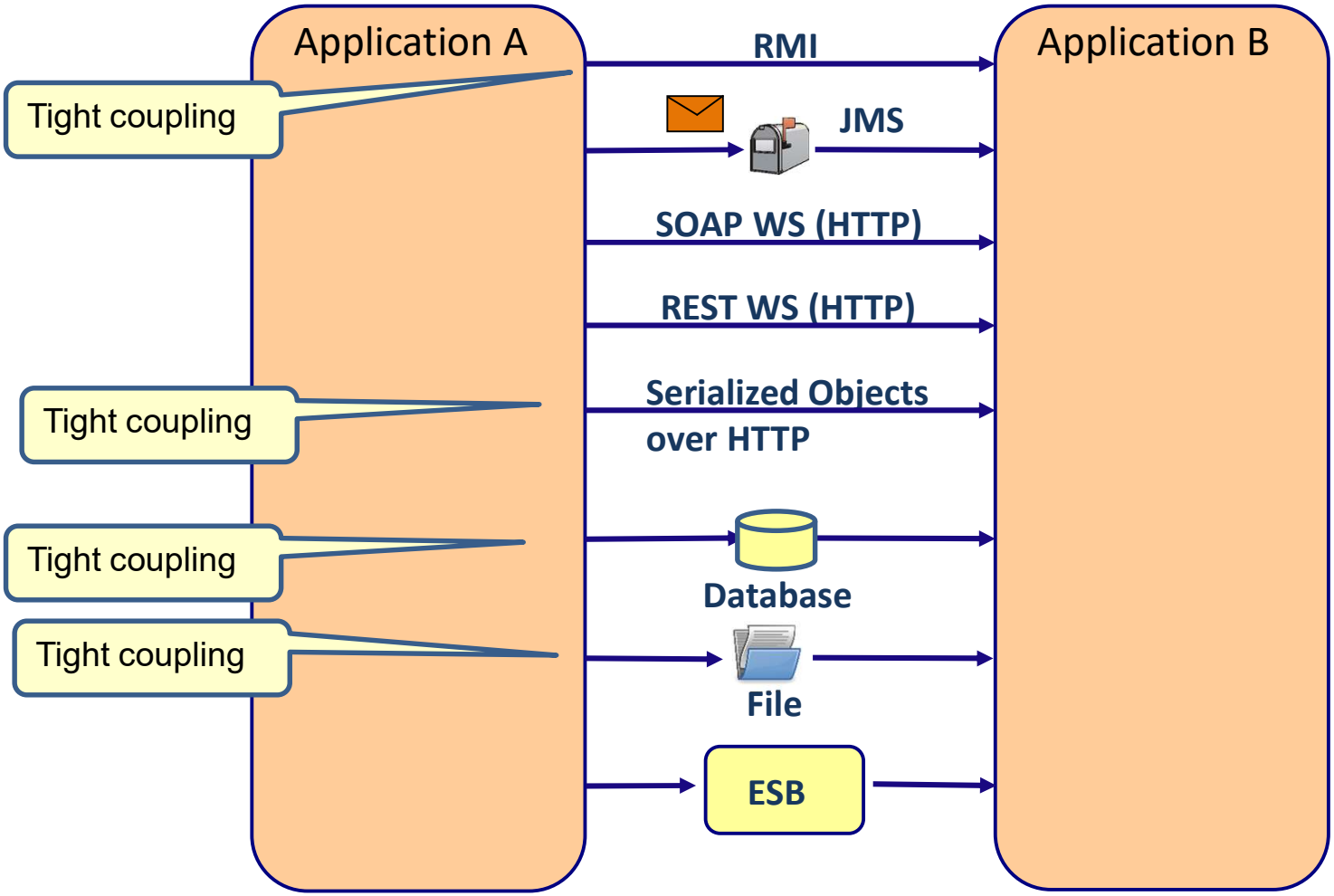


Serialized objects

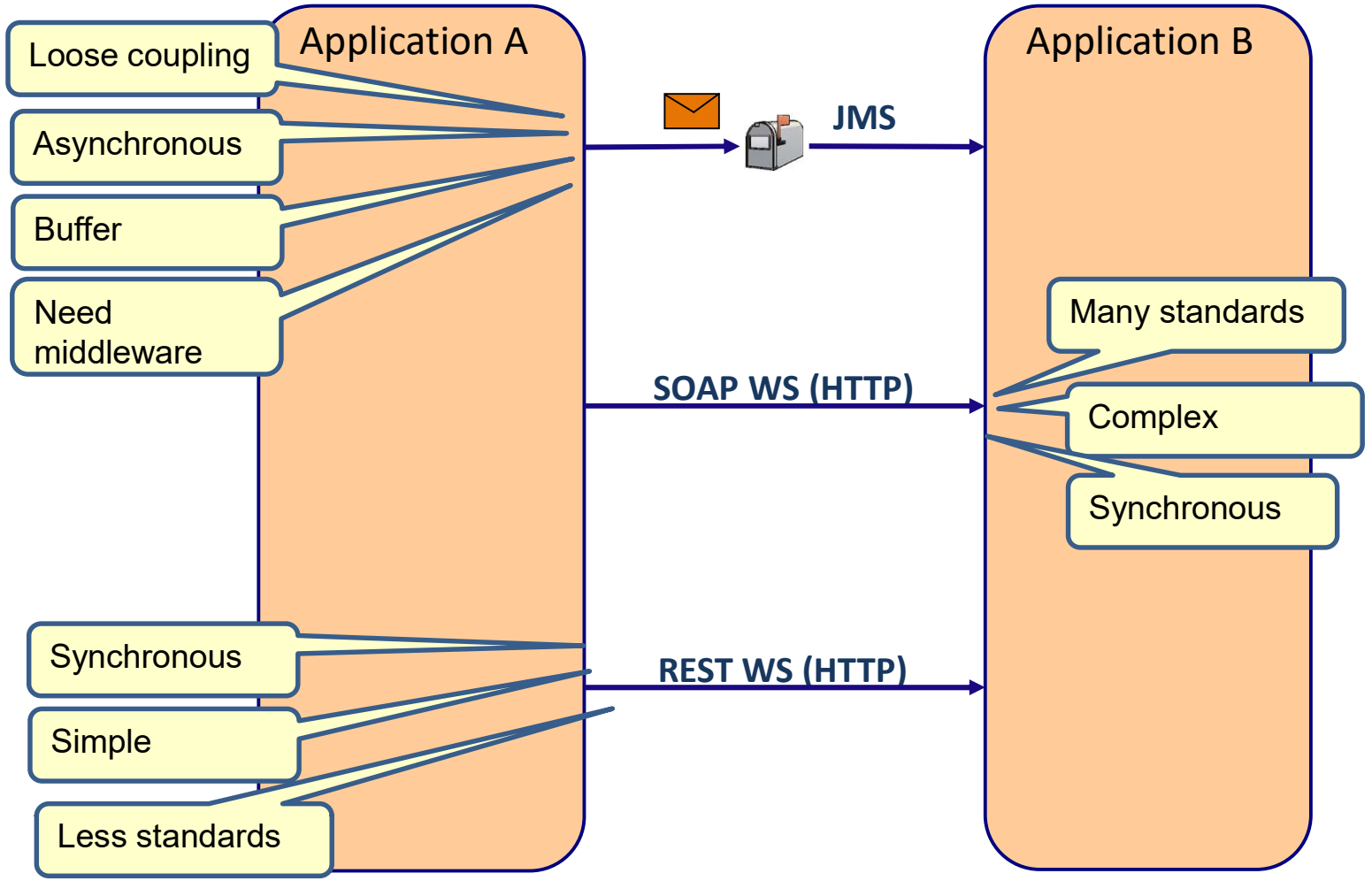


- If the client and server are both Java
- Sending serialized object is faster than sending XML
- Like RMI over HTTP

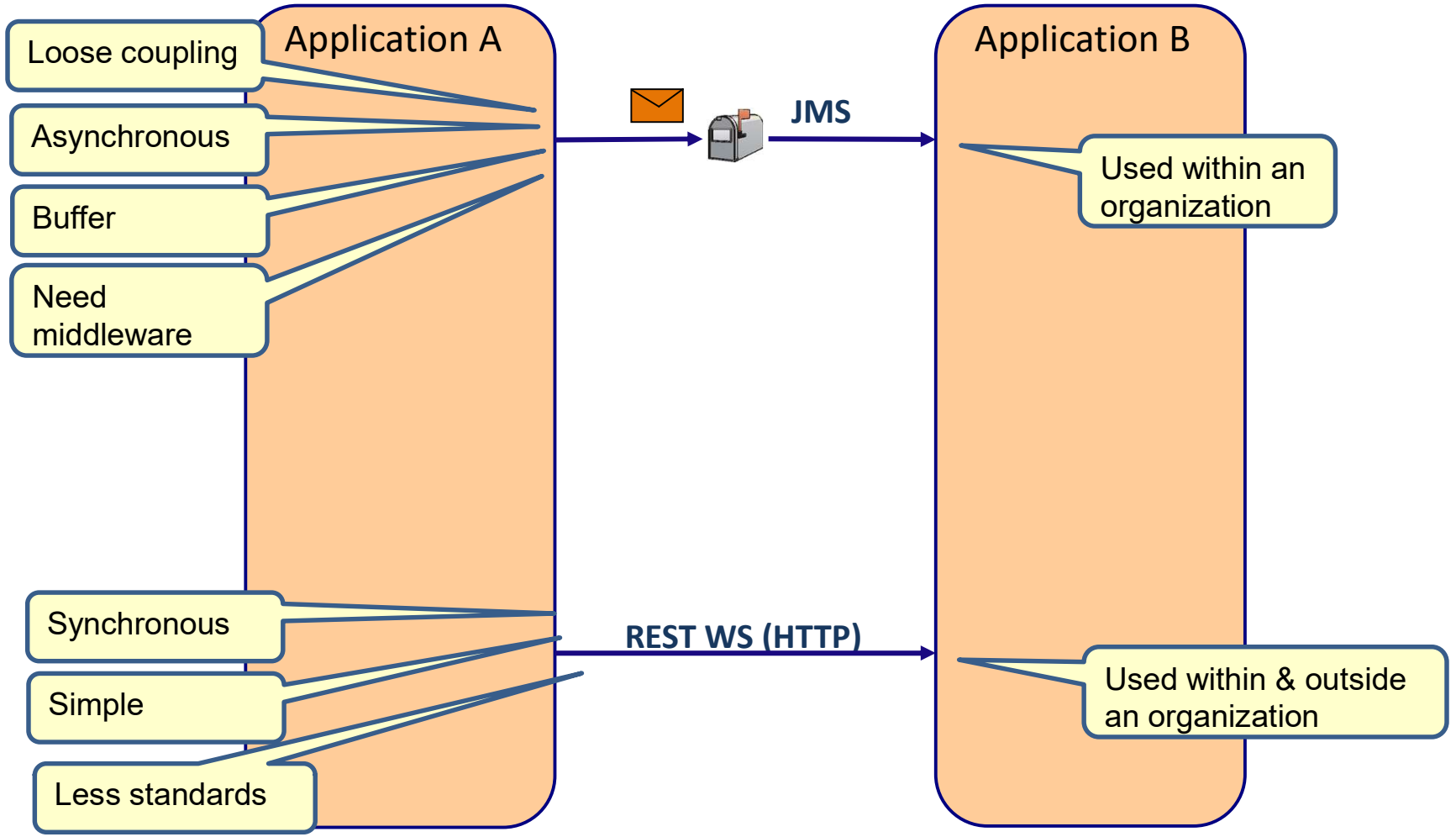
Integration possibilities



Integration possibilities



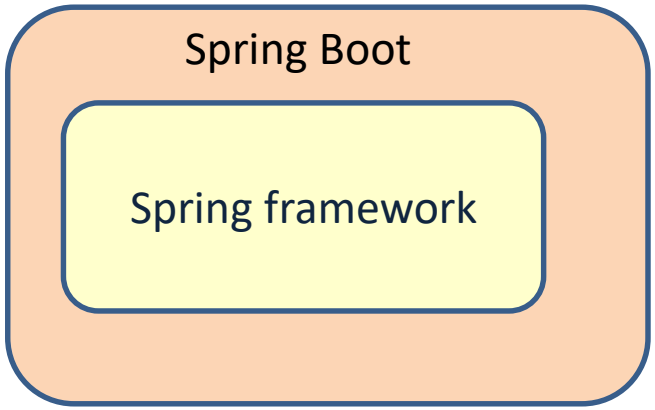
Integration possibilities



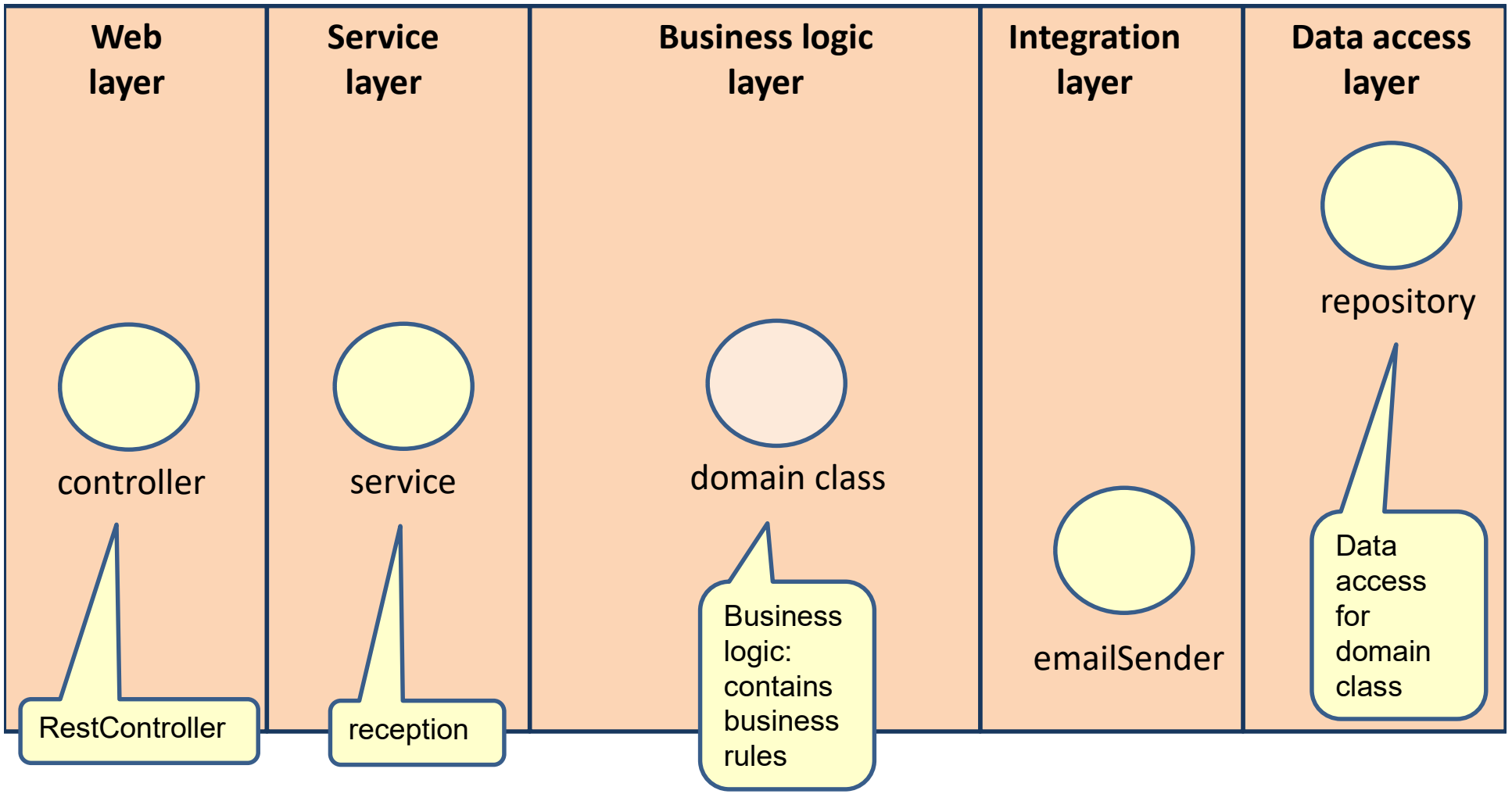
SPRING BOOT

Spring boot

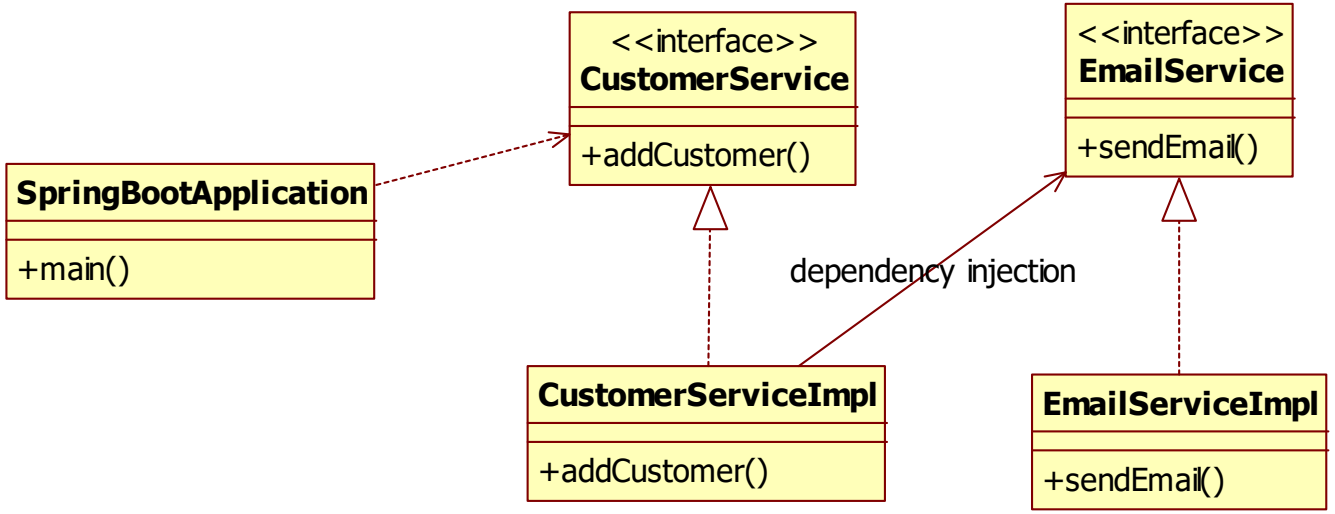
- Framework that makes it easy to configure and run spring applications
- Simple maven configuration
- Default/auto spring configuration
  - Opinionated framework
    - Convention over configuration
- Containerless deployment

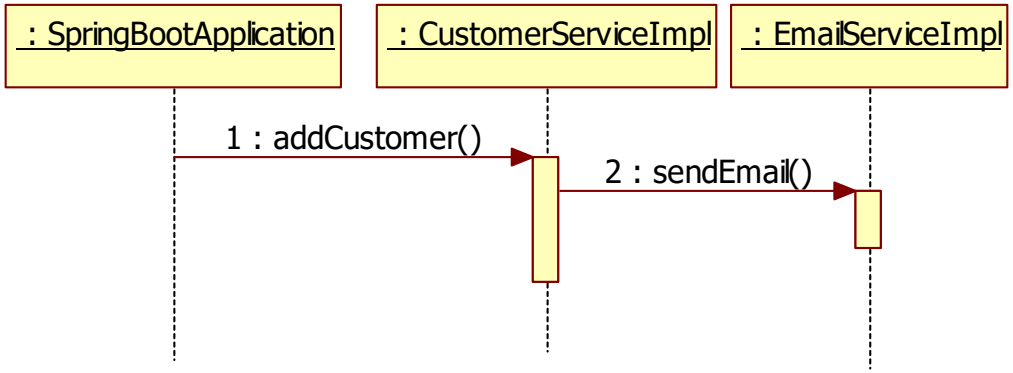


Layered architecture



Dependency injection





Dependency injection: Setter injection

```java
@Service
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  @Autowired
  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }


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

Setter injection

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

Dependency injection: Constructor injection

```java
@Service
public class CustomerServiceImpl implements CustomerService {

  private EmailService emailService;

  @Autowired
  public CustomerService(EmailService emailService) {
    this.emailService = emailService;
  }


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

Costructor injection

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

Dependency injection: Field injection

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

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

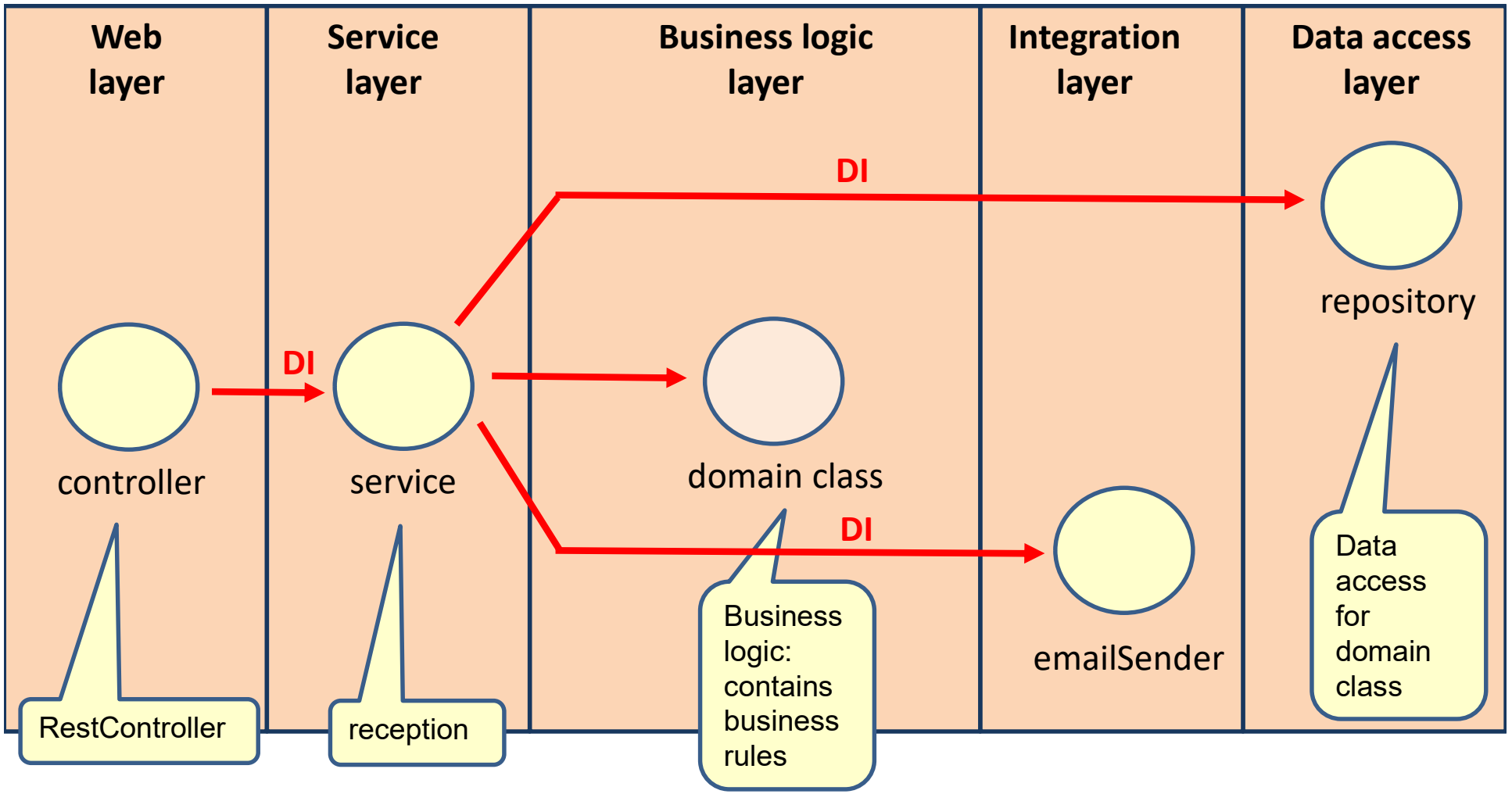
Field injection

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

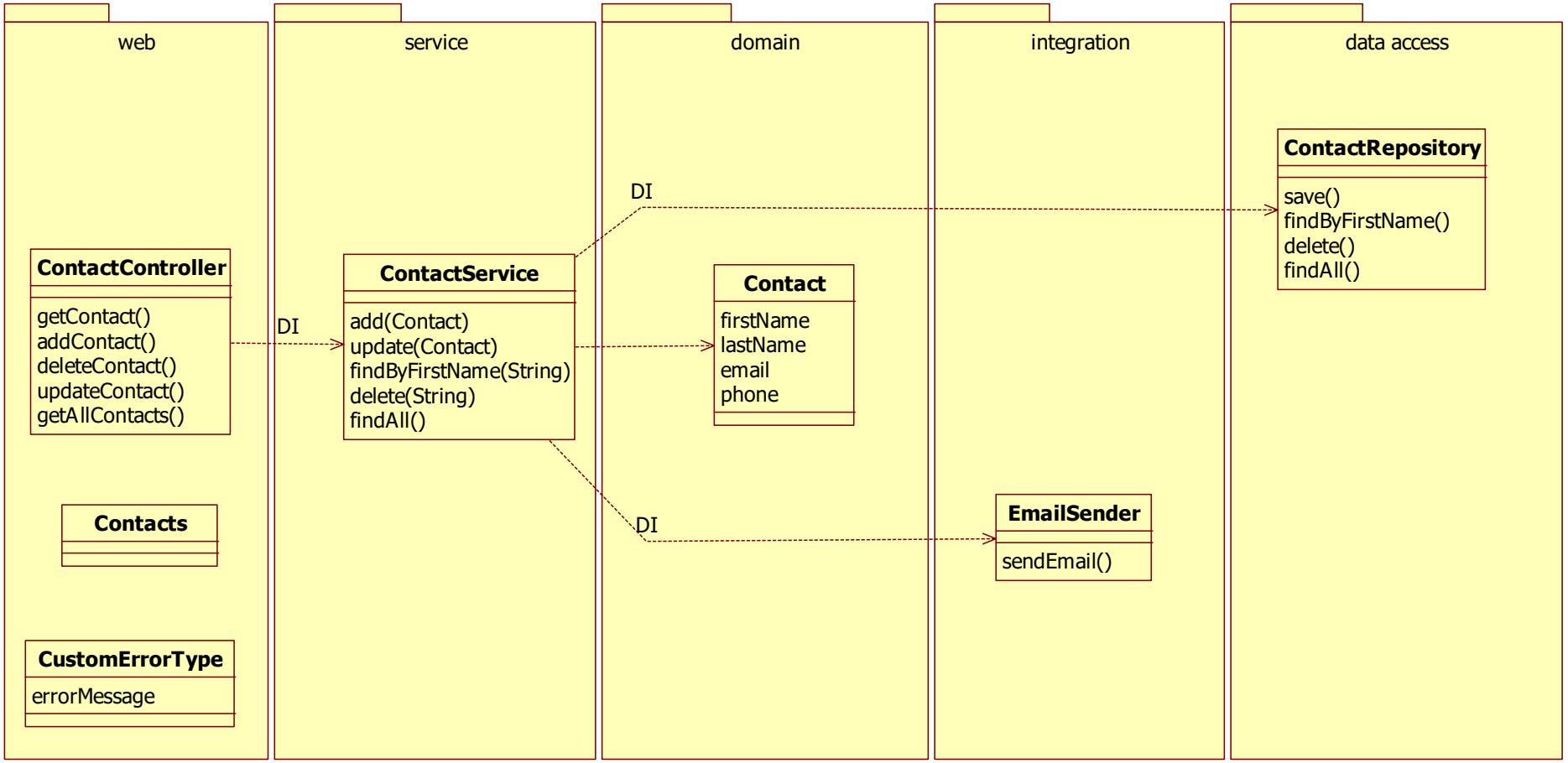
Main point

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

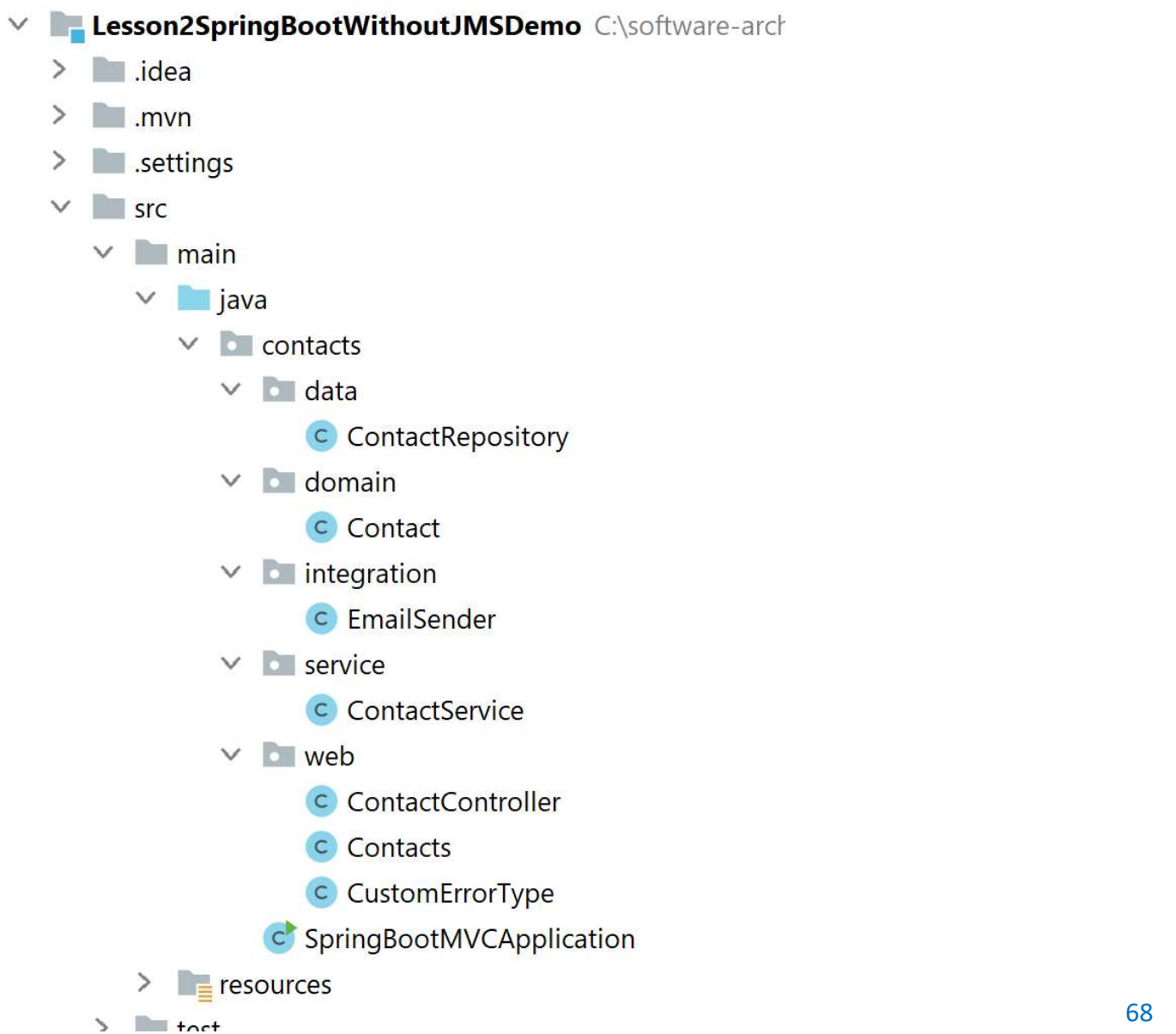
Layered architecture



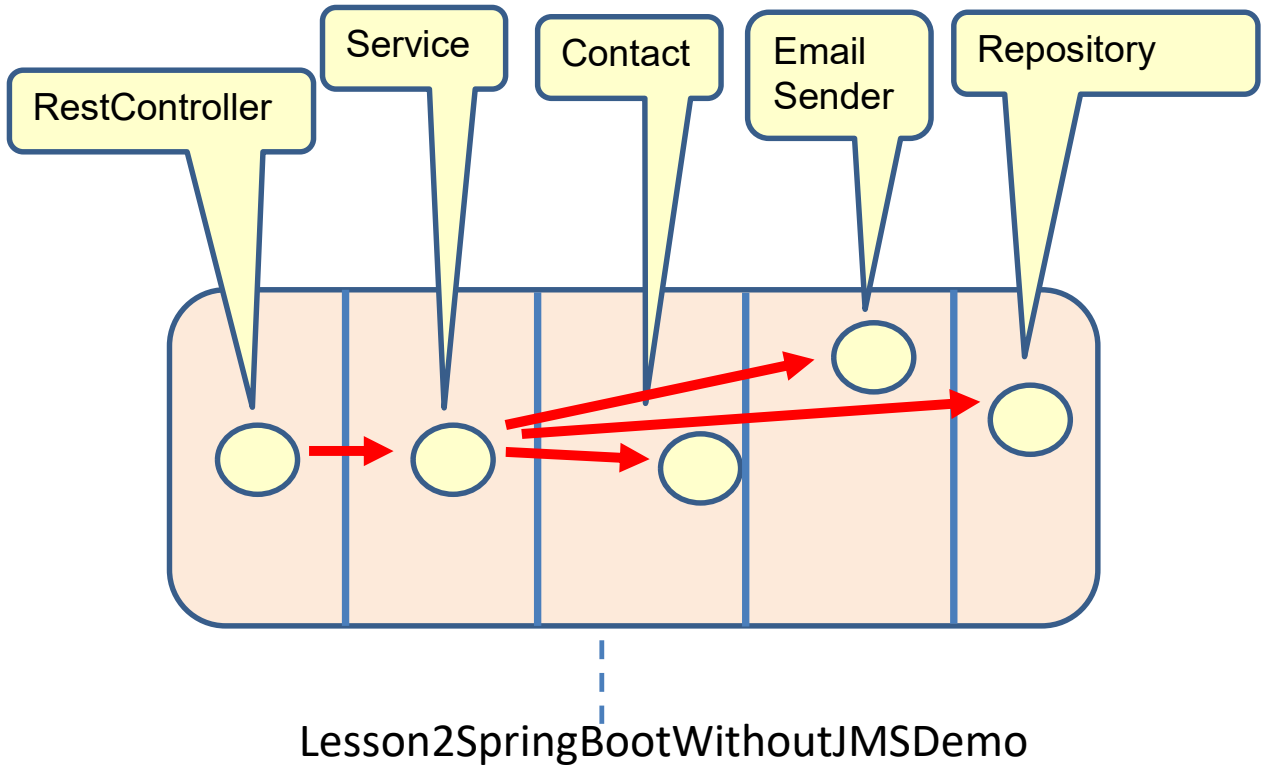
Spring Boot example



Spring Boot example



Demo application



Repository

@Repository

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

   public void save(Contact contact){
      contacts.put(contact.getFirstName(),contact);
   }

   public Contact findByFirstName(String firstName){
      return contacts.get(firstName);
   }

   public void delete(String firstName){
      contacts.remove(firstName);
   }

   public Collection<Contact> findAll(){
      return contacts.values();
   }
}
```

EmailSender

@Component

```
@Component
public class EmailSender {
   public void sendEmail (String message, String emailAddress){
      System.out.println("Send email message '"+ message+"' to"+emailAddress);
   }
}
```

Service

@Service

@Autowired

```
@Service
public class ContactService {
   @Autowired
   ContactRepository contactRepository;
   @Autowired
   EmailSender emailSender;

   public void add(Contact contact){
      contactRepository.save(contact);
      emailSender.sendEmail(contact.getEmail(), "Welcome");
   }

   public void update(Contact contact){
      contactRepository.save(contact);
   }

   public Contact findByFirstName(String firstName){
      return contactRepository.findByFirstName(firstName);
   }

   public void delete(String firstName){
      Contact contact = contactRepository.findByFirstName(firstName);
      emailSender.sendEmail(contact.getEmail(), "Good By");
      contactRepository.delete(firstName);
   }

   public Collection<Contact> findAll(){
      return contactRepository.findAll();
   }
}
```

Controller(1/2)

@RestController

@Autowired

```
@RestController
public class ContactController {
   @Autowired
   private ContactService contactService;


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

   @PostMapping("/contacts")
   public ResponseEntity<?> addContact(@RequestBody Contact contact) {
      contactService.add(contact);
      return new ResponseEntity<Contact>(contact, HttpStatus.OK);
   }

...
```

Controller(2/2)

```
@RestController
public class ContactController {
   …

   @DeleteMapping("/contacts/{firstName}")
   public ResponseEntity<?> deleteContact(@PathVariable String firstName) {
      Contact contact = contactService.findByFirstName(firstName);
      if (contact == null) {
         return new ResponseEntity<CustomErrorType>(new CustomErrorType("Contact with firstname= " + firstName + " is
                  not available"),HttpStatus.NOT_FOUND);
      }
      contactService.delete(firstName);
      return new ResponseEntity<>(HttpStatus.NO_CONTENT);
   }

   @PutMapping("/contacts/{firstName}")
   public ResponseEntity<?> updateContact(@PathVariable String firstName, @RequestBody Contact contact) {
      contactService.update(contact);
      return new ResponseEntity<Contact>(contact, HttpStatus.OK);
   }

   @GetMapping("/contacts")
   public ResponseEntity<?> getAllContacts() {
      Contacts allcontacs = new Contacts(contactService.findAll());
      return new ResponseEntity<Contacts>(allcontacs, HttpStatus.OK);
   }
}
```

Containerless deployment

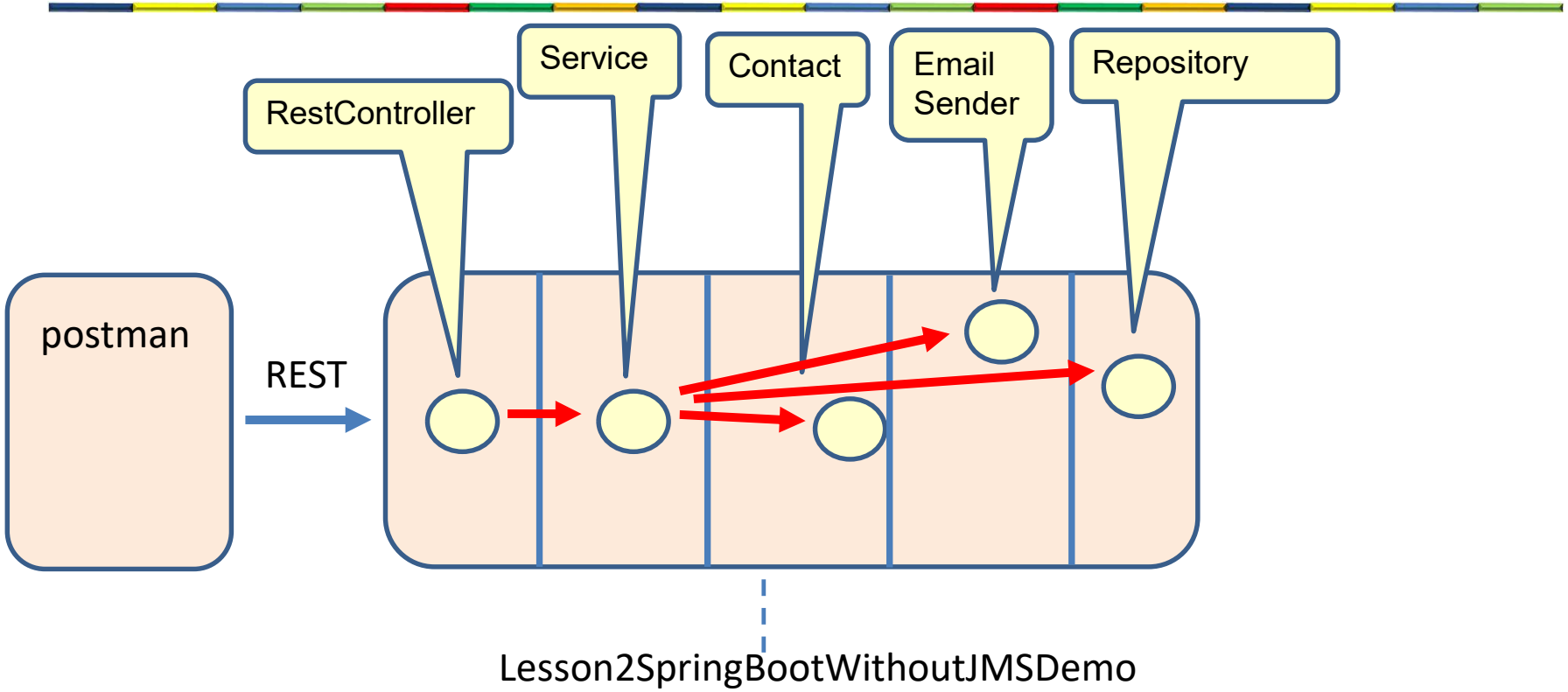
Container Deployments

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

Application Deployments

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

Demo application



Calling the REST interface: Postman

```
@PostMapping("/contacts")
public ResponseEntity<?> addContact(@RequestBody Contact contact) {
   contactService.add(contact);
   return new ResponseEntity<Contact>(contact, HttpStatus.OK);
}
```

POST request

Get the Contact class from the HTTP request message

POST

URL

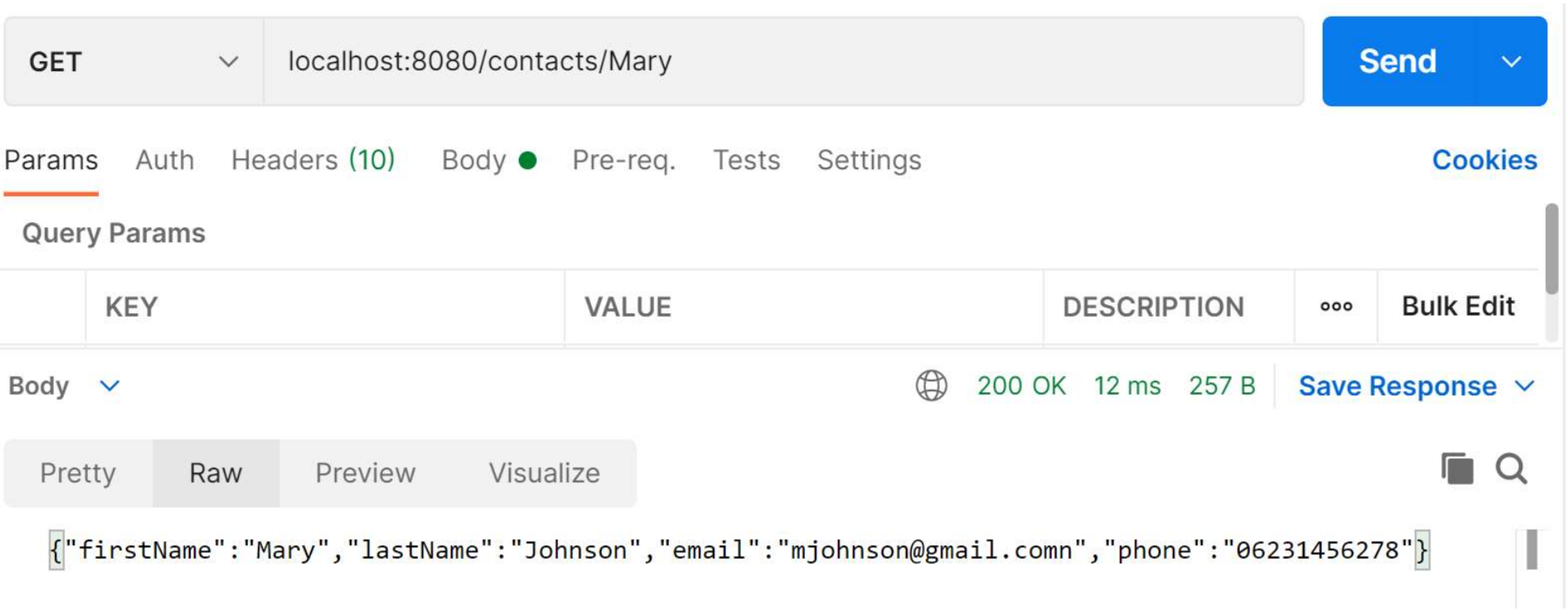
Body

JSON

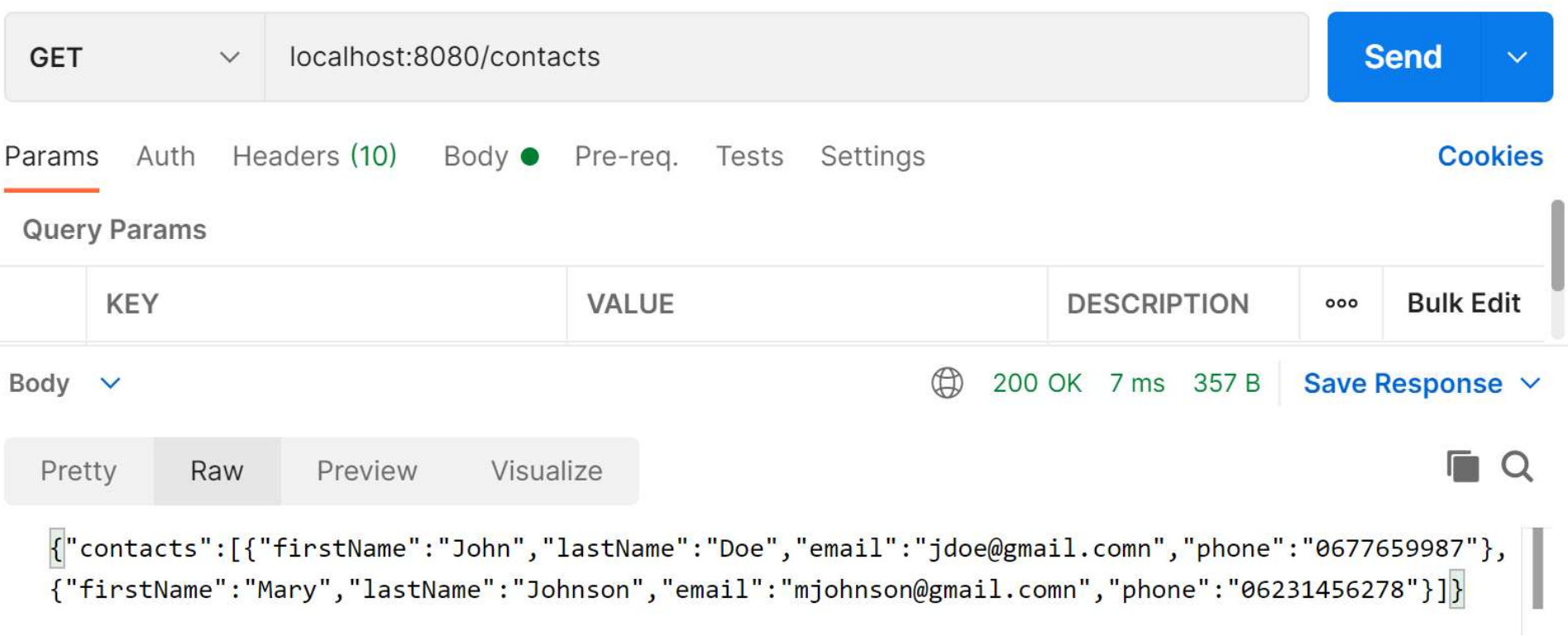
Body of the request

Body of the response

Get one contact

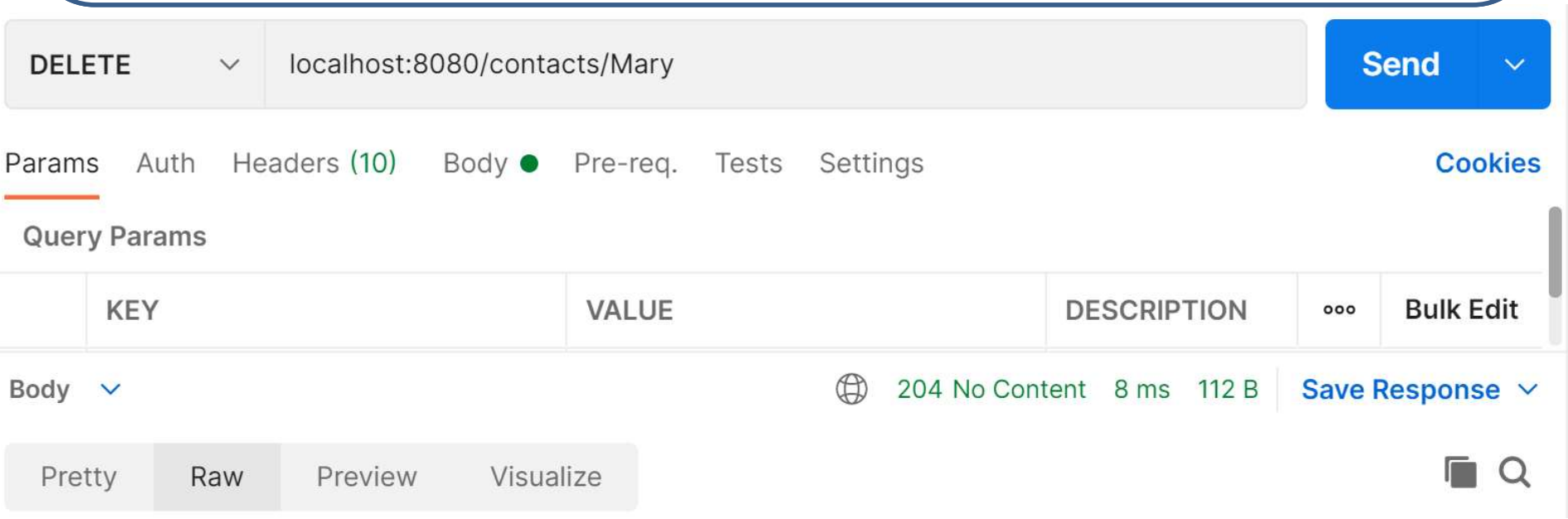


Get all contacts



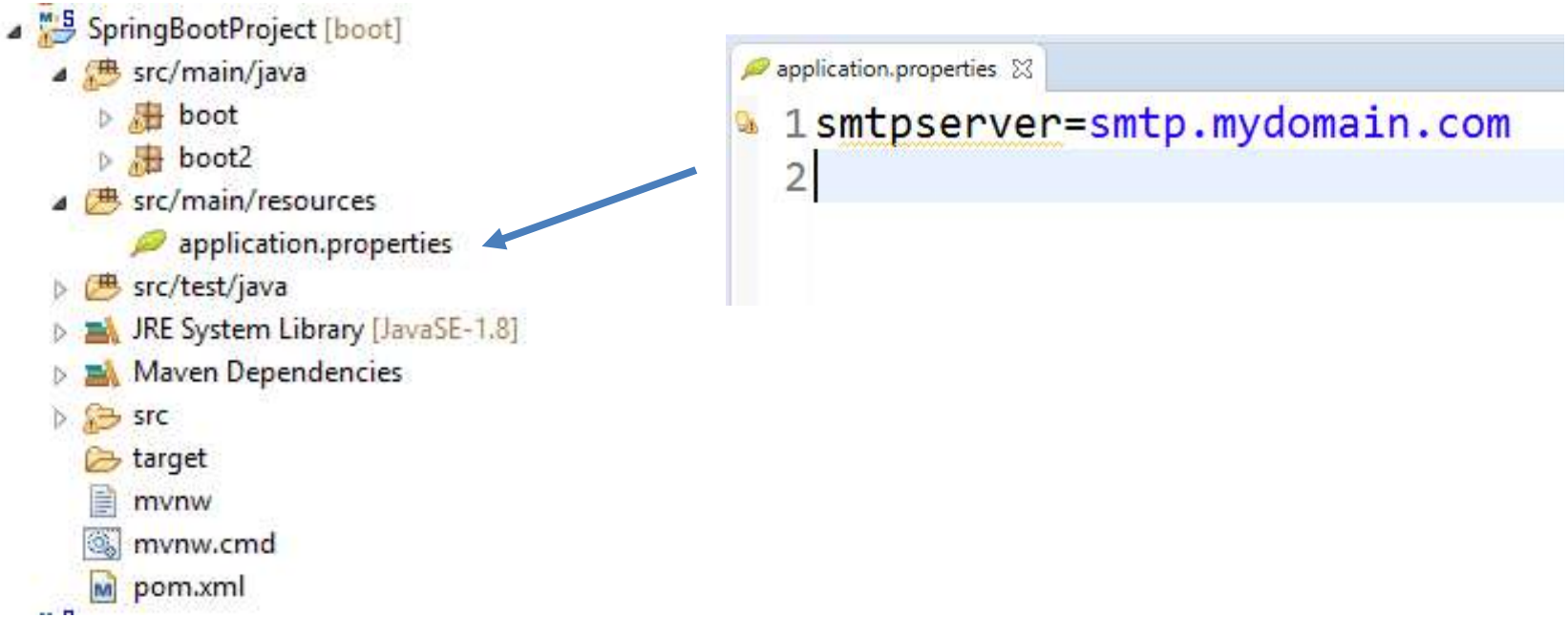
Delete a contact

```
@DeleteMapping("/contacts/{firstName}")
 public ResponseEntity<?> deleteContact(@PathVariable String firstName) {
    Contact contact = contactService.findByFirstName(firstName);
    if (contact == null) {
       return new ResponseEntity<CustomErrorType>(new CustomErrorType("Contact with firstname= " + firstName + " is
                    not available"),HttpStatus.NOT_FOUND);
    }
    contactService.delete(firstName);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
 }
```



Spring Boot configuration

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



application.properties

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

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

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

Inject the value from the properties file

application.properties

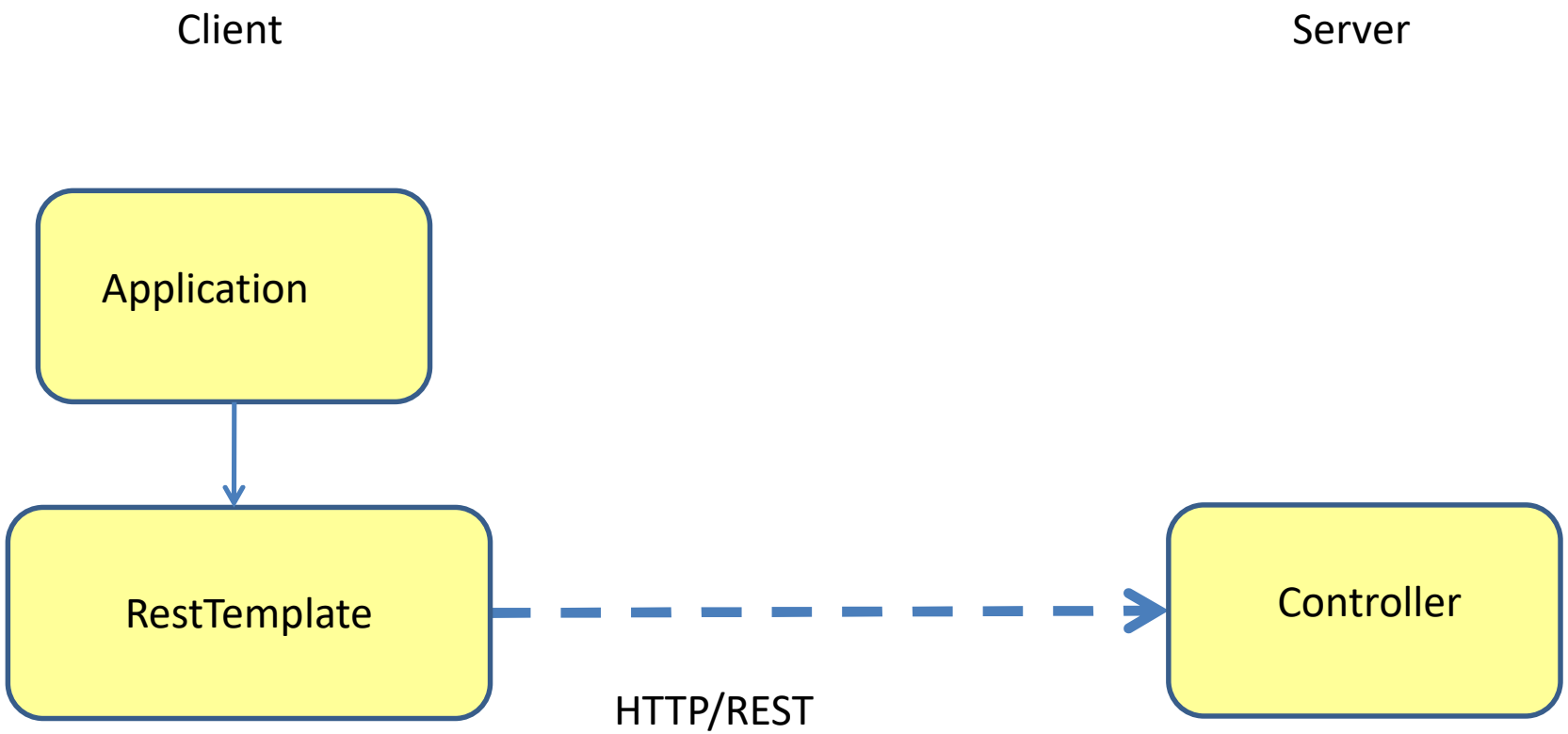
```
1 smtpserver=smtp.mydomain.com
2
```

Set the logging level in application.properties

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

SPRING BOOT REST CLIENT

Creating a REST client



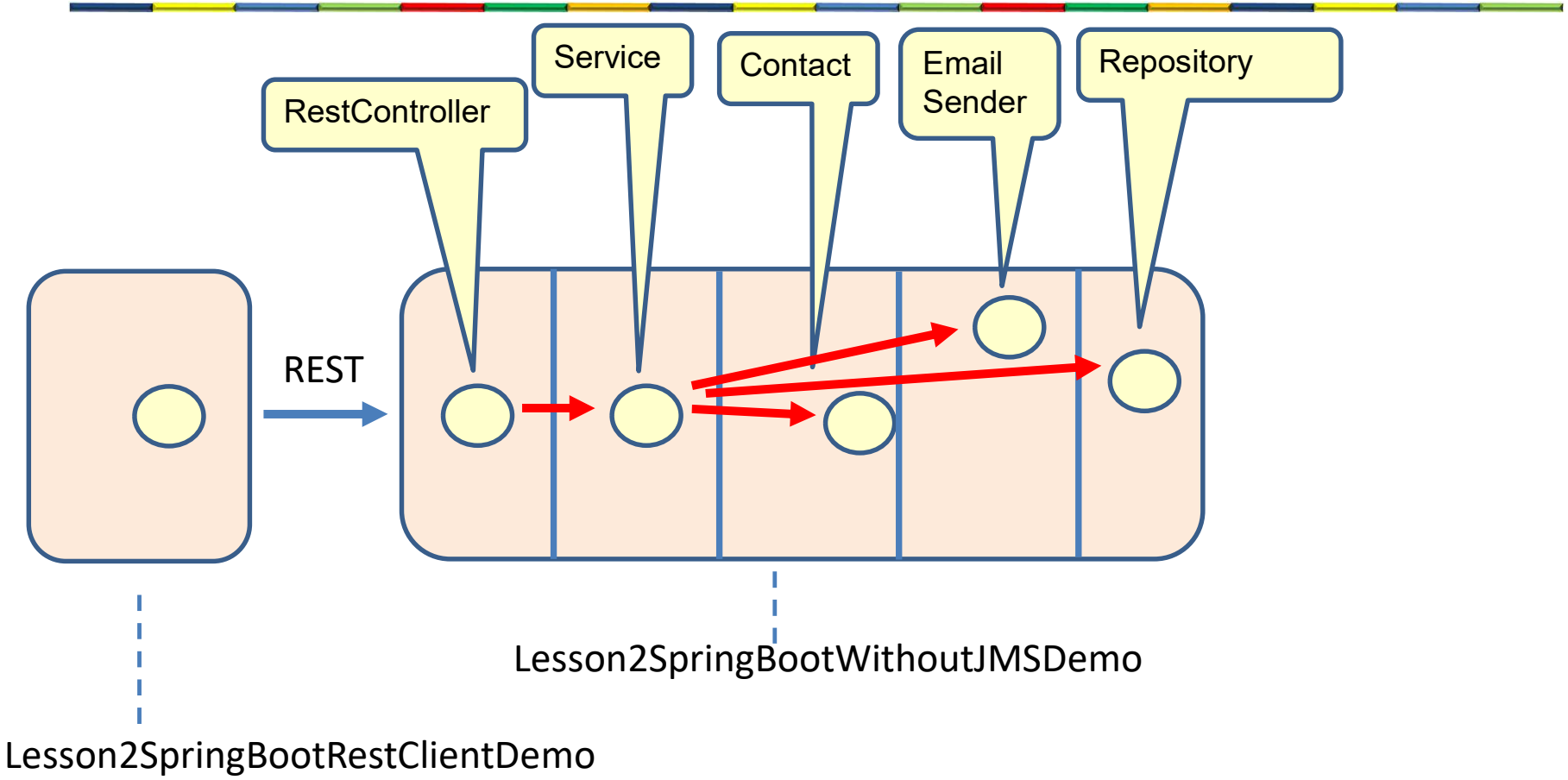
RestClient(1/2)

```
@SpringBootApplication
public class RestClientApplication implements CommandLineRunner {
  @Autowired
  private RestOperations restTemplate;

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

  @Override
  public void run(String... args) throws Exception {
    String serverUrl = "http://localhost:8080/contacts";

    // add Frank
    restTemplate.postForLocation(serverUrl, new Contact("Frank","Browns", "fbrowns@acme.com",
        "0639332163"));
    // add John
    restTemplate.postForLocation(serverUrl, new Contact("John","Doe", "jdoe@acme.com",
        "6739127563"));
    // get frank
    Contact contact= restTemplate.getForObject(serverUrl+"/{firstName}", Contact.class, "Frank");
    System.out.println("---------- get John----------------------");
    System.out.println(contact.getFirstName()+" "+contact.getLastName());
```

RestClient(2/2)

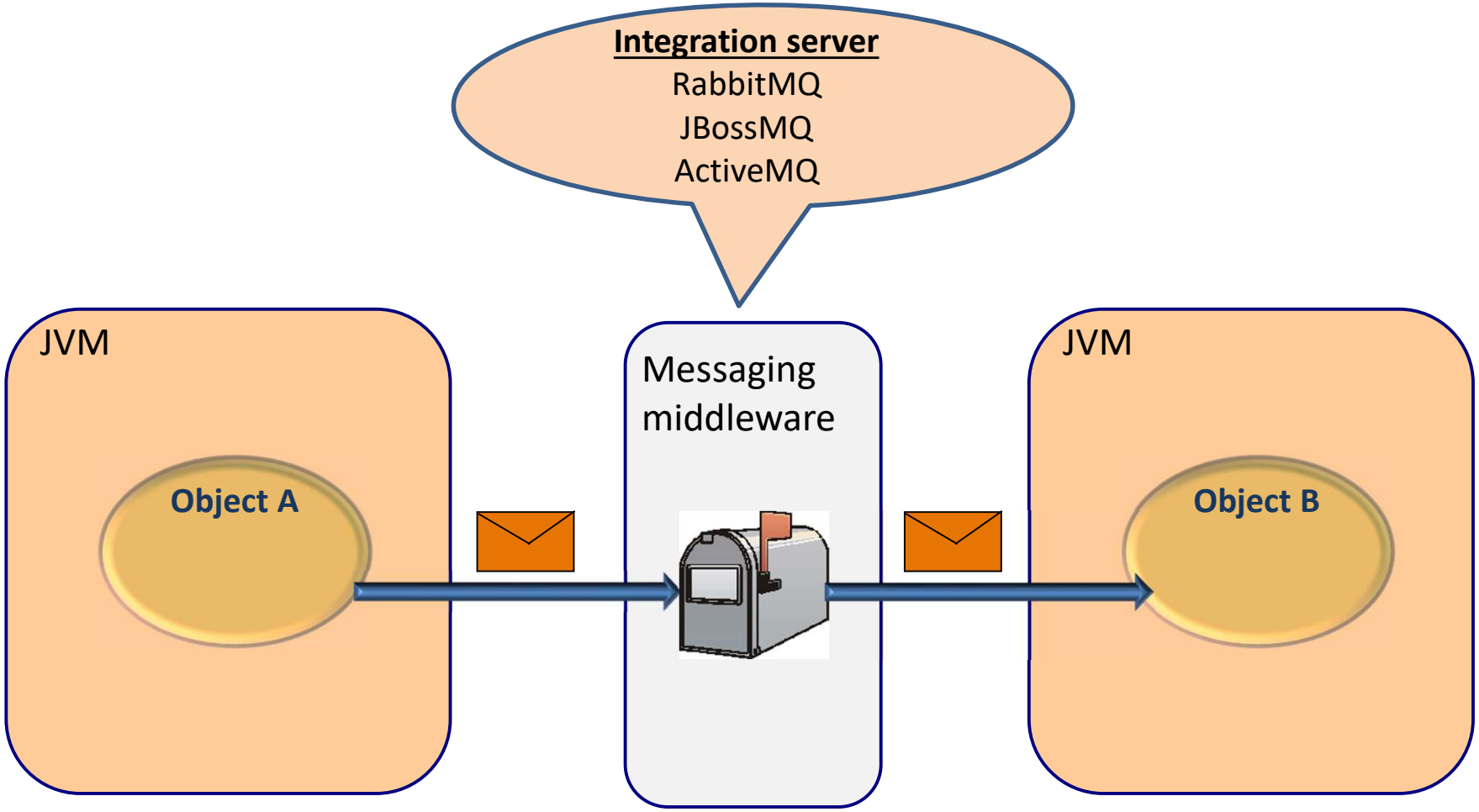
```
    // get all
    Contacts contacts= restTemplate.getForObject(serverUrl, Contacts.class);
    System.out.println("---------- get all contacts----------");
    System.out.println(contacts);

    // delete John
    restTemplate.delete(serverUrl+"/{firstName}", "John");

    // update frank
    contact.setEmail("franky@gmail.com");
    restTemplate.put(serverUrl+"/{firstName}", contact, contact.getFirstName());

    // get all
    contacts= restTemplate.getForObject(serverUrl, Contacts.class);
    System.out.println("---------- get all contacts----------");
    System.out.println(contacts);
  }


  @Bean
  RestOperations restTemplate() {
    return new RestTemplate();
  }
}
```

Demo application



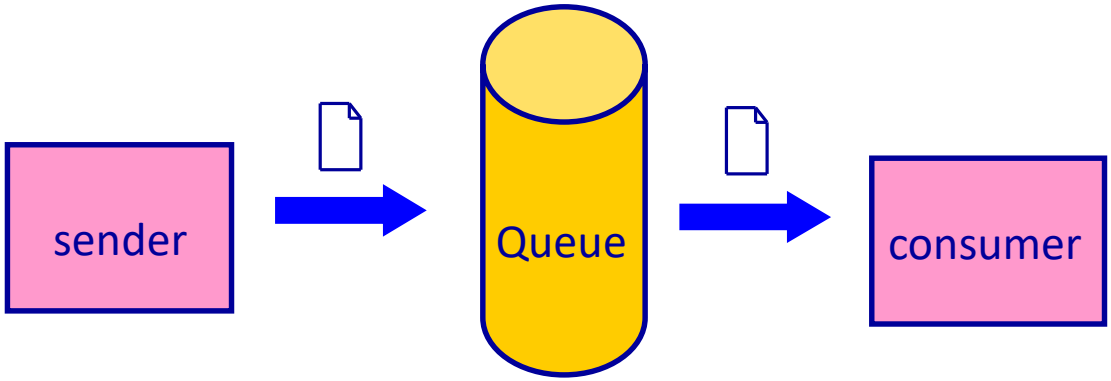
SPRING MESSAGING

Java Message Service (JMS)



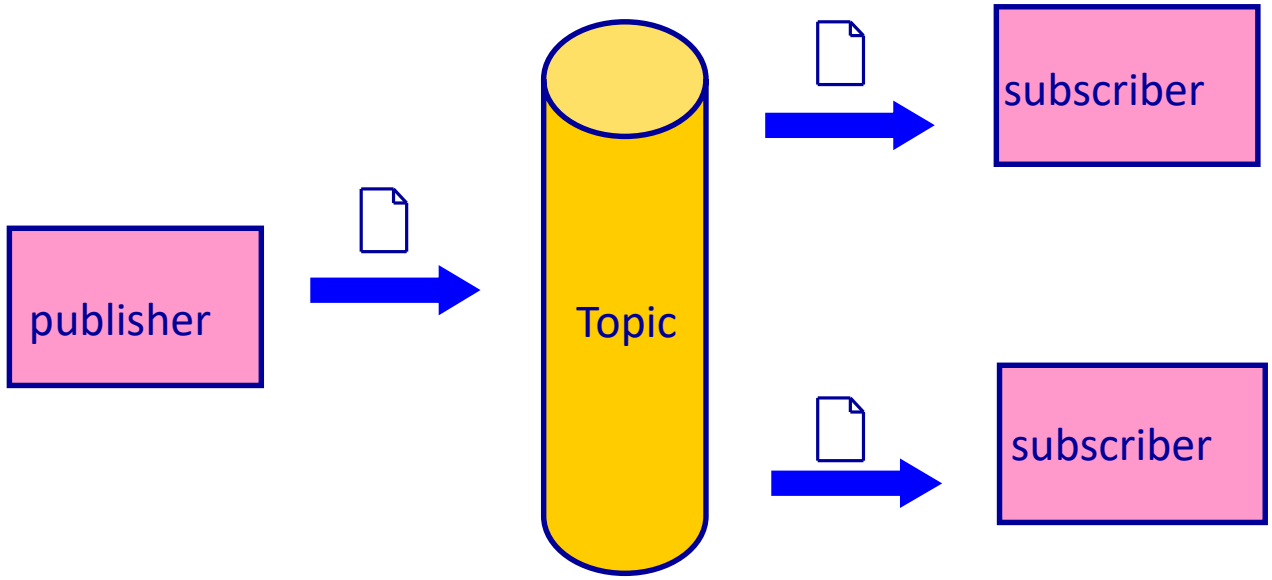
Point-To-Point (PTP)

- A dedicated consumer per Queue message



Publish-Subscribe (Pub-Sub)

- A message channel can have more than one '*consumer*'
  - Ideal for broadcasting

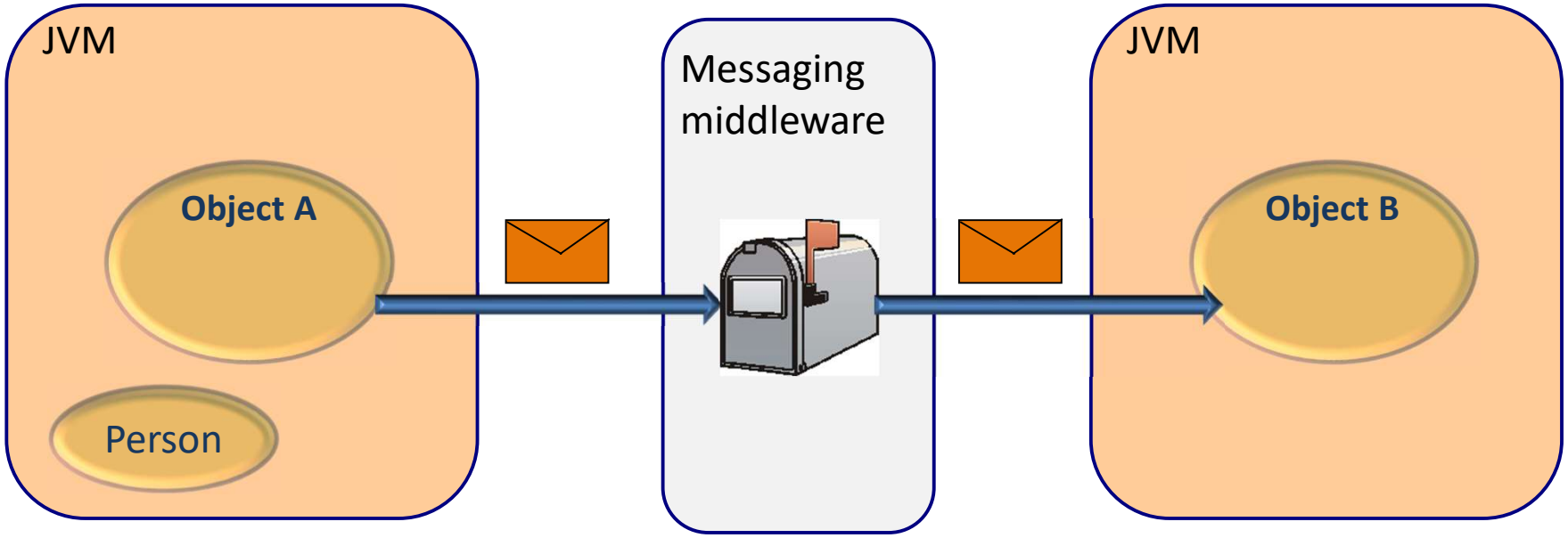


Spring ActiveMQ libraries

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

Sending an object

```
public class Person {
  private String firstName;
  private String lastName;
  ...
}
```



Sending an object

```
@SpringBootApplication
@EnableJms
public class SpringJmsPersonSenderApplication implements CommandLineRunner {
  @Autowired
  JmsTemplate jmsTemplate;

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

  @Override
  public void run(String... args) throws Exception {
    Person person = new Person("Frank", "Brown");
    //convert person to JSON string
    ObjectMapper objectMapper = new ObjectMapper();
    String personAsString = objectMapper.writeValueAsString(person);

    System.out.println("Sending a JMS message:" + personAsString);
    jmsTemplate.convertAndSend("testQueue",personAsString);
  }

}
```

Convert object to JSON

Sender application

application.properties

```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

Receiving an object

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

```
@Component
public class PersonMessageListener {

  @JmsListener(destination = "testQueue")
  public void receiveMessage(final String personAsString) {
    ObjectMapper objectMapper = new ObjectMapper();
    try {
      Person person = objectMapper.readValue(personAsString, Person.class);
      System.out.println("JMS receiver received message:" + person.getFirstName()+" "+person.getLastName());

    } catch (IOException e) {
      System.out.println("JMS receiver: Cannot convert : " + personAsString+" to a Person object");
    }
  }
}
```

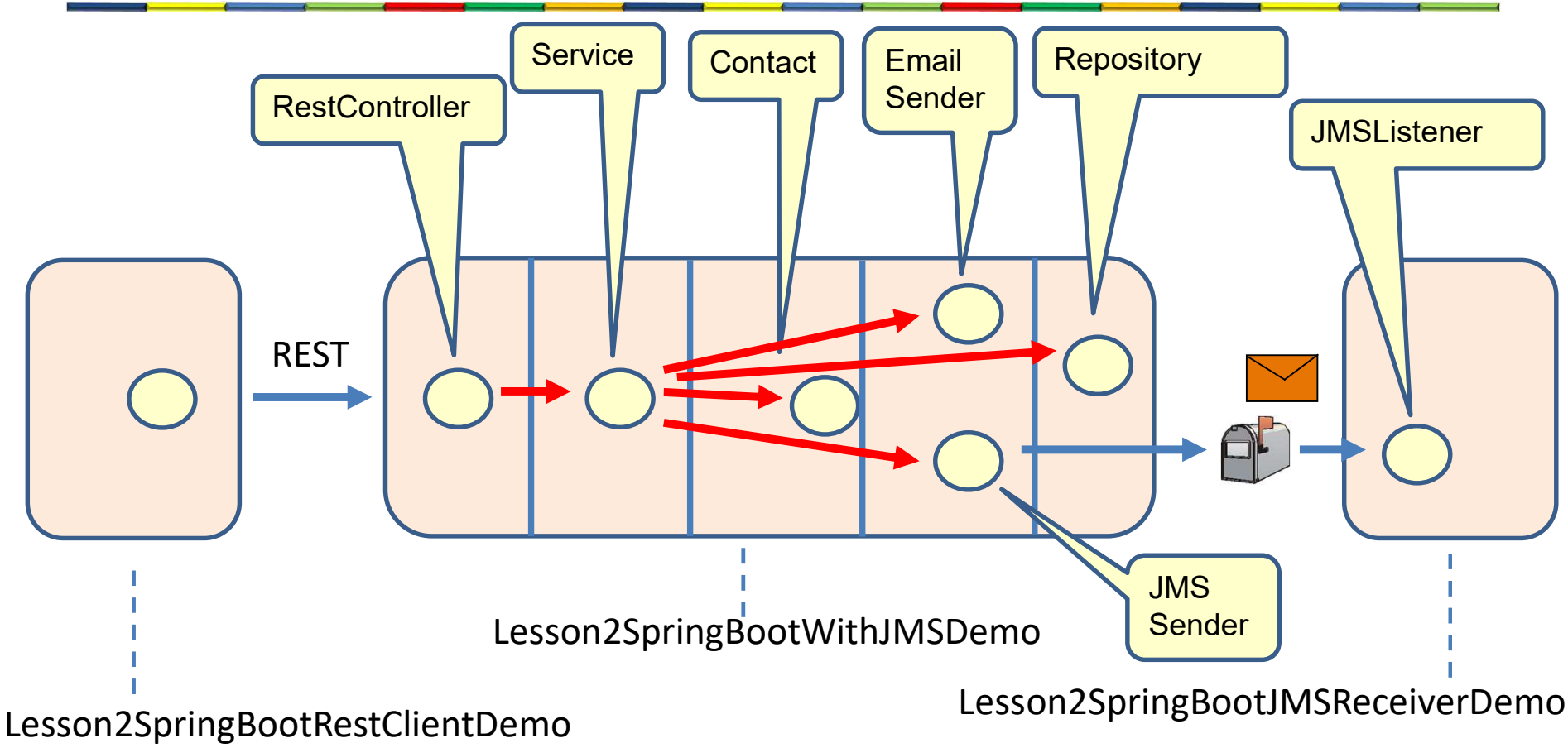
Convert JSON String to object

Receiver application

application.properties

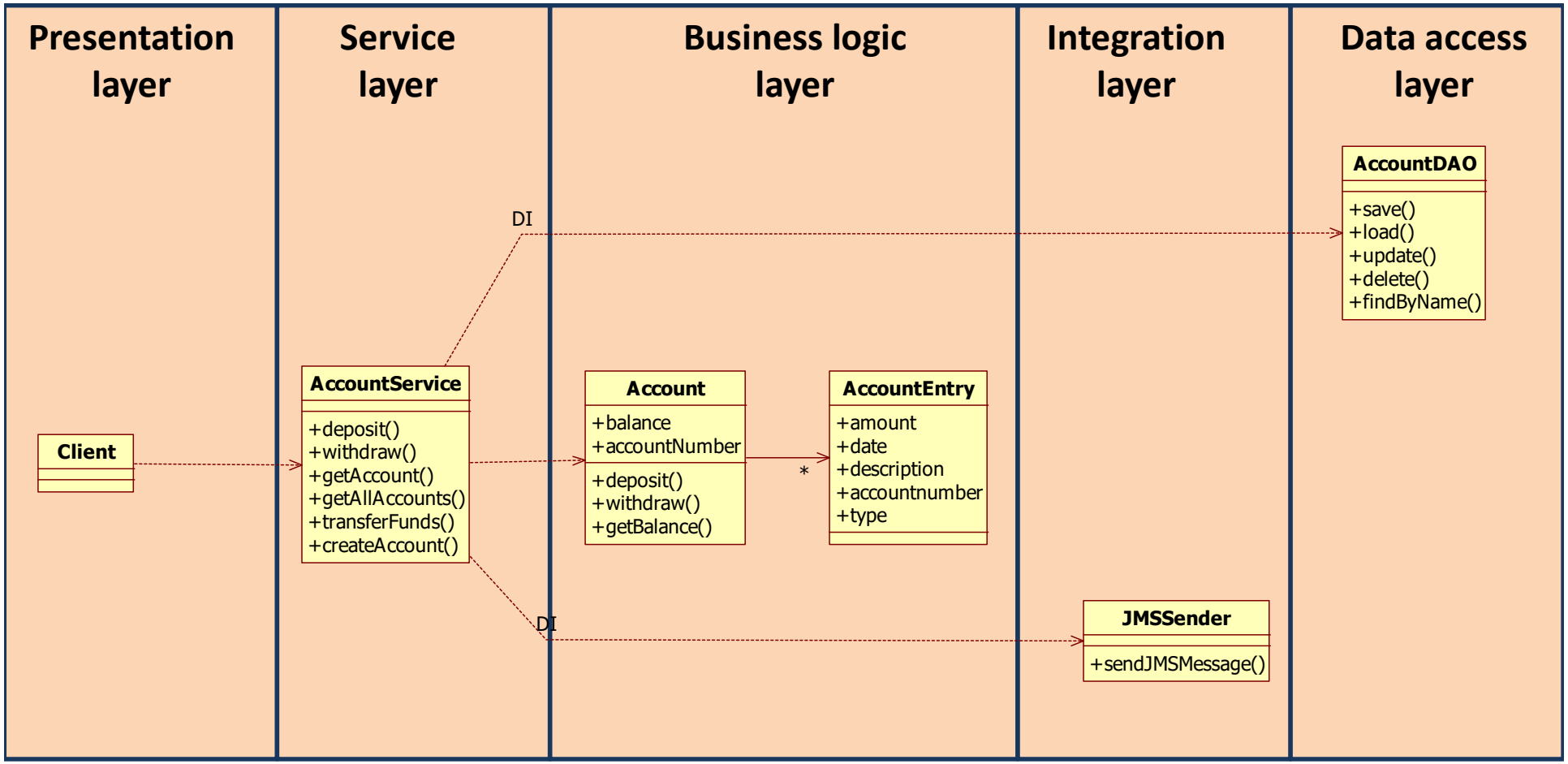
```
spring.activemq.broker-url=tcp://localhost:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

Demo application



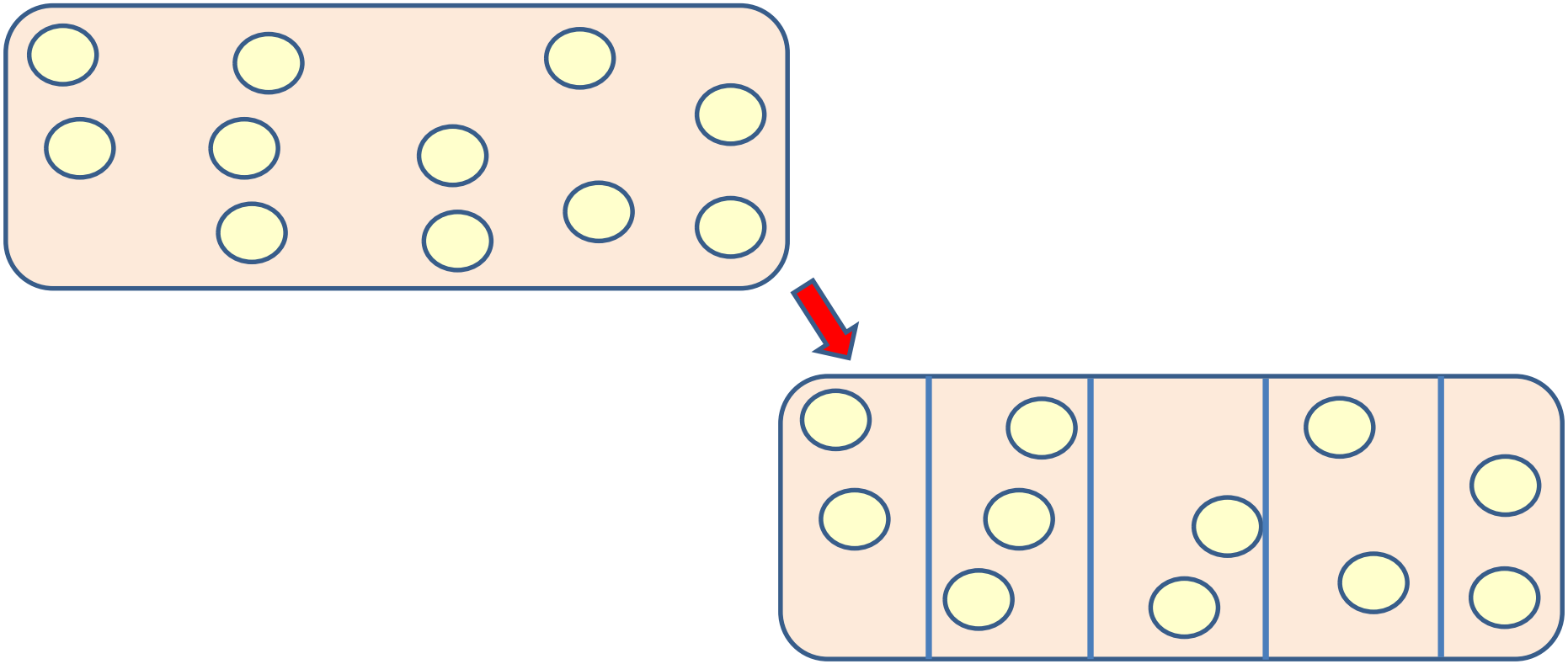
SUMMARY

Application layers

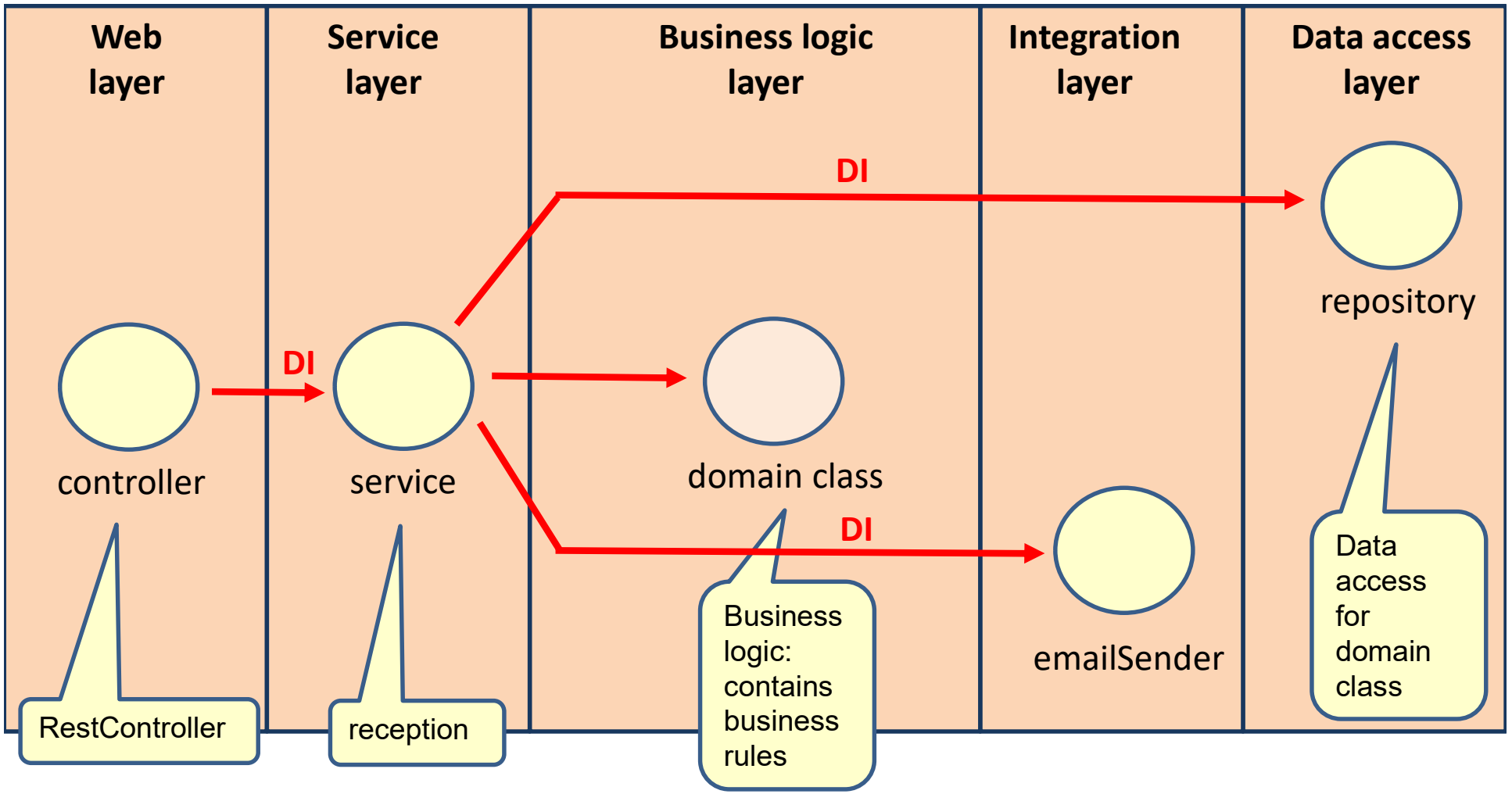


Key principle 3

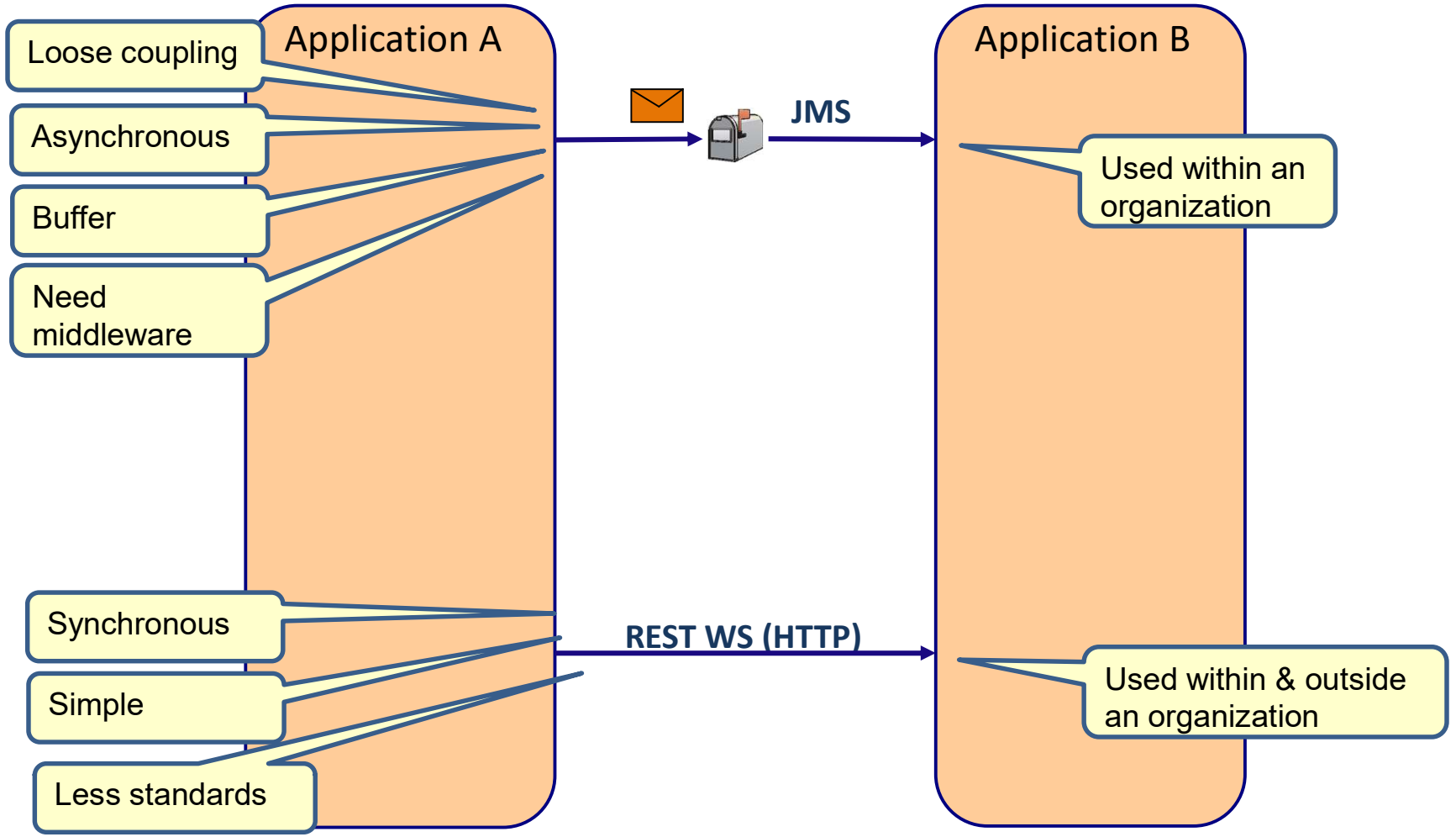
- When something becomes too complex, divide it into simpler parts



Layered architecture



Integration possibilities



Connecting the parts of knowledge with the wholeness of knowledge

1. Layering is a powerful technique to separate different aspects of a system
2. The service class is the connection point between the different layers

3. **Transcendental consciousness** is the direct experience of pure consciousness, the unified field of all the laws of nature.
4. **Wholeness moving within itself:** In unity consciousness, one appreciates the inherent underlying unity that underlies all the diversity of creation.