



Lesson 19

17.03.2022

```
public class Ex1 {  
    public static void main(String[] args) {  
        boolean flag1 = "Java" == "Java"  
            .replace( 'J' , 'J' );  
        boolean flag2 = "Java" == "Java"  
            .replace( 'J' , 'J' );  
        System.out.println(flag1);  
        System.out.println(flag2);  
        System.out.println(flag1 && flag2);  
    }  
}
```

```
public class Ex2 {  
    public static void main(String[] args) {  
        String s1 = new String( "Java" );  
        String s2 = "JaVa" ;  
        String s3 = "JaVa" ;  
        String s4 = "Java" ;  
        String s5 = "Java" ;  
        int i = 1 ;  
        Integer j = 122 ;  
        Integer k = 129 ;  
        Integer m = 128 ;  
    }  
}
```

```
public class Ex3 {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int i = 0; i < 7; i++) {  
            if (i == 4)  
                break;  
            else  
                continue;  
            sum += i;  
        }  
        System.out.println(sum);  
    }  
}
```

```
public class Ex4 {  
    public static void main(String[] args) {  
        String str = "BEVERAGE" ;  
        String [] arr = str.split( "E" , 3 );  
        System. out .println(String. join ( "." , arr));  
    }  
}
```

```
public class Ex5 {  
    public static void main(String[] args) {  
        Boolean [] arr = new Boolean[ 2 ];  
        List<Boolean> list = new ArrayList<>();  
        list.add(arr[ 0 ]);  
        list.add(arr[ 1 ]);  
        if (list.remove( 0 )) {  
            list.remove( 1 );  
        }  
        System.out.println(list);  
    }  
}
```

S. O. L. I. D.

Principles in **React Application** Architecture

Single
Responsibility

Open-closed
Principle

Liskov
substitution
Principle

Interface
Segregation
Principle

Dependency
Inversion
Principle



Принцип единственной ответственности (Single Responsibility Principle)

Существует лишь одна причина, приводящая к изменению класса.

Один класс должен решать только какую-то одну задачу. Он может иметь несколько методов, но они должны использоваться лишь для решения общей задачи. Все методы и свойства должны служить одной цели. Если класс имеет несколько назначений, его нужно разделить на отдельные классы.

Принцип открытости/закрытости (Open-closed Principle)

Программные сущности должны быть открыты для расширения, но закрыты для модификации.

Программные сущности (классы, модули, функции и прочее) должны быть расширяемыми без изменения своего содержимого. Если строго соблюдать этот принцип, то можно регулировать поведение кода без изменения самого исходника.



Принцип подстановки Барбары Лисков (Liskov Substitution Principle)

Функции, использующие указатели ссылок на базовые классы, должны уметь использовать объекты производных классов, даже не зная об этом.

Попросту говоря: подкласс/производный класс должен быть взаимозаменяем с базовым/родительским классом.

Значит, любая реализация абстракции (интерфейса) должна быть взаимозаменяемой в любом месте, в котором принимается эта абстракция. По сути, когда мы используем в коде интерфейсы, то используем контракт не только по входным данным, принимаемым интерфейсом, но и по выходным данным, возвращаемым разными классами, реализующими этот интерфейс. В обоих случаях данные должны быть одного типа.

Принцип разделения интерфейса (Interface Segregation Principle)

Нельзя заставлять клиента реализовать интерфейс, которым он не пользуется.

Это означает, что нужно разбивать интерфейсы на более мелкие, лучше удовлетворяющие конкретным потребностям клиентов.

Как и в случае с принципом единственной ответственности, цель принципа разделения интерфейса заключается в минимизации побочных эффектов и повторов за счёт разделения ПО на независимые части.



Принцип инверсии зависимостей (Dependency Inversion Principle)

Высокоуровневые модули не должны зависеть от низкоуровневых. Оба вида модулей должны зависеть от абстракций.

Абстракции не должны зависеть от подробностей. Подробности должны зависеть от абстракций.

Проще говоря: зависьте от абстракций, а не от чего-то конкретного.

Применяя этот принцип, одни модули можно легко заменять другими, всего лишь меняя модуль зависимости, и тогда никакие переменные в низкоуровневом модуле не повлияют на высокоуровневый.



Что такое веб-сервисы?

Прежде всего, веб-сервисы (или веб-службы) — это технология. И как и любая другая технология, они имеют довольно четко очерченную среду применения.

Если посмотреть на веб-сервисы в разрезе стека сетевых протоколов, мы увидим, что это, в классическом случае, не что иное, как еще одна надстройка поверх протокола HTTP.

По сути, веб-сервисы — это реализация абсолютно четких интерфейсов обмена данными между различными приложениями, которые написаны не только на разных языках, но и распределены на разных узлах сети.

Именно с появлением веб-сервисов развилась идея SOA — сервис-ориентированной архитектуры веб-приложений (Service Oriented Architecture).

Протоколы веб-сервисов


На сегодняшний день наибольшее распространение получили следующие протоколы реализации веб-сервисов:

SOAP (Simple Object Access Protocol) — по сути это тройка стандартов SOAP/WSDL/UDDI

REST (Representational State Transfer)

XML-RPC (XML Remote Procedure Call)

На самом деле, SOAP произошел от XML-RPC и является следующей ступенью его развития. В то время как REST — это концепция, в основе которой лежит скорее архитектурный стиль, нежели новая технология, основанный на теории манипуляции объектами CRUD (Create Read Update Delete) в контексте концепций WWW.

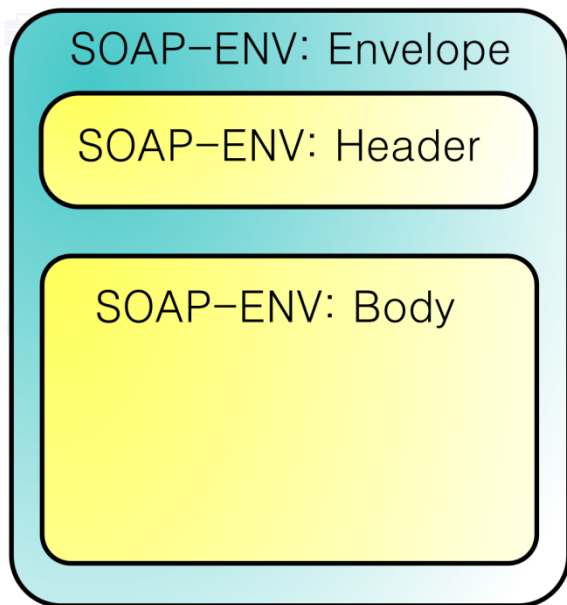


SOAP (от англ. Simple Object Access Protocol — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде. Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур (RPC). Сейчас протокол используется для обмена произвольными сообщениями в формате XML, а не только для вызова процедур. Официальная спецификация последней версии 1.2 протокола никак не расшифровывает название SOAP. SOAP является расширением протокола XML-RPC.

SOAP может использоваться с любым протоколом прикладного уровня: SMTP, FTP, HTTP, HTTPS и др. Однако его взаимодействие с каждым из этих протоколов имеет свои особенности, которые должны быть определены отдельно. Чаще всего SOAP используется поверх HTTP.



Структура протокола



Envelope – Корневой элемент, который определяет сообщение и пространство имен, использованное в документе.

Header – Содержит атрибуты сообщения, например: информация о безопасности или о сетевой маршрутизации.

Body – Содержит сообщение, которым обмениваются приложения.

Fault – Необязательный элемент, который предоставляет информацию об ошибках, которые произошли при обработке сообщений

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>12345</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productID>12345</productID>
        <productName>Стакан граненый</productName>
        <description>Стакан граненый. 250 мл.</description>
        <price>9.95</price>
        <currency>
          <code>840</code>
          <alpha3>USD</alpha3>
          <sign>$</sign>
          <name>US dollar</name>
          <accuracy>2</accuracy>
        </currency>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```




REST

(REpresentational State Transfer) — это архитектура, т.е. принципы построения распределенных гипермедиа систем, того что другими словами называется World Wide Web, включая универсальные способы обработки и передачи состояний ресурсов по HTTP

Автор идеи и термина Рой Филдинг 2000г.



Что нам дает REST подход

- ✓ Масштабируемости взаимодействия компонентов системы (приложения)
- ✓ Общность интерфейсов
- ✓ Независимое внедрение компонентов
- ✓ Промежуточные компоненты, снижающие задержку, усиливающие безопасность

Когда использовать REST?

- ✓ Когда есть ограничение пропускной способности соединения
- ✓ Если необходимо кэшировать запросы
- ✓ Если система предполагает значительное масштабирование
- ✓ В сервисах, использующих AJAX

Преимущества REST:

- Отсутствие дополнительных внутренних прослоек, что означает передачу данных в том же виде, что и сами данные. Т.е. данные не оборачиваются в [XML](#), как это делает [SOAP](#) и [XML-RPC](#).
- Каждая единица информации (**ресурс**) однозначно определяется URL — это значит, что URL по сути является первичным ключом для единицы данных. Причем совершенно не имеет значения, в каком формате находятся данные по адресу — это может быть и HTML, и jpeg, и документ Microsoft Word.
- Как происходит управление информацией ресурса — это целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол конечно же [HTTP](#). Для HTTP действие над данными задается с помощью методов : GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия [CRUD](#) (Create-Read-Update-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST.



RESTful критерии:

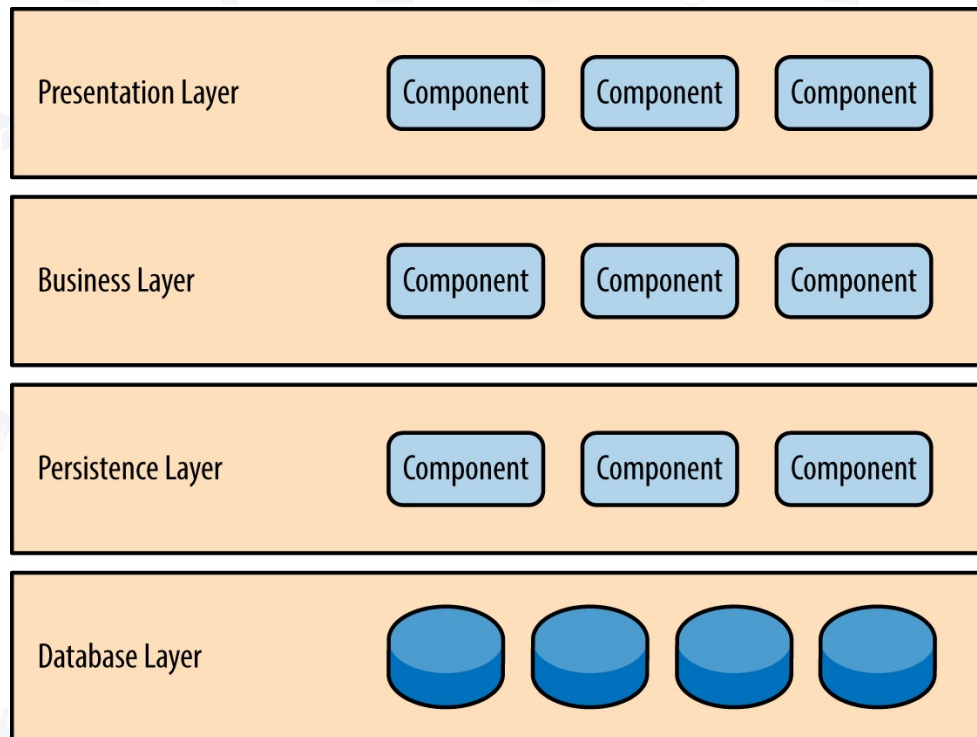
Client-Server. Система должна быть разделена на клиентов и на серверов. Разделение интерфейсов означает, что, например, клиенты не связаны с хранением данных, которое остается внутри каждого сервера, так что мобильность кода клиента улучшается. Серверы не связаны с интерфейсом пользователя или состоянием, так что серверы могут быть проще и масштабируемы. Серверы и клиенты могут быть заменяемы и разрабатываться независимо, пока интерфейс не изменяется.


Stateless. Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и если необходимо, идентификации клиента.

Cache. Каждый ответ должен быть отмечен является ли он кэшируемым или нет, для предотвращения повторного использования клиентами устаревших или некорректных данных в ответ на дальнейшие запросы.

Uniform Interface. Единый интерфейс определяет интерфейс между клиентами и серверами. Это упрощает и отделяет архитектуру, которая позволяет каждой части развиваться самостоятельно.

5 **.Layered System.** В REST допускается разделить систему на иерархию слоев но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя. Например:





HTTP метод GET используется **для получения (или чтения)** представления ресурса. В случае “удачного” (или не содержащего ошибок) адреса, GET возвращается представление ресурса в формате XML или JSON в сочетании с кодом состояния HTTP 200 (OK). В случае наличия ошибок обычно возвращается код 404 (NOT FOUND) или 400 (BAD REQUEST).


[GET] <http://www.example.com/api/v1.0/users>

(вернуть список пользователей)

[GET] <http://www.example.com/api/v1.0/users/12345>

(вернуть данные о пользователе с id 12345)

[GET] <http://www.example.com/api/v1.0/users/12345/orders>



HTTP метод PUT обычно используется для предоставления возможности обновления ресурса. Тело запроса при отправке PUT-запроса к существующему ресурсу URI должно содержать обновленные данные оригинального ресурса (полностью, или только обновляемую часть).


При успешном обновлении посредством выполнения PUT запроса возвращается код 200 PUT не безопасная операция, так как вследствие ее выполнения происходит модификация (или создание) экземпляров ресурса на стороне сервера, но этот метод идемпотентен. Другими словами, создание или обновление ресурса посредством отправки PUT запроса — ресурс не исчезнет, будет располагаться там же, где и был при первом обращении, а также, многократное выполнение одного и того же PUT запроса не изменит общего состояния системы

[PUT] <http://www.example.com/api/v1.0/users/12345>

(обновить данные пользователя с id 12345)

[PUT] <http://www.example.com/api/v1.0/users/12345/orders/98765>

(обновить данные заказа с id 98765 для пользователя с id 12345)



HTTP метод POST запрос наиболее часто используется **для создания новых ресурсов**. На практике он используется для создания вложенных ресурсов. Другими словами, при создании нового ресурса, POST запрос отправляется к родительскому ресурсу и, таким образом, сервис берет на себя ответственность на установление связи создаваемого ресурса с родительским ресурсом, назначение новому ресурсу ID и т.п.

При успешном создании ресурса возвращается HTTP код 201, а также в заголовке `Location` передается адрес созданного ресурса.

POST не является безопасным или идемпотентным запросом. Потому рекомендуется его использование для не идемпотентных запросов. В результате выполнения идентичных POST запросов предоставляются сильно похожие, но не идентичные данные.

[POST] `http://www.example.com/api/v1.0/customers`

(создать новый ресурс в разделе customers)

[POST] `http://www.example.com/api/v1.0/customers/12345/orders`

(создать заказ для ресурса с id 12345)



HTTP метод DELETE используется **для удаления ресурса**, идентифицированного конкретным URI (ID).

При успешном удалении возвращается 200 (OK) код HTTP, совместно с телом ответа, содержащим данные удаленного ресурса. Также возможно использование HTTP кода 204 (NO CONTENT) без тела ответа. Если вы выполняете DELETE запрос к ресурсу, он удаляется. Повторный DELETE запрос к ресурсу закончится также: ресурс удален.

Тем не менее, существует предостережение об идемпотентности DELETE. Повторный DELETE запрос к ресурсу часто сопровождается 404 (NOT FOUND) кодом HTTP по причине того, что ресурс уже удален (например из базы данных) и более не доступен. Это делает DELETE операцию не идемпотентной, но это общепринятый компромисс на тот случай, если ресурс был удален из базы данных, а не помечен, как удаленный.

[DELETE] <http://www.example.com/api/v1.0/customers/12345>

(удалить из customers ресурс с id 12345)

[DELETE] <http://www.example.com/api/v1.0/customers/12345/orders/21>

(удалить у ресурса с id 12345 заказ с id 21)



Коды состояний HTTP

1xx: Information

100: Continue

2xx: Success

200: OK

201: Created

202: Accepted

204: No Content

3xx: Redirect

301: Moved Permanently

307: Temporary Redirect

4xx: Client Error

400: Bad Request

401: Unauthorized

403: Forbidden

404: Not Found

5xx: Server Error

500: Internal Server Error

501: Not Implemented

502: Bad Gateway

503: Service Unavailable

504: Gateway Timeout

REST и SOAP

REST и SOAP на самом деле не сопоставимы. REST — это архитектурный стиль. SOAP — это формат обмена сообщениями. Давайте сравним популярные реализации стилей REST и SOAP.

Пример реализации RESTful: JSON через HTTP

Пример реализации SOAP: XML поверх SOAP через HTTP

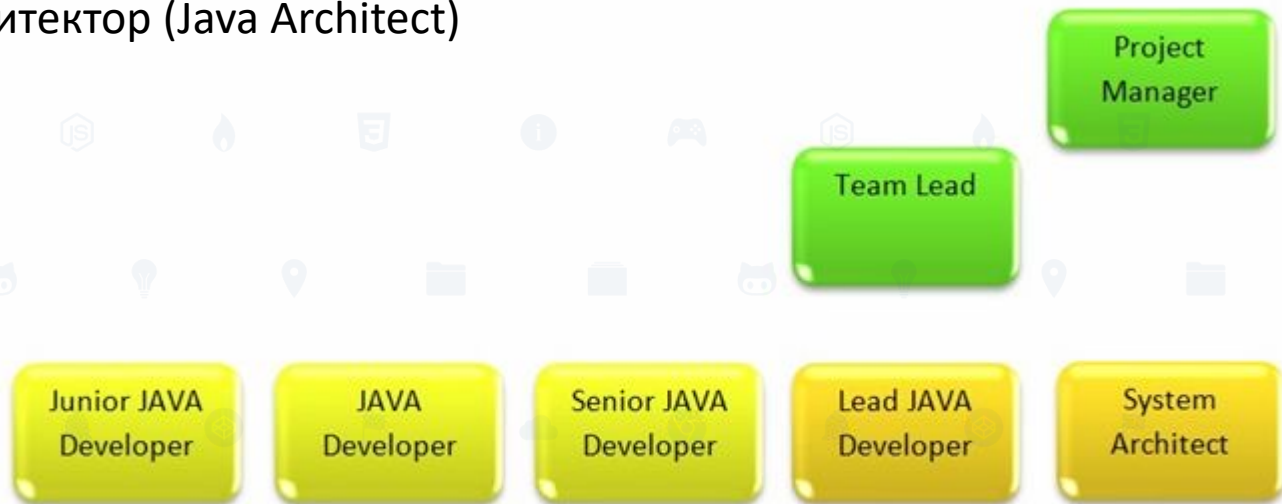
На верхнем уровне SOAP ограничивает структуры ваших сообщений, тогда как REST — это архитектурный подход, ориентированный на использование HTTP в качестве транспортного протокола.

Специфика SOAP — это формат обмена данными. С SOAP это всегда SOAP-XML, который представляет собой XML

Специфика REST — использование HTTP в качестве транспортного протокола. Он подразумевает наилучшее использование функций, предоставляемых HTTP — методы запросов, заголовки запросов, ответы, заголовки ответов и т. д.

Уровни Java разработчика


- Интерн (Intern)
- Младший разработчик (Java Junior)
- Разработчик (Java Middle)
- Старший разработчик (Senior)
- Технический руководитель / Глава команды (Tech Lead / Team Lead)
- Архитектор (Java Architect)





Технологии

- Знание ООП
- Java Core
- Java EE / Spring
- Опыт работы с базами данных (SQL, NoSQL)
- ORM: JPA, Hibernate
- Maven, Gradle
- Паттерны проектирования
- JUnit
- Log4J, Slf4J и т.д.
- Git / Mercurial
- Tomcat / Jboss
- HTML/CSS
- JavaScript
- Cache
- Message Broker
- Методологии SCRUM, Waterfall и т.д.



Интерн (Intern)

Решаемые задачи :

Грамотное изучение фреймворков.

Создание простых, чаще всего, не коммерческих приложений для тренировки.

Изучение технологий, которые используются в проекте.

Заполнение пробелов в знаниях.

Цели компании:

Воспитать Java Junior'а за 3-6 месяцев и “полуфабриката”

Потратить минимум денег на обучение специалиста

Интерны, как правило, не получают оплаты, а если и получают, то она находится на уровне стипендии студента.

Интерн не приносит доход компании, она вкладывает деньги в разработчика и инвестирует в будущее.

Потратить минимум времени своих специалистов

Цели интерна:

Стать Java Junior

Получить практический опыт работы в проекте серьёзного уровня.

Получить предложение о работе.



Java Junior

Решаемые задачи:

Создание простых классов.

Исправить некорректное отображение чего-либо.

Добавить уже имеющийся функционал в другое место.

Исправление небольших багов.

Цели компании:

Возложить на специалиста выполнение рутинных и простых задач для того, чтобы освободить более квалифицированных программистов.

Постараться как можно дольше оставить человека в компании, чтобы получить полноценного разработчика уровня “мидл”.

Цели младшего разработчика:

Получить опыт работы в реальном коммерческом проекте.

Научиться работать в команде разработчиков.

Учиться работать эффективно.



Java Middle

Решаемые задачи:

Работы на уровне модулей.

Написание функционала в соответствии со спецификацией.

Покрытие проекта тестами.

Цели компании:

Выполнение основной работы.

Прибыль.

Цели разработчика:

Углубить собственные знания используемых технологий.

Заполнить пробелы в знаниях.

Научиться активно и грамотно работать в команде.

“Вырасти” до уровня Senior разработчика.



Senior

Решаемые задачи:

Выполнение задач любой сложности.

Работы на уровне фреймворков.

Работа с клиентом.

Цели компании:

Иметь специалиста, способного решать любые задачи в рамках проекта.

Генерирование максимальной прибыли.

Старшие разработчики приносят компании максимальную прибыль.

Цели старшего разработчика:

В совершенстве освоить инструменты работы.

Решить куда идти дальше.

С этого момента программист решает, хочет ли он дальше расти как специалист, либо идти в менеджмент (PM и т.д.).