

# Лекция 2

Форматы передачи данных  
Резервное копирование

# Содержание лекции

- Форматы передачи данных
- CSV
- XML
- JSON
- YAML
- Подходы к резервному копированию

# Форматы передачи данных

- нужны для обмена информацией между системами
- ..или внутри одной системы:  
импорт/экспорт данных, настройки
- протокол взаимодействия
- html тоже можно рассматривать как формат передачи данных

# Форматы передачи данных

Сейчас особенно распространены:

- CSV
- XML
- JSON
- YAML

# CSV

- Comma-Separated Values — значения, разделённые запятыми
- разделителем может быть не только ',', но и ';', '|', табуляция, и прочее (хотя не все называют эти файлы csv, некоторые называют tsv и пр.)
- используется, чтобы хранить таблицы в текстовых файлах
- импорт и экспорт данных из одной системы в другую

# CSV

```
#ID;Исх. ИНН;ИНН;Статус ИНН;Код качества ИНН
1;ИНН: 1234567894;;NOT_SURE;NOT_VALID_TAX_CODE
2;№7726555220*;7726555220;EDITED;GOOD_CHANGED
3;1111111111;;NOT_SURE;NOT_VALID_FORMAT
4;0000000000;;NOT_SURE;NOT_VALID_FORMAT
5;***;;NOT_SURE;NOT_VALID_FORMAT
6;1111111011;1111111011;EDITED;GOOD
7;1430567890;1430567890;EDITED;GOOD
8;143056789;;NOT_SURE;NOT_VALID_LENGTH
9;516281903;;NOT_SURE;NOT_VALID_LENGTH
10;123456789110;;NOT_SURE;NOT_VALID_TAX_CODE
11;1111111111130;1111111111130;EDITED;GOOD
12;245461794475;245461794475;EDITED;GOOD
13;640318946352;;NOT_SURE;NOT_VALID_TAX_CODE
14;462900261569;;NOT_SURE;NOT_VALID_CHECK_SUM
15;/880201001;;NOT_SURE;NOT_VALID_LENGTH
```

# CSV

- формат популярен и прост, легко открывается и конвертируется в другие
- сложно производить манипуляции над данными, поэтому эти файлы анализируют и редактируют в Excel и аналогах, в БД и т.д.
- человекочитаемый (хотя иногда сложно разобратся, где какая колонка)

# Стандарт CSV

- каждая запись находится на отдельной строчке, строчки разделены переносом CRLF (`\r\n`). У последней строки перенос необязателен
- необязательная строка заголовка
- каждая строка (запись) должна содержать одинаковое количество полей



# Стандарт CSV

- поле может быть заключено в двойные кавычки
- если поле содержит запятые, переносы строк, двойные кавычки, то это поле должно быть заключено в двойные кавычки. Если этого не сделать, то данные невозможно будет корректно обработать
- символ двойной кавычки в поле должен быть удвоен

1	"R", "Перечень стран", "Режим въезда с дипломатическим паспортом", "Режим въезда со служебным паспортом", "Режим въезда с
2	1, "Абхазия", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)"
3	2, "Австралия", "Визовый", "Визовый", "Визовый"
4	3, "Австрия", "Безвизовый (до 90 дней в течение 180 дней)", "Визовый", "Визовый"
5	4, "Азербайджан", "Безвизовый", "Безвизовый", "Безвизовый"
6	5, "Албания", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Визовый"
7	6, "Алжир", "Визовый", "Визовый", "Визовый"
8	7, "Ангола", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Визовый"
9	8, "Андорра", "Визовый", "Визовый", "Визовый"
10	9, "Антигуа и Барбуда", "Безвизовый (до 30 дней)", "Безвизовый (до 30 дней)", "Безвизовый (до 30 дней)"
11	10, "Аргентина", "Безвизовый (до 3-х месяцев с правом многократного въезда и выезда)", "Безвизовый (до 3-х месяцев с пра
12	11, "Армения", "Безвизовый", "Безвизовый", "Безвизовый"
13	12, "Афганистан", "Визовый", "Визовый", "Визовый"
14	13, "Багамские Острова", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)"
15	14, "Бангладеш", "Безвизовый (до 30 дней)", "Безвизовый (до 30 дней)", "Визовый"
16	15, "Барбадос", "Безвизовый", "Безвизовый", "Безвизовый"
17	16, "Бахрейн", "Безвизовый (до 90 дней в течение 180 дней)", "Безвизовый (до 90 дней в течение 180 дней)", "Визовый"
18	17, "Белиз", "Визовый", "Визовый", "Визовый"
19	18, "Белоруссия", "Безвизовый", "Безвизовый", "Безвизовый"
20	19, "Бельгия", "Безвизовый (до 90 дней в течении 180 дней)", "Визовый", "Визовый"
21	20, "Бенин", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Визовый"
22	21, "Болгария", "Безвизовый (до 90 дней в течение 180 дней)", "Безвизовый (до 90 дней в течение 180 дней)", "Визовый"
23	22, "Боливия", "Безвизовый (до 3-х месяцев с правом многократного въезда и выезда)", "Безвизовый (до 3-х месяцев с прав
24	23, "Босния и Герцеговина", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Безвизовый (до 30 дней в течении 60 д
25	24, "Ботсвана", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Безвизовый (до 30 дней)"
26	25, "Бразилия", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)"
27	26, "Бруней-Даруссалам", "Безвизовый (до 14 дней)", "Безвизовый (до 14 дней)", "Безвизовый (до 14 дней)"
28	27, "Буркина фасо", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Визовый"
29	28, "Бурунди", "Визовый", "Визовый", "Визовый"
30	29, "Бутан", "Визовый", "Визовый", "Визовый"
31	30, "Вануату", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)", "Безвизовый (до 90 дней)"

<https://data.gov.ru/opendata/7704206201-country>

Перечень стран и режимов въезда на их территорию

# XML: eXtensible Markup Language

Где сейчас используется:

- rss/atom - новостная лента, список последних изменений и пр.
- http-запрос/ответ
- макет мобильных приложений, описание интерфейса (например, Qt)
- настройки программ
- на xml основаны такие форматы, как .docx, .xlsx и .pptx

# Пример XML

```
1 <catalog>
2   <book>
3     <name>Книга 1</name>
4     <author>Иван</author>
5     <comment>Просто книга 1</comment>
6   </book>
7   <book>
8     <name>Книга 2</name>
9     <author>Сергей</author>
10    <comment>Просто книга 2</comment>
11  </book>
12  <book>
13    <name>Книга 3</name>
14    <author>Роман</author>
15    <comment>Просто книга 3</comment>
16  </book>
17 </catalog>
```

# Плюсы XML

- формат платформонезависимый, можно легко переносить между приложениями
- поддержка юникода, можно записывать любые символы
- есть возможность валидации с помощью схем (DTD, Schema)
- строгий формат упрощает парсинг



# Минусы XML

- подробный и громоздкий синтаксис иногда усложняет читаемость
  - избыточность описания приводит к увеличению размера документа
  - нет строгой типизации
  - пространства имён сложно поддерживать при парсинге
- <f:width> и <h:width> - разные элементы

# Минусы XML

из-за строгих ограничений одна и та же структура может быть записана по-разному

```
<a><b>1</b><c>1</c></a>
```

```
<a b="1" c="1"/>
```

```
<a><b value="1"/><c value="1"/></a>
```

```
<a><fields b="1" c="1"/></a>
```

# Правила формирования XML

- каждый элемент должен быть заключен в теги. Тег - это некий текст, обернутый в угловые скобки
- теги используются парами:  
открывающий+закрывающий (бывает пустой элемент, который является открывающим и закрывающим одновременно)
- с помощью тегов мы показываем системе «Вот тут начинается элемент, а вот тут заканчивается»



# Теги

<req>



Это открывающий тег

<query>Виктор Иван</query>

<count>7</count>


</req>



Это закрывающий тег

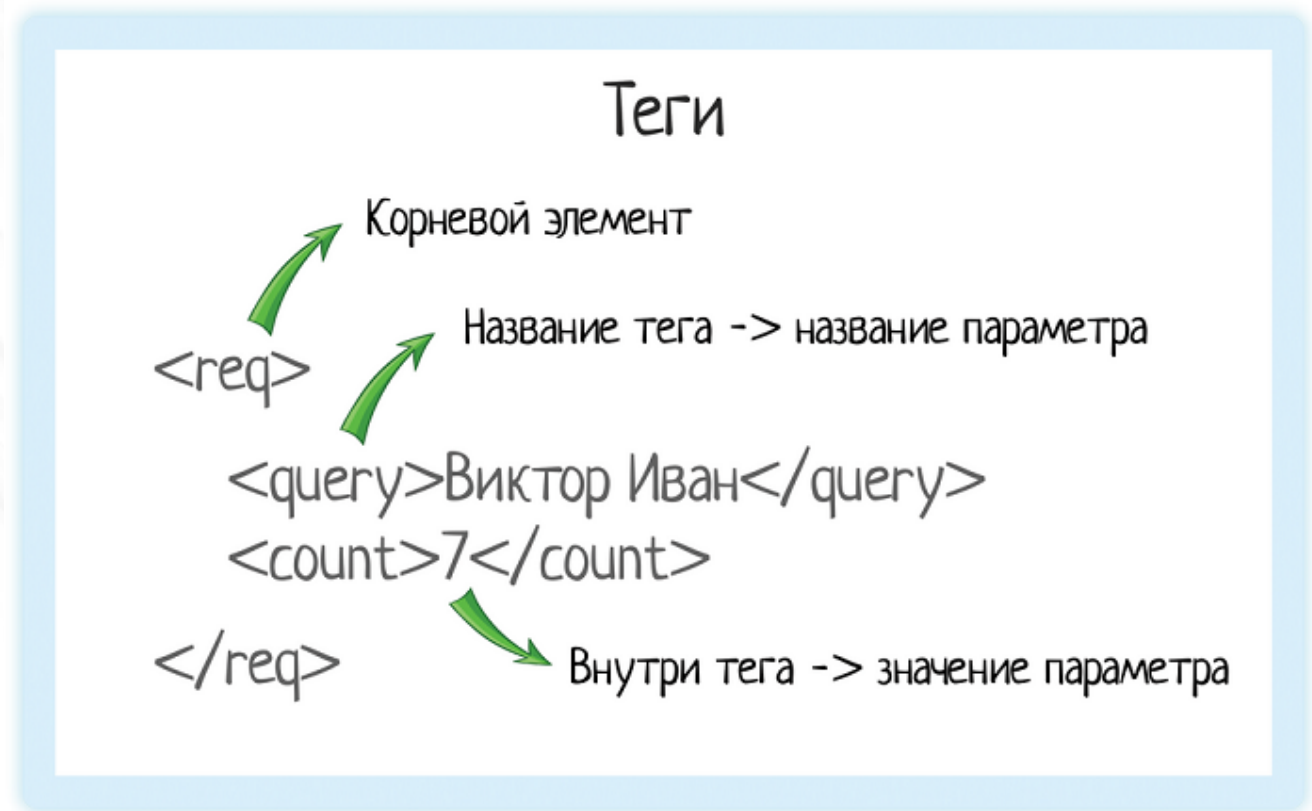
# Корневой элемент

- тег, с которого документ начинается, и которым заканчивается. Корневой тег всегда один

```
<req>  Это корневой элемент  
  <query>Виктор Иван</query>  
  <count>7</count>  
</req>
```

# Теги

- значение элемента хранится между открывающим и закрывающим тегами. Это может быть число, строка, или даже вложенные теги



# Теги

- и числа, и строки идут без кавычек. В XML нам нет нужды брать строковое значение в кавычки (а вот в JSON это сделать придется)
- у элемента могут быть атрибуты — один или несколько. Их мы указываем внутри отрывающегося тега после названия тега через пробел в виде:

название\_атрибута = "значение атрибута"

## Атрибуты

`<query>Виктор Иван</query>`

`<query attr1="value 1">Виктор Иван</query>`

`<query attr1="value 1" attr2="value 2">Виктор Иван</query>`

# Атрибуты

- помогают системе отличать один элемент от другого

```
<party type="PHYSICAL" sourceSystem="AL" rawId="2">  
  <field name="name">Олег </field>  
  <field name="birthdate">02.01.1980</field>  
  <attribute type="PHONE" rawId="AL.2.PH.1">  
    <field name="type">MOBILE</field>  
    <field name="number">+7 916 1234567</field>  
  </attribute>  
</party>
```

# Валидация XML

- какие поля будут в запросе;
- какие поля будут в ответе;
- какие типы данных у каждого поля;
- какие поля обязательны для заполнения, а какие нет;
- есть ли у поля значение по умолчанию, и какое оно;
- есть ли у поля ограничение по длине;
- есть ли у поля другие параметры;
- какая у запроса структура по вложенности элементов;



# Пример XSD

<https://mac-blog.org.ua/xsd-example/>

Портал открытых данных:

<https://data.gov.ru/opendata>

Например:

<https://data.gov.ru/opendata/7705846236-permtodistributeforeignperiodicals>



# Преимущества валидации

- запрос сначала проверяется на корректность схемы и только потом запускается
- бизнес-логика может быть сложной и валидация позволяет выдать ошибку сразу, а не через 5 минут

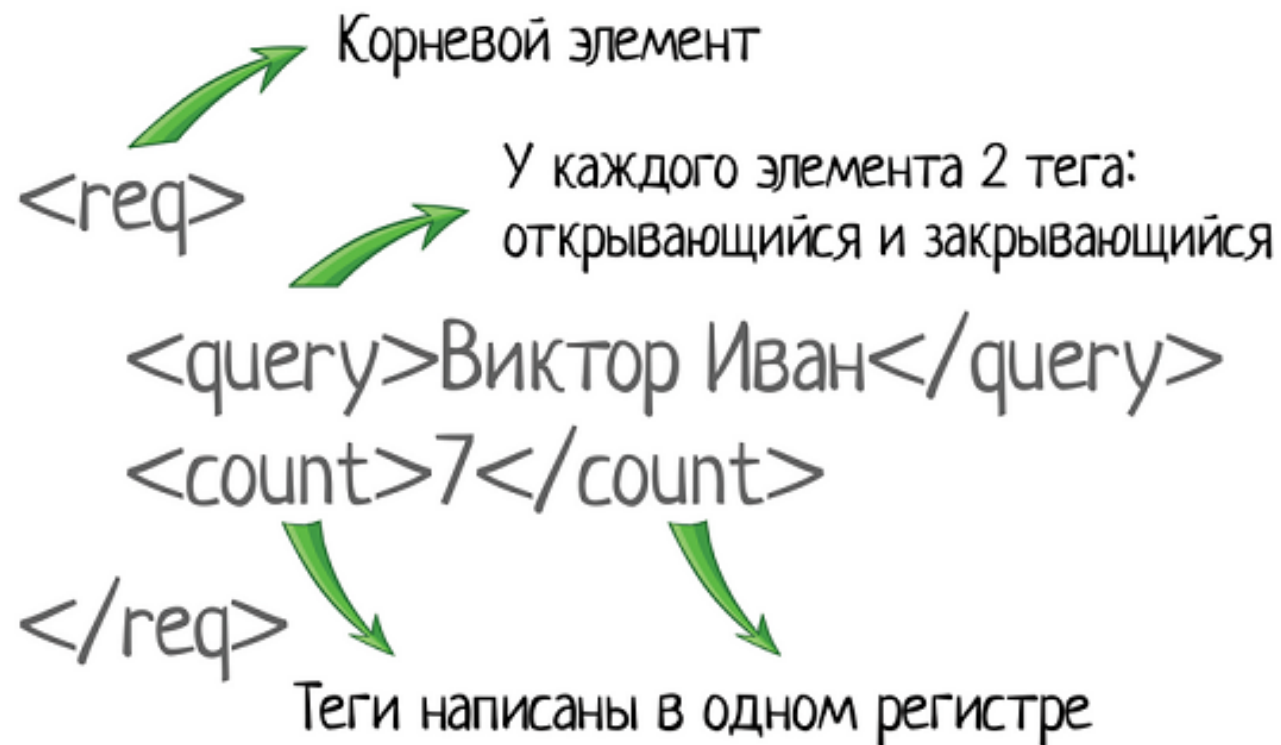
# Как всё работает?

- наш разработчик пишет XSD-схему для API запроса
- разработчик системы-заказчика, которая интегрируется с нашей, читает эту схему и строит свои запросы по ней
- система-заказчик отправляет запросы нам.
- наша система проверяет запросы по XSD — если что-то не так, сразу ошибка
- если по XSD запрос проверку прошел — включаем бизнес-логику

# Well Formed XML (синтаксис)

- есть корневой элемент
- у каждого элемента есть закрывающийся тег
- теги регистрозависимы
- соблюдается правильная вложенность элементов
- атрибуты оформлены в кавычках

# Теги



# Правильная вложенность

Правильно:

1. **<fio>Иванов Иван Иванович</fio>**

**<name>Иван</name>**

2. **<fio>Иванов <name>Иван</name>  
Иванович</fio>**

# Неправильно:

1. **<fio>Иванов <name>Иван</fio>**

**Иванович </name>**

2. **<fio>Иванов <b> <name>Иван</name>**

**Иванович</fio></b>**

# XPath

- язык запросов к элементам XML-документа
- используется для навигации по элементам XML-документов

- примеры:

`//input[@name='uid']`

`/html/body/div[2]/div[1]/div/h4[1]`

- возможно позже разберём подробнее

# Итоги по XML

- используется для хранения и передачи данных
- формат XML подчиняется стандартам
- есть строгий синтаксис, но при этом свобода действий (передавать значение в теге или атрибуте)



# JSON

- JavaScript Object Notation
- в основном используется для передачи данных между сервером и клиентом
- проще, чем XML

# JSON

- начинается и заканчивается фигурными скобками {}
- два основных элемента: ключи и значения
- ключи должны быть строками
- значения являются допустимым типом данных JSON: массив, объект, строка, логическое значение, число или значение null

**JSON Object** → {

**String Value**

"company": "mycompany",

"companycontacts": { ← **Object Inside Object**

"phone": "123-123-1234",

"email": "myemail@domain.com"

},

"employees": [ ← **JSON Array**

{

"id": 101,

"name": "John",

"contacts": [

"email1@employee1.com",

"email2@employee1.com"

]

},

{

"id": 102, ← **Number Value**

"name": "William",

"contacts": null ← **Null Value**

}

]

}

**Array Inside Array** →

# Массив

- упорядоченная коллекция значений
- заключен в квадратные скобки [], а каждое значение внутри разделено запятой

```
"students": [  
  {"firstName": "Tom", "lastName": "Jackson"},  
  {"firstName": "Linda", "lastName": "Garner"},  
  {"firstName": "Adam", "lastName": "Cooper"}  
]
```

# Объект

- объект содержит ключ и значение
- после каждого ключа стоит двоеточие, а после каждого значения – запятая
- объекты могут быть вложенными

```
"cars": {  
  "car1": "Ford",  
  "car2": "BMW",  
  "car3": "Fiat"  
}
```

# XML vs JSON

- XML – древовидная структура, JSON – ключ-значение
- парсить JSON быстрее
- JSON занимает меньше места
- XML поддерживает пространства имён
- XML не поддерживает массивы явно, для этого нужно создавать отдельный тег (вложенные можно считать массивом)

# XML vs JSON

- JSON проще понять, чем XML
- JSON менее защищен, чем XML
- JSON предпочтителен для доставки данных между серверами и браузерами
- XML предпочтителен для хранения информации на стороне сервера

# YAML

- Yet Another Markup Language
- YAML Ain't Markup Language
- часто используется для записи файлов конфигурации



```
logging:
  version: 1

formatters:
  console:
    format: "[% (asctime)s] %(name)s %(levelname)s %(message)s"
  logstash:
    message_type: asmsmk_sources_test
    (): logstash_async.formatter.LogstashFormatter

handlers:
  console:
    level: DEBUG
    class: logging.StreamHandler
    formatter: console
  logstash:
    level: DEBUG
    class: logstash_async.handler.AsynchronousLogstashHandler
    formatter: logstash
    host: 192.168.1.10
    port: 10522
    database_path: ./logstash.db
    transport: logstash_async.transport.UdpTransport

loggers:
  sources:
    handlers: [console, logstash]
    level: DEBUG
    propagate: false
```

# Синтаксис

- отступы – самое главное, они используются для разделения информации
- нужно использовать только пробелы, табы не допускаются
- YAML использует синтаксис ключ/значение
- чувствителен к регистру

# Синтаксис

- каждый отступ с двумя пробелами представляет новый уровень
- каждый новый уровень - это объект
- каждый уровень может содержать либо одну пару ключ-значение (словарь), либо список (список дефисов)
- значения для каждого ключа могут быть заключены в кавычки. Если в значении есть двоеточие или кавычка, нужно заключить его в кавычки

```
apiVersion: flagger.app/v1alpha3
kind: Canary
metadata:
  name: app
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app
  service:
    port: 8080
  canaryAnalysis:
    interval: 1m
    threshold: 5
    maxWeight: 50
    stepWeight: 5
    metrics:
      - name: request-success-rate
        threshold: 99
        interval: 1m
      - name: request-duration
        threshold: 500
        interval: 1m
  webhooks:
    - name: acceptance-test
      type: pre-rollout
      url: http://test-runner.prod
      metadata:
        type: helm
        cmd: "test app --cleanup"
```

# Полезные ссылки

<https://habr.com/ru/post/524288/>

<https://habr.com/ru/post/31225/>

[https://www.codeproject.com/Articles/  
1214409/Learn-YAML-in-five-minutes](https://www.codeproject.com/Articles/1214409/Learn-YAML-in-five-minutes)

# Резервное копирование

Три подхода к резервному копированию:

- выгрузка в sql
- копирование на уровне файлов
- непрерывное архивирование

# Выгрузка в SQL

- генерация текстового файла с командами SQL, которые при выполнении на сервере пересоздадут базу данных в том же самом состоянии, в котором она была на момент выгрузки
- создание дампа:  
`pg_dump имя_базы > файл_дампа`
- восстановление дампа:  
`psql имя_базы < файл_дампа`
- подробнее эти и другие команды будут рассмотрены на практике

```
DROP TABLE IF EXISTS `user_groups`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client  = utf8 */;
CREATE TABLE `user_groups` (
  `ug_user` int(5) unsigned NOT NULL DEFAULT 0,
  `ug_group` varbinary(255) NOT NULL DEFAULT '',
  `ug_expiry` varbinary(14) DEFAULT NULL,
  PRIMARY KEY (`ug_user`,`ug_group`),
  KEY `ug_group` (`ug_group`),
  KEY `ug_expiry` (`ug_expiry`)
) ENGINE=InnoDB DEFAULT CHARSET=binary ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8;
/*!40101 SET character_set_client  = @saved_cs_client */;
```

```
--
-- Dumping data for table `user_groups`
--
```

```
/*!40000 ALTER TABLE `user_groups` DISABLE KEYS */;
```

```
INSERT INTO `user_groups` VALUES
```

```
(2,'uploader',NULL),
```

```
(6,'autoeditor',NULL),
```

```
(10,'autoeditor',NULL),
```

```
(10,'uploader',NULL),(14,'uploader',NULL),(18,'uploader',NULL),(23,'uploader',NULL),(30,'uploader',NULL),(32,'autoe
```

```
INSERT INTO `user_groups` VALUES (987565,'uploader',NULL),(987577,'uploader',NULL),(987631,'uploader',NULL),(987700
```

```
/*!40000 ALTER TABLE `user_groups` ENABLE KEYS */;
```

```
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
```



# Преимущества

- можно загрузить в более новые версии сервера, в то время как резервная копия на уровне файловой системы и непрерывное архивирование жёстко зависят от версии
- будет работать при переносе базы данных на другую машинную архитектуру, например, при переносе с 32-битной на 64-битную версию сервера

# Если БД очень большая:

- можно использовать сжатые дампы  
`pg_dump имя_базы | gzip > имя_файла.gz`
- можно разделить большой файл на несколько  
`pg_dump имя_базы | split -b 1m -  
имя_файла`
- использовать специальный формат дампа  
`pg_dump`

# Резервное копирование на уровне файлов

- непосредственное копирование файлов, в которых PostgreSQL хранит содержимое базы данных
- чтобы полученная резервная копия была годной, сервер баз данных должен быть остановлен
- сервер нужно будет остановить и перед восстановлением данных

# Резервное копирование на уровне файлов

- невозможно восстановить только одну таблицу, потому что в результате нерабочими станут все другие таблицы в кластере баз данных
- размер копии на уровне файлов обычно больше, чем дампа SQL. (pg\_dump не нужно, например, записывать содержимое индексов, достаточно команд для их пересоздания)
- копирование на уровне файлов может выполняться быстрее

# Непрерывное архивирование и восстановление на момент времени

- всё время в процессе работы PostgreSQL ведёт журнал предзаписи
- в журнал записываются все изменения, вносимые в файлы данных
- журнал существует для безопасного восстановления после краха сервера: если происходит крах, целостность СУБД может быть восстановлена в результате «воспроизведения» записей, зафиксированных после последней контрольной точки

- можно сочетать резервное копирование на уровне файловой системы с копированием файлов WAL
- воспроизводить все записи WAL до самого конца нет необходимости
- воспроизведение можно остановить в любой точке и получить целостный снимок базы данных на этот момент времени.
- технология поддерживает восстановление на момент времени: можно восстановить состояние базы данных на любое время с момента создания резервной копии

- метод позволяет восстанавливать только весь кластер баз данных целиком, но не его части
- для архивов требуется большое хранилище
- метод предпочитается во многих ситуациях, где необходима высокая надёжность
- потенциальный объем данных, который теряется в случае, например, отключения электроэнергии, гораздо меньше при непрерывном архивировании, чем при дампе

# Лекция всё

Спасибо за внимание!

Вопросы?