нетология

Servlet Containers

Григорий
Вахмистров

Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. Предисловие
2. Java EE
3. Servlet Container
4. Layers
5. Итоги
6. Домашнее задание

Предисловие

Предисловие

На прошлой лекции мы разобрали, возможности JS в части передачи форм посредством HTTP, а также познакомились с http-сервером Grizzly.

Сегодня наша задача - поговорить о Servlet Container'ах и Java EE и продумать, как мы можем сделать нечто, похожее на REST.

Java EE

Java EE

Java EE* - набор спецификаций, описывающих промышленные сервисы для Java.

*Сейчас переходный период, поэтому Jakarta EE и EE4J - это всё о том же.

Ключевая идея

Существуют специальные контейнеры, задача которых:

1. Управлять жизненным циклом компонентов
2. Предоставлять сервисы компонентам

Ключевая идея

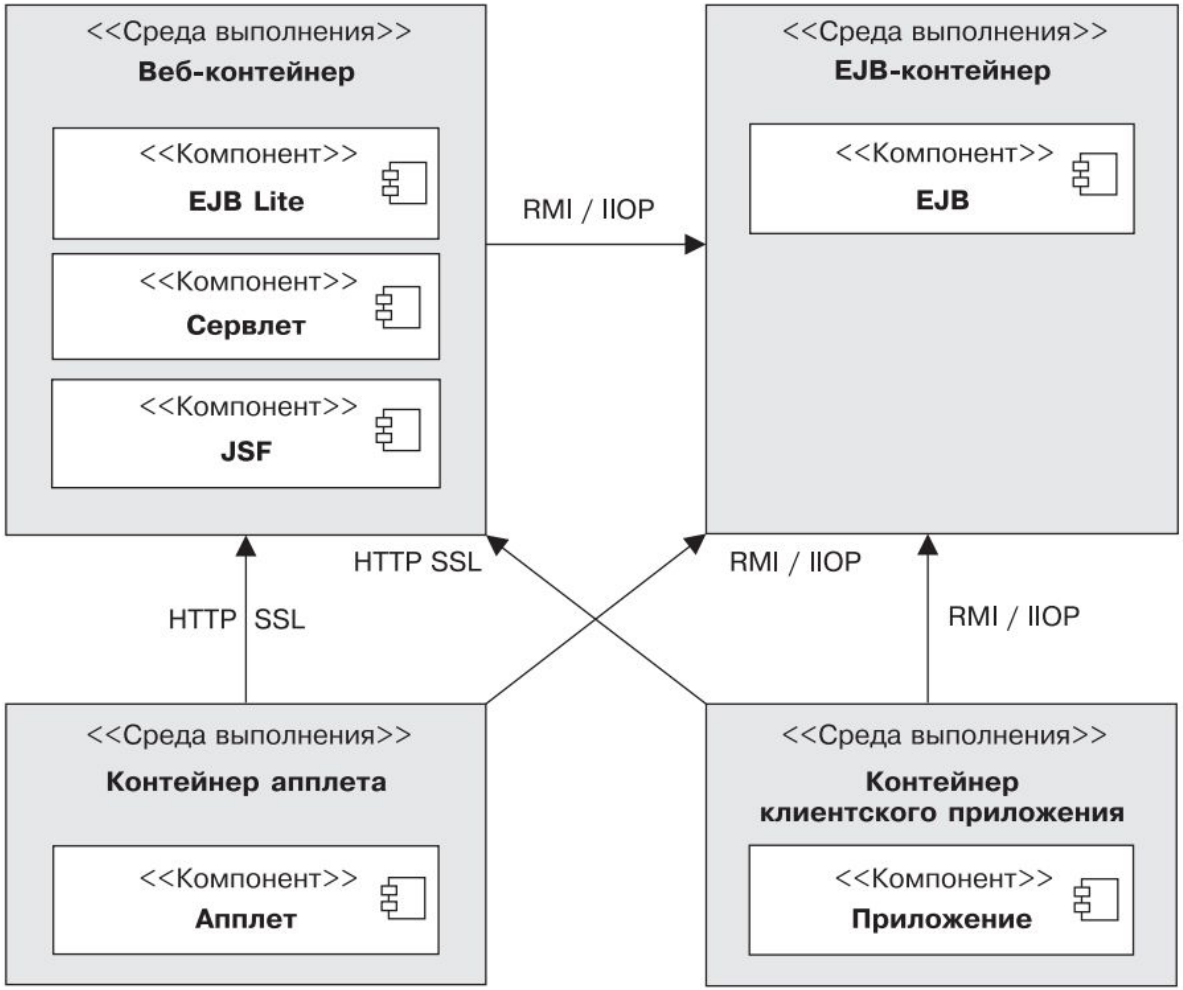


Рис. 1.1. Стандартные контейнеры Java EE

Изображение из книги Antonio Goncalves: Java EE 7

Цель

Разработчики должны писать **бизнес-логику,**

а не инфраструктурный код. За инфраструктуру отвечает контейнер.

Компоненты

Компоненты - классы, которые разработчики должны писать по определённым правилам (чаще всего через наследование или реализацию интерфейсов).

Жизненный цикл

Под жизненным циклом понимается:

1. Создание компонентов
2. Вызов специфических методов

   (при обработки запросов, завершении работы и т.д.)

Сервисы

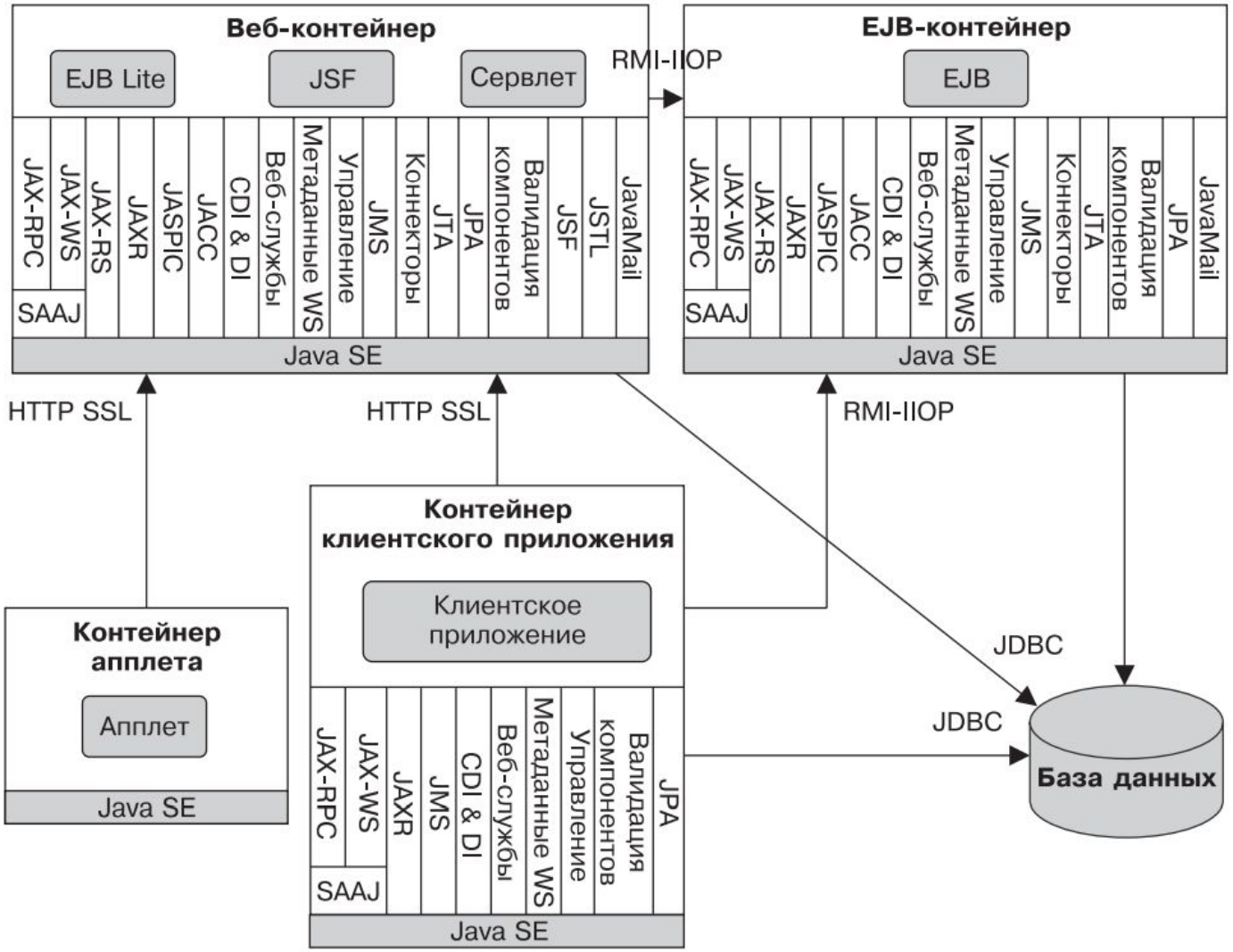


Рис. 1.2. Сервисы, предоставляемые контейнерами

Изображение из книги Antonio Goncalves: Java EE 7

Сервисы

Как видно из предыдущей диаграммы, контейнер должен предоставлять реализацию достаточно большого количества сервисов, чтобы удовлетворять требованиям.

Развёртывание

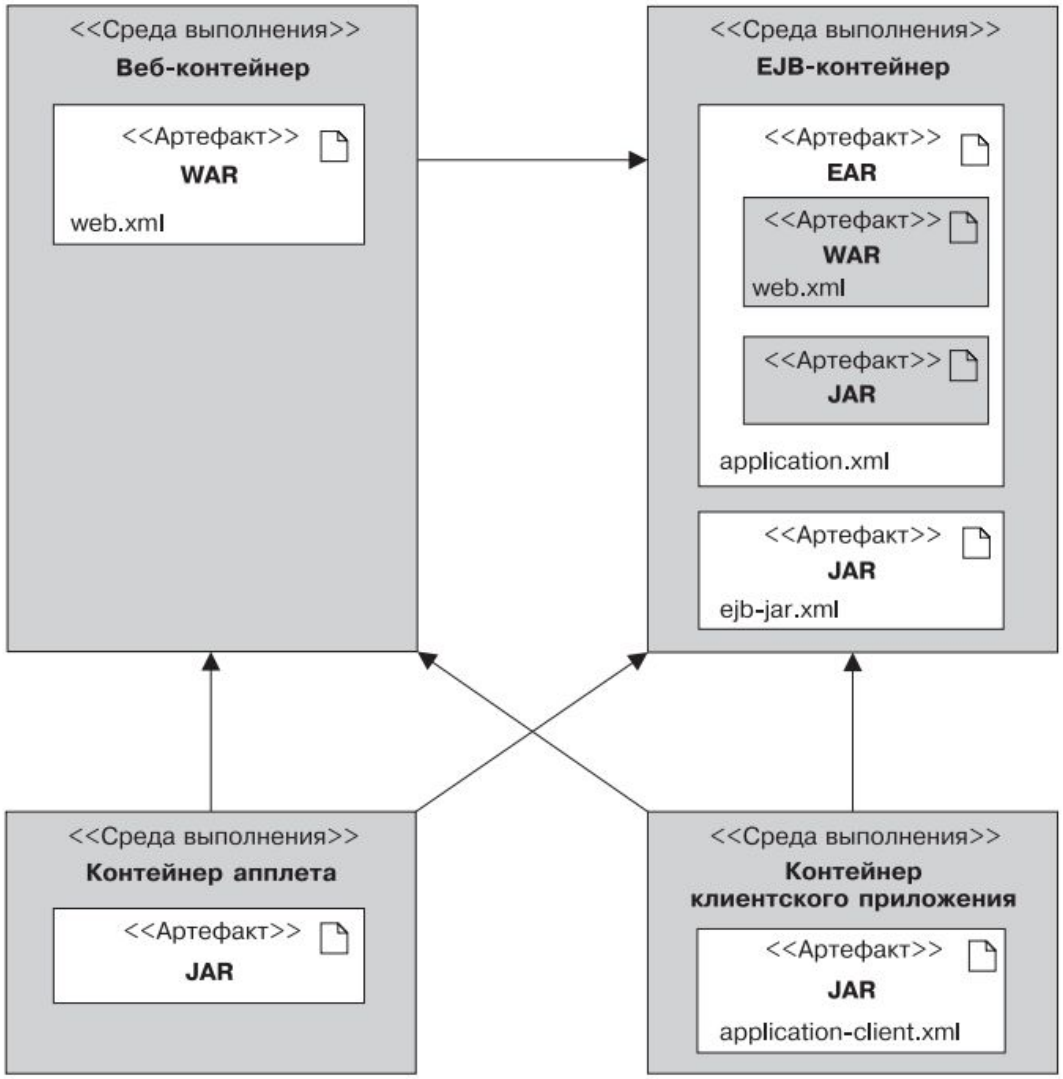


Рис. 1.3. Архивы в контейнерах

Изображение из книги Antonio Goncalves: Java EE 7

Развёртывание

Каждый тип контейнера предъявляет определённые требования к формату разворачиваемых артефактов. Артефакты должны быть упакованы в соответствующий формат и предоставлять дескрипторы развёртывания в формате xml*.

*Мы рассказываем "как было", чтобы у вас сложилось понимание. Сейчас многие процедуры упрощены и можно всё делать программно, а не через XML.

Существующие решения

- GlassFish
- WildFly
- WebSphere
- Apache TomEE
- и т.д.

Q & A

Q: зачем мы изучаем Java EE, ведь курс по Spring?

A: Spring активно использует* Java EE, поэтому чтобы понимать, как работает Spring, нужно понимать технологии, лежащие в его основе.

---

*Речь идёт о классическом стеке.

Servlet Container

Servlet Container

Реализация полноценного контейнера зачастую не нужна (т.к. не требуются все описанные сервисы). Поэтому появились контейнеры, которые реализуют лишь часть наиболее популярных спецификаций, например, Servlet Container'ы.

Servlet

Servlet - web-компонент, управляемый контейнером, предназначенный для генерации динамического контента.

Вопрос к аудитории:

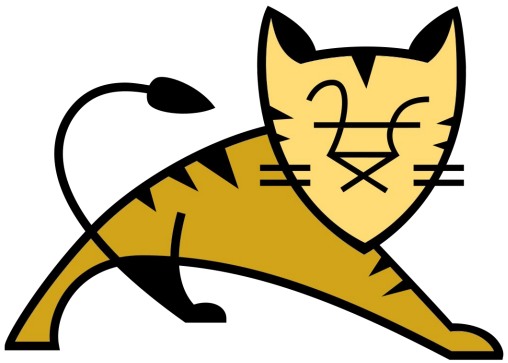
что такое динамический контент?

Спецификации

- v4.0: JSR-369 (июль 2017)
- v3.1: JSR-340 (апрель 2013)

Более ранние нас не интересуют.

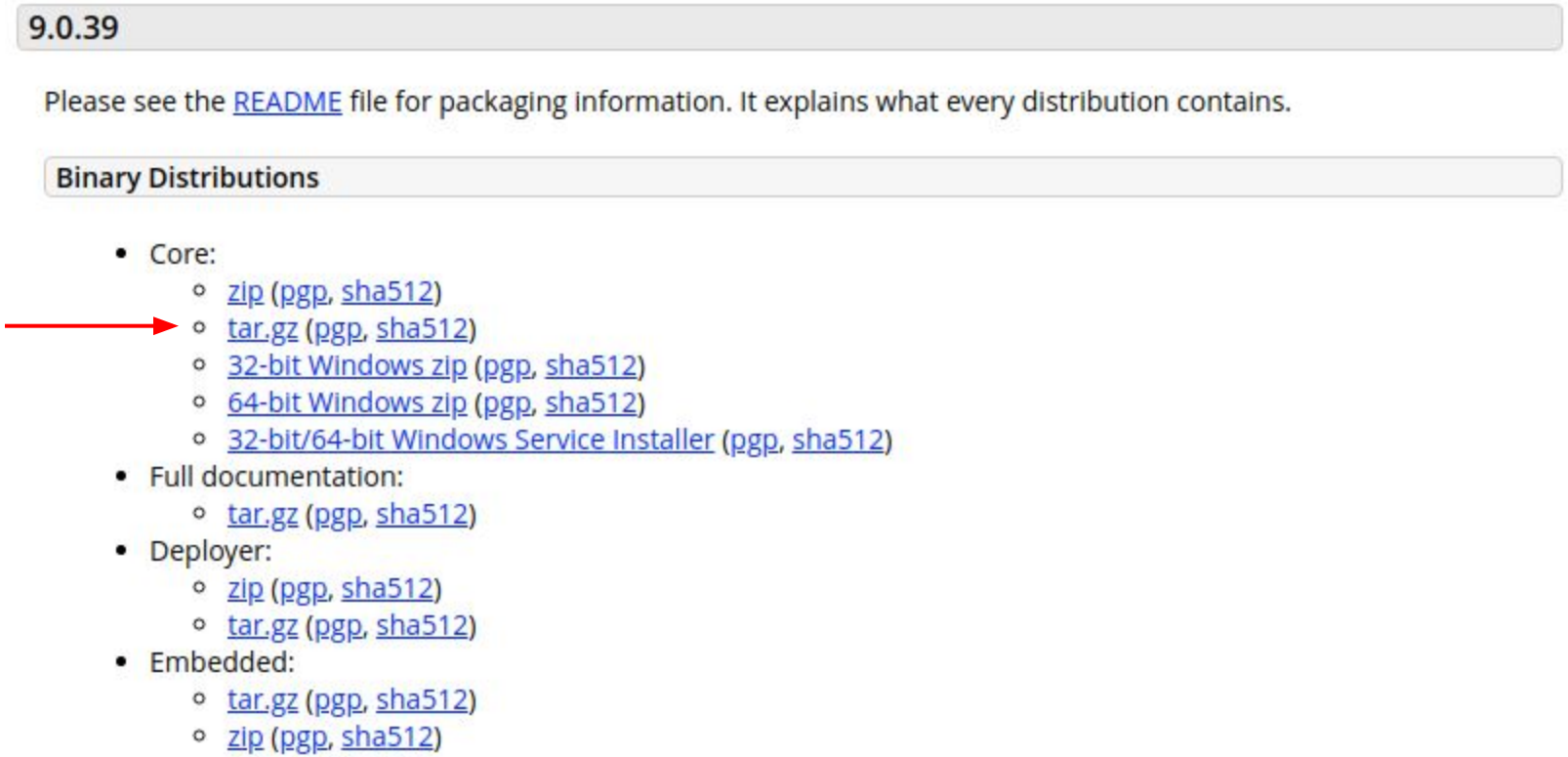
Apache Tomcat



Apache Tomcat - самая часто используемая реализация контейнера сервлетов (хотя есть и более "продвинутые" вроде Jetty).

Установка

Выбираем на сайте последнюю стабильную версию, скачиваем и распаковываем:



9.0.39

Please see the README file for packaging information. It explains what every distribution contains.

Binary Distributions

- Core:
  - zip (pgp, sha512)
  - tar.gz (pgp, sha512)
  - 32-bit Windows zip (pgp, sha512)
  - 64-bit Windows zip (pgp, sha512)
  - 32-bit/64-bit Windows Service Installer (pgp, sha512)
- Full documentation:
  - tar.gz (pgp, sha512)
- Deployer:
  - zip (pgp, sha512)
  - tar.gz (pgp, sha512)
- Embedded:
  - tar.gz (pgp, sha512)
  - zip (pgp, sha512)

IntelliJ IDEA

Для создания и запуска проекта с Java EE вам понадобится **IntelliJ IDEA Ultimate**. JetBrains предоставляет пробную версию на 30 дней.

Maven

```
<groupId>ru.netology</groupId>
<artifactId>servlets</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

war - формат упаковки

provided - предоставляется в рантайме

Servlet

```
public class MainServlet extends HttpServlet {
}
```

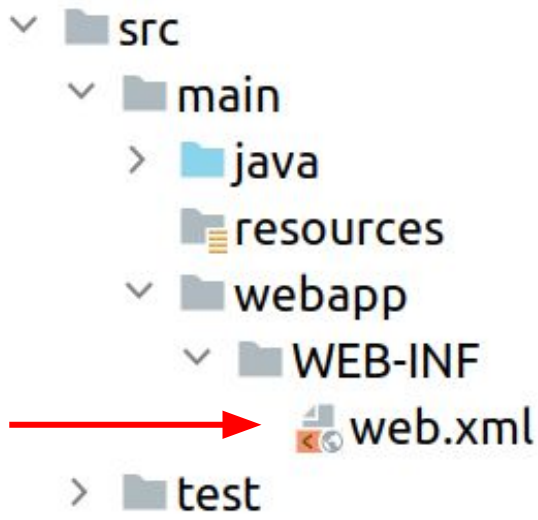
Ключевые методы сервиса:

- init (инициализация)
- destroy (уничтожение)
- service (обработка запроса)

В HttpServlet'е они уже реализованы, поэтому мы пока просто отнаследуемся.

Web.xml

Дескриптор развёртывания определяет регистрацию сервлетов и их привязку к путям обработки запросов:



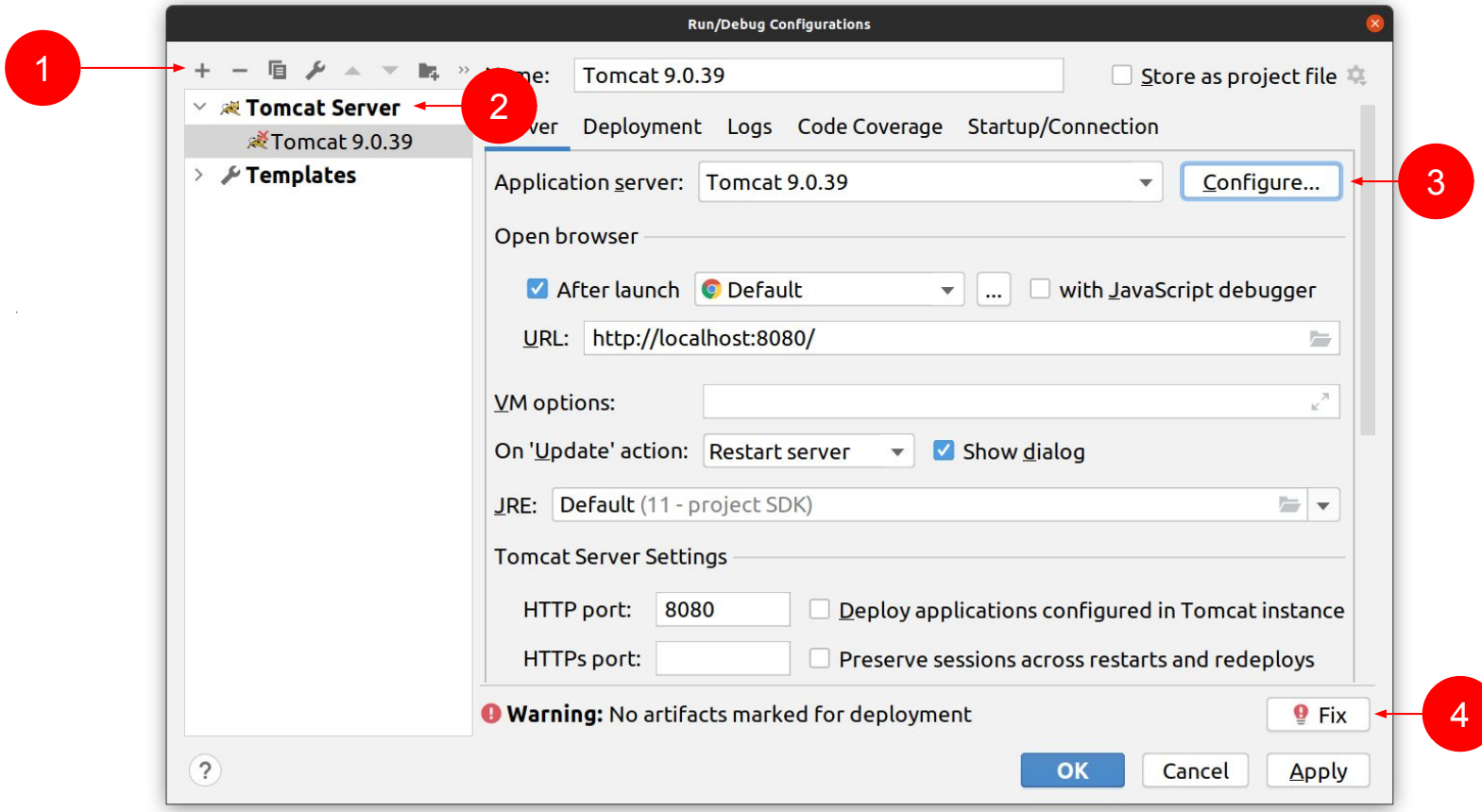
Web.xml

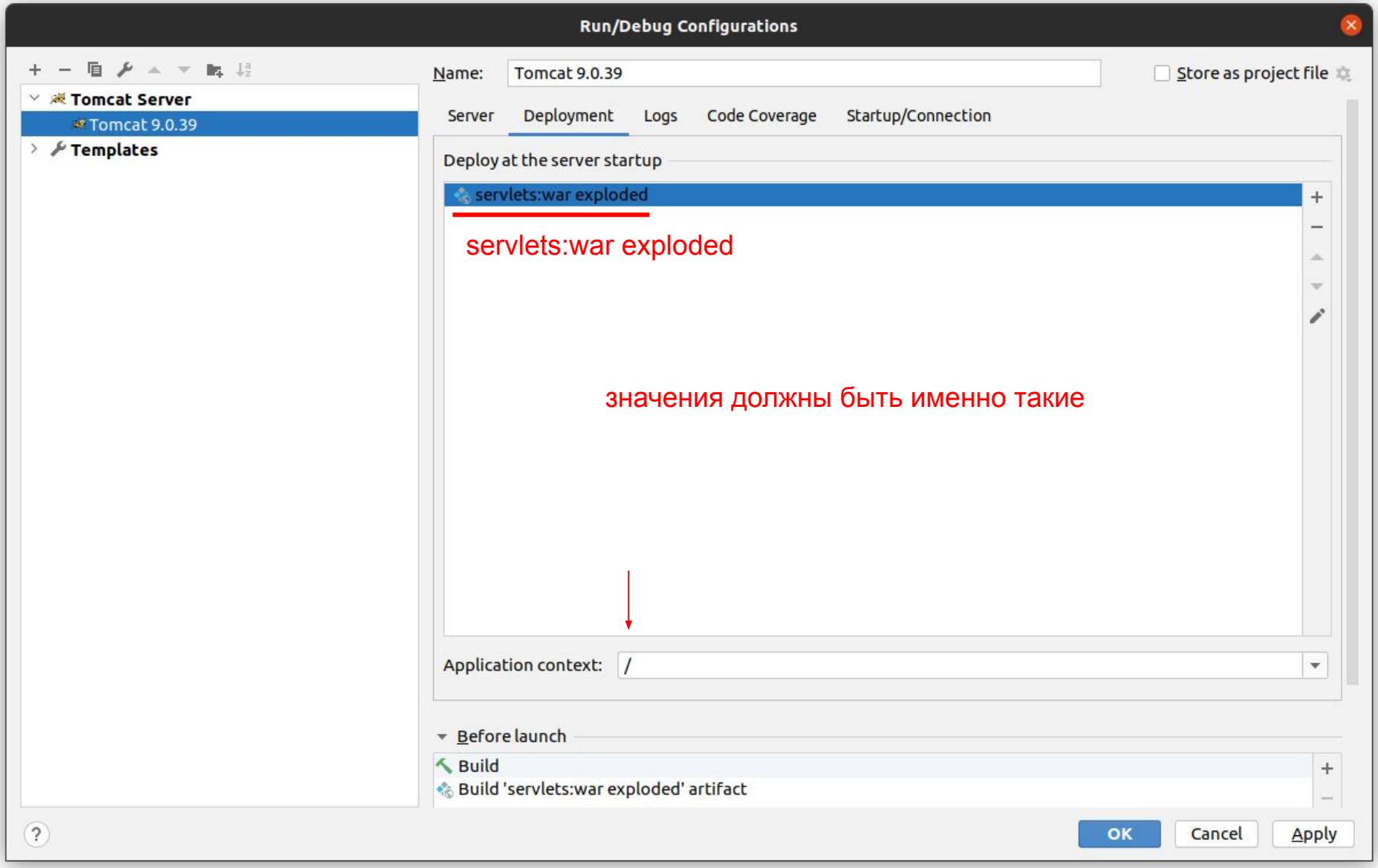
```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
         version="4.0">
    <!-- определение servlet'а -->
    <servlet>
        <servlet-name>main</servlet-name>
        <servlet-class>ru.netology.servlet.MainServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- mapping - привязка к определённому URL-->
    <servlet-mapping>
        <servlet-name>main</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Запуск

Можно либо упаковать в war (mvn clean package), либо настроить запуск средствами IDEA (Add Configurations):




Run/Debug Configurations
Tomcat Server
Tomcat 9.0.39
Templates
Name:
Tomcat 9.0.39
Store as project file
Server
Deployment
Logs
Code Coverage
Startup/Connection
Deploy at the server startup
servlets:war exploded
servlets:war exploded
значения должны быть именно такие
Application context:
/
Before launch
Build
Build 'servlets:war exploded' artifact
OK
Cancel
Apply

Запуск

При попытке запуска будем получать:

Почему? Это дефолтная реализация HttpServlet.

HttpServlet

```
public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException {
  if (req instanceof HttpServletRequest && res instanceof HttpServletResponse) {
    HttpServletRequest request = (HttpServletRequest)req;
    HttpServletResponse response = (HttpServletResponse)res;
    this.service(request, response);
  } else {
    throw new ServletException("non-HTTP request or response");
  }
}
```

```
protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {...}
```

именно здесь решается, какой конкретно метод будет всё обрабатывать

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
  String protocol = req.getProtocol();
  String msg = lStrings.getString( key: "http.method_get_not_supported");
  if (protocol.endsWith("1.1")) {
    resp.sendError( i: 405, msg);
  } else {
    resp.sendError( i: 400, msg);
  }

}
```

MainServlet

```
public class MainServlet extends HttpServlet {
  @Override
  protected void service(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    resp.setContentType("text/plain");
    resp.getWriter().print("ok");
  }
}
```

Итоги

Мы посмотрели, что можем создавать сервлеты, регистрировать их на определённые пути (или даже на шаблоны путей: сейчас в MainServlet обрабатывает все пути).

Архитектура

Вопрос к аудитории:

как организовать функциональность по просмотру списка элементов, добавлению, удалению и т.д.?

Сейчас мы рассматриваем концептуально - как организовать структуру кода.

Варианты

1. Один сервлет на всё (там же пишем логику)
2. На каждый путь - свой сервлет (там же пишем логику)
3. Альтернативный вариант

Layers

Layers

Здесь нам стоит вспомнить о принципе **Single Responsibility**. Если сервлет будет отвечать и за приём запросов, и за бизнес-логику, и за хранение данных, и за отправку ответов, то это будет явно не Single Responsibility.

Значит, нам нужно придумать что-то, что позволит **разделить всё на зоны ответственности**, а ещё лучше - слои, которые будут отвечать строго за отдельные задачи.

Layers

Один из подходов выглядит вот так:

- Controller - приём запросов и подготовка ответов
- Service - бизнес-логика
- Repository - хранение данных

Social Service

Попробуем смоделировать это на примере небольшого Social-сервиса: есть посты, которые можно просматривать, добавлять, удалять, изменять.

Repository

```
// Stub
public class PostRepository {
  public List<Post> all() {
    return Collections.emptyList();
  }

  public Optional<Post> getById(long id) {
    return Optional.empty();
  }

  public Post save(Post post) {
    return post;
  }

  public void removeById(long id) {
  }
}
```

Напоминаем, что по нашей договорённости, репозиторий не занимается бизнес-логикой, он отвечает только за хранение данных.

Service

```
public class PostService {
  private final PostRepository repository;

  public PostService(PostRepository repository) {
    this.repository = repository;
  }

  public List<Post> all() {
    return repository.all();
  }

  public Post getById(long id) {
    return repository.getById(id).orElseThrow(NotFoundException::new);
  }

  public Post save(Post post) {
    return repository.save(post);
  }

  public void removeById(long id) {
    repository.removeById(id);
  }
}
```

Сервис занимается исключительно бизнес-логикой, и не отвечает за хранение данных. Поэтому ему необходим репозиторий, которому он делегирует задачи.

Controller

```
public class PostController {
  public static final String APPLICATION_JSON = "application/json";
  private final PostService service;

  public PostController(PostService service) {
    this.service = service;
  }

  public void all(HttpServletResponse response) throws IOException {
    response.setContentType(APPLICATION_JSON);
    final var data :List<Post> = service.all();
    final var gson = new Gson();
    response.getWriter().print(gson.toJson(data));
  }

  public void getById(long id, HttpServletResponse response) {
    // TODO: deserialize request & serialize response
  }

  public void save(Reader body, HttpServletResponse response) throws IOException {
    response.setContentType(APPLICATION_JSON);
    final var gson = new Gson();
    final var post :Post = gson.fromJson(body, Post.class);
    final var data :Post = service.save(post);
    response.getWriter().print(gson.toJson(data));
  }

  public void removeById(long id, HttpServletResponse response) {
    // TODO: deserialize request & serialize response
  }
}
```

Задача контроллера - сериализация и десериализация данных, вызов нужного сервиса.

MainServlet

А чем же тогда будет заниматься **MainServlet**? Он будет заниматься двумя вещами:

1. Инициализировать все необходимые объекты
2. Диспетчеризовать запросы

Конечно, принцип Single Responsibility немного пострадает, но мы намеренно не усложняем пример.

```
public class MainServlet extends HttpServlet {
  private PostController controller;

  @Override
  public void init() throws ServletException {
    final var repository = new PostRepository();
    final var service = new PostService(repository);
    controller = new PostController(service);
  }

  @Override
  protected void service(HttpServletRequest req, HttpServletResponse resp) {
    // если деплоились в root context, то достаточно этого
    try {
      final var path :String = req.getRequestURI();
      final var method :String = req.getMethod();
      // primitive routing
      if (method.equals("GET") && path.equals("/api/posts")) {...}
      if (method.equals("GET") && path.matches( regex: "/api/posts/\\d+")) {...}
      if (method.equals("POST") && path.equals("/api/posts")) {...}
      if (method.equals("DELETE") && path.matches( regex: "/api/posts/\\d+")) {...}
      resp.setStatus(HttpServletResponse.SC_NOT_FOUND);
    } catch (Exception e) {
      e.printStackTrace();
      resp.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    }
  }
}
```

метод service вызывается на каждый запрос, а init - при инициализации

Обсуждение

Как вы думаете, что плохо в написанном нами коде?*

*Конечно же, мы максимально упростили код. Например, нет смысла десериализовать из JSON, если Content-Type != application/json. И некоторые ошибки будут не ошибками сервера, а клиента - 4xx. А также общий стиль, вроде выноса GET, POST в public static final.

Масштабируемость

Ключевое: он не масштабируется.

Когда наш социальный сервис станет большим (появятся группы, личные сообщения, товары, фото, видео, музыка и т.д.) один только метод init, который "связывает" зависимости друг с другом, станет просто гигантским.

А наш **ad hoc** роутинг (диспетчеризация запросов) вырастет ещё больше.

Масштабируемость

Поэтому нам нужны два инструмента:

1. Автоматическое создание и связывание зависимостей
2. Удобные регистрация и управление обработчиками запросов (включая вспомогательные вещи вроде автоматической сериализации/десериализации для JSON)

Итоги

Итоги

Сегодня мы познакомились с Servlet'ами. Ключевое: их можно использовать для создания REST приложений, но очень уж много кода приходится писать самим.

Совершенно очевидно, что как и в случае с веб-серверами, наверняка этот код уже кто-то написал и нам остаётся лишь научиться им пользоваться.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера .
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Григорий Вахмистров