

# Spring MVC (часть 1)



ПРЕПОДАВАТЕЛЬ



# Юрий Костяной

**Java/Kotlin backend-разработчик**

- 3+ года опыта в коммерческой разработке
- 2+ года опыта в преподавании
- Проекты по интеграции сторонних платформ, CRM
- Проблемно-ориентированный подход в преподавании



# ВАЖНО:

- Камера должна быть включена на протяжении всего занятия.
- Если у Вас возник вопрос в процессе занятия, пожалуйста, поднимите руку и дождитесь, пока преподаватель закончит мысль и спросит Вас, также можно задать вопрос в чате или когда преподаватель скажет, что начался блок вопросов.
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях.
- Вести себя уважительно и этично по отношению к остальным участникам занятия.
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя.

# ПЛАН ЗАНЯТИЯ

1. Повторение
2. Основной блок
3. Вопросы по основному блоку
4. Домашняя работа



TEL-RAN  
by Starta Institute

1

# ПОВТОРЕНИЕ ИЗУЧЕННОГО

# Повторение

Что такое REST? Каковы его преимущества?

# Повторение

Что такое REST? Каковы его преимущества?

REST – архитектурный стиль организации API клиент-серверного приложения. Основные преимущества:

использует стандартный протокол http для транспорта данных;

шифрование «из коробки» благодаря https;

не сохраняет состояние сервиса между запросами (stateless);

CRUD-совместим;

позволяет использовать разные типы данных для передачи полезной нагрузки (JSON, XML и др.)

# Повторение

Из чего состоит сообщение http?



# Повторение

Из чего состоит сообщение http?

- Стартовая строка (версия протокола, URI, метод запроса/код ответа с пояснением);
- Заголовки сообщения (пары ключ-значение);
- Тело сообщения (опционально)

# Повторение

Возможно ли передать текст, картинку и архив в одном сообщении http?

# Повторение

Возможно ли передать текст, картинку и архив в одном сообщении http?

Да, для этого в заголовке *Content-Type* нужно указать *multipart/form-data; текст\_разделителя*. Тело такого запроса состоит из секций, отделённых друг от друга текстом разделителем. Каждая секция имеет свои заголовки, указывающие в том числе на длину секции в байтах.

# Повторение

Для чего нужны коды состояния в ответе сервера?

Расшифруйте значения кодов состояния:

200

400

401

403

404

500

# Повторение

Для чего нужны коды состояния в ответе сервера?

Для отображения результата обработки запроса клиента сервером.

Расшифруйте значения кодов состояния:

200 OK

400 Bad Request

401 Unauthorized

403 Forbidden

404 Not Found

500 Internal Server Error

# Повторение

Разберите URL по составу:

<https://habr.com/ru/companies/otus/articles/665952>

<https://habr.com/ru/companies/otus/articles/665555>

[https://en.wikipedia.org/wiki/Footloose\\_\(1984\\_film\)#Filming](https://en.wikipedia.org/wiki/Footloose_(1984_film)#Filming)

[https://learngitbranching.js.org/?locale=ru\\_RU](https://learngitbranching.js.org/?locale=ru_RU)

# Повторение

Разберите URL по составу:

<https://habr.com/ru/companies/otus/articles/665952>

<https://habr.com/ru/companies/otus/articles/665555>

Протокол https, хост – habr.com, путь - /ru/companies/otus/articles/{id}, где id – переменная пути

[https://en.wikipedia.org/wiki/Footloose\\_\(1984\\_film\)#Filming](https://en.wikipedia.org/wiki/Footloose_(1984_film)#Filming)

Протокол https, хост – en.wikipedia.org, путь - /wiki/Footloose\_(1984\_film), раздел просматриваемой страницы – Filming

[https://learngitbranching.js.org/?locale=ru\\_RU](https://learngitbranching.js.org/?locale=ru_RU)

Протокол https, хост – learngitbranching.js.org, путь - /, параметр locale имеет значение ru\_RU

# Повторение

В чём прикол мема?





2

# ОСНОВНОЙ БЛОК

# Введение

- Вебненький
- Полный Rest
- Сэрвис



# Проблема

Сервлеты – классы, расширяющие возможности веб-серверов (например, *Apache Tomcat*).

Изначально требовалось написать много кода, что настроить сервлеты и веб-приложение в целом. Со временем появились хорошие практики такие, как применение паттерна MVC, а также автоматизации для настроек сервлетов в виде фреймворков.

*А что приготовил Spring для упрощения этой задачи?*



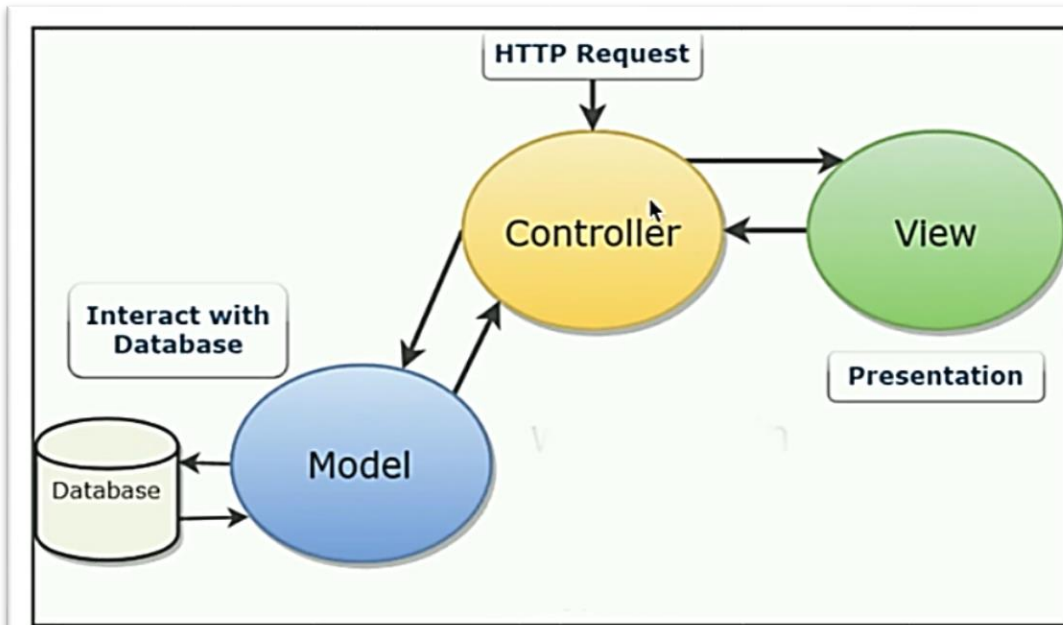
# Вебненький Spring MVC

**Spring MVC** – это модуль, входящий в состав Spring, который позволяет создавать веб-приложения с помощью паттерна *MVC (Model-View-Controller)*. Запрос пользователя в первую очередь попадает на **Контроллер** (контроль навигации, обработка запросов).

Контроллер производит запросы в

**Модель**, которая выполняет бизнес-логику и взаимодействует с базой данных.

По данным из модели **Представление** формирует ответ для пользователя (например, html-страницу для отображения).



# Вебненький Spring MVC

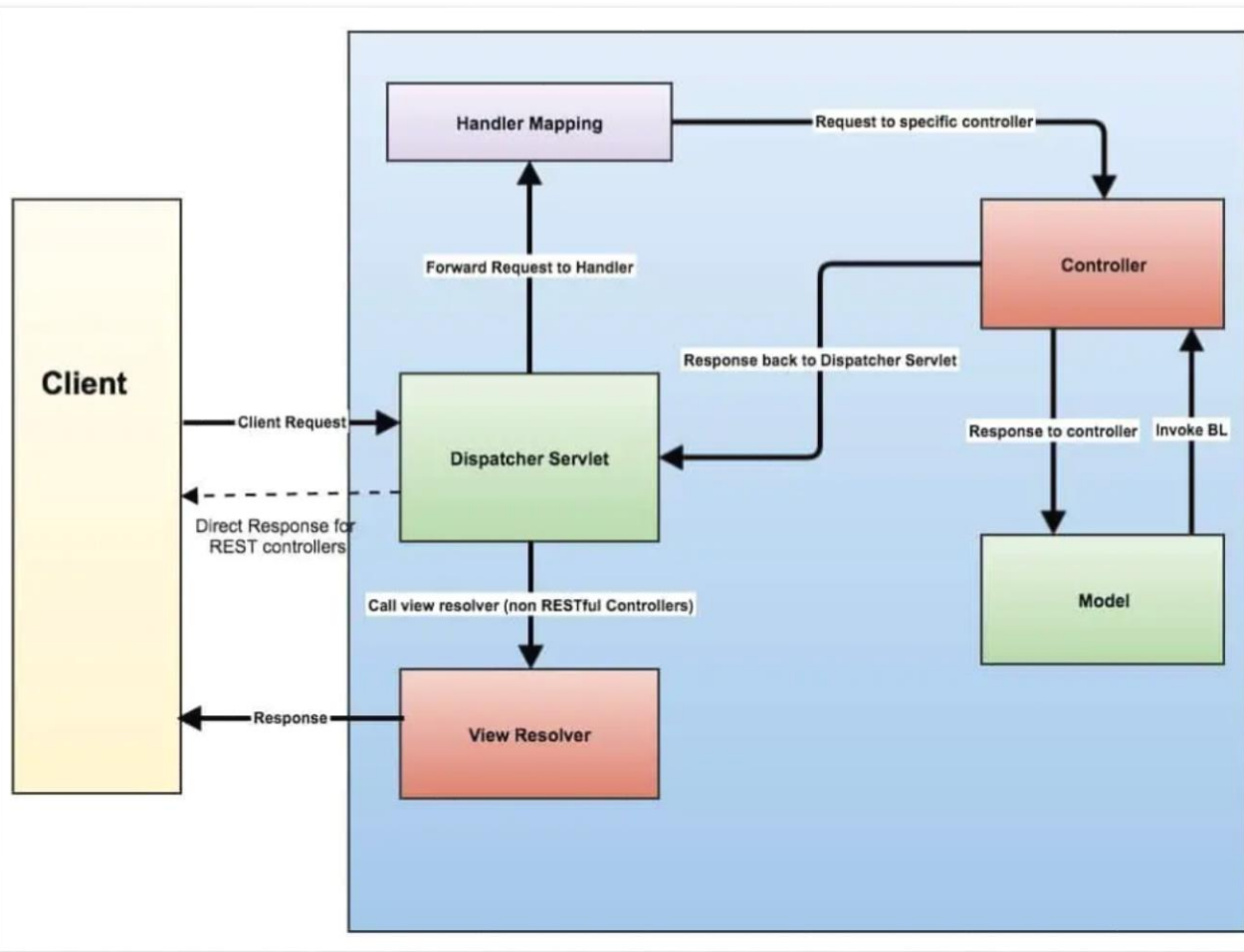
В составе Spring MVC-приложения можно выделить:

- Java-классы для настройки Spring (конфигурации, компоненты);
- Java-классы, выполняющие роль контроллеров, т.е. приём запроса на нужный URI и первичную обработку (помечаются аннотацией **@Controller**);
- **Web-сервер** – это основная программа, обеспечивающая обмен данными по *REST*. В неё встраиваются наши MVC-приложения. Примером служит [Oracle GlassFish server](#).
- **DispatcherServlet** – класс, обеспечивающий основную автоматизацию в Spring MVC. Spring MVC при запуске анализирует имеющиеся классы с *@Controller* и настраивает под них сервлеты, которыми управляет *DispatcherServlet*. При поступлении запроса сервер принимает запрос первым и передаёт его в настроенный *DispatcherServlet*. *DispatcherServlet* перенаправляет его в MVC-приложение на нужный контроллер.
- *html*-страницы, *css*-файлы для оформления страниц и *JavaScript*-код для создания логики работы страниц.

# Вебненький Spring MVC

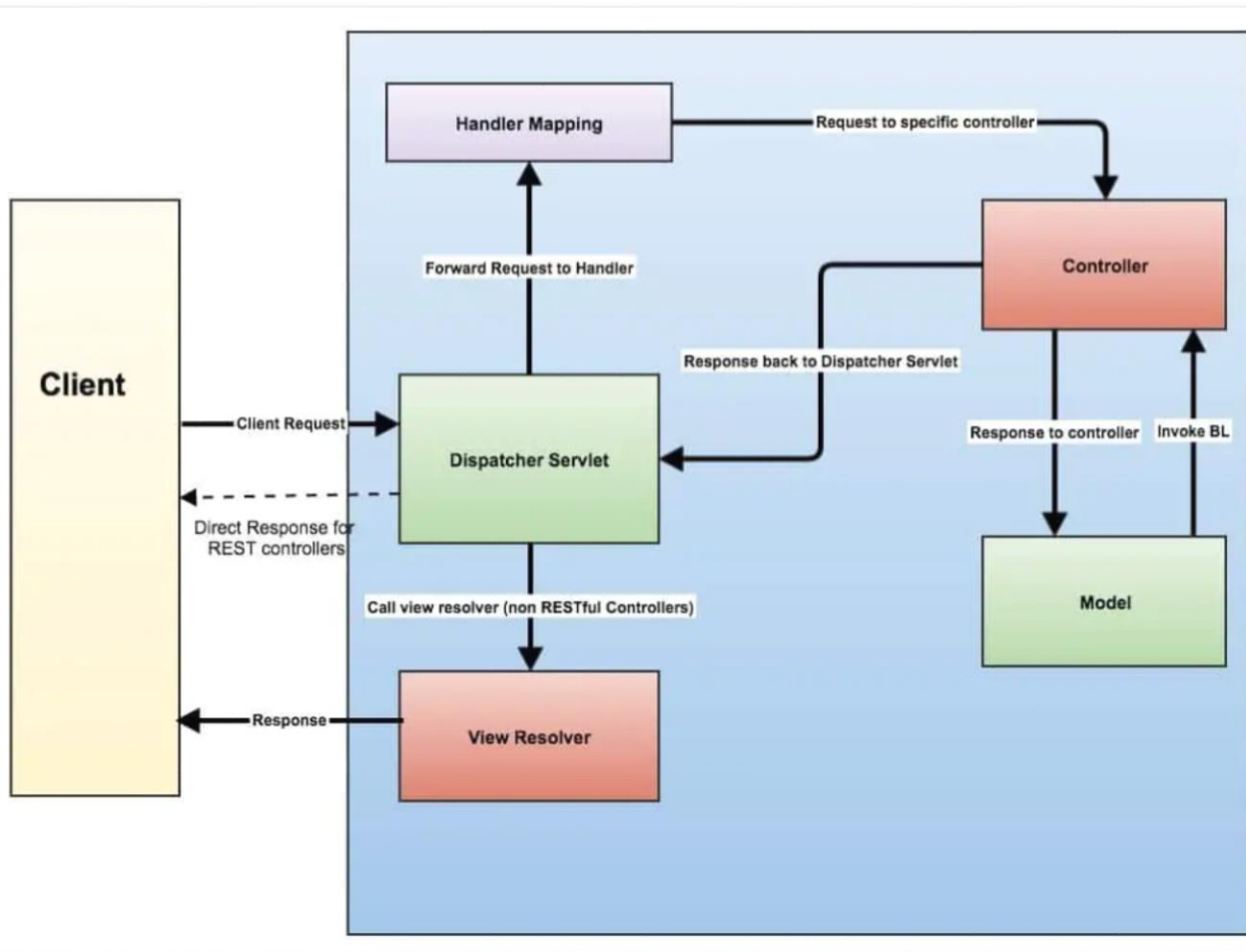
*DispatcherServlet*

перехватывает все входящие запросы, преобразует полезную нагрузку тела запроса в структуру данных *Spring MVC*. Затем отправляет запрос на конкретный контроллер, отвечающий за *URL*, по которому пришёл запрос.



# Вебненький Spring MVC

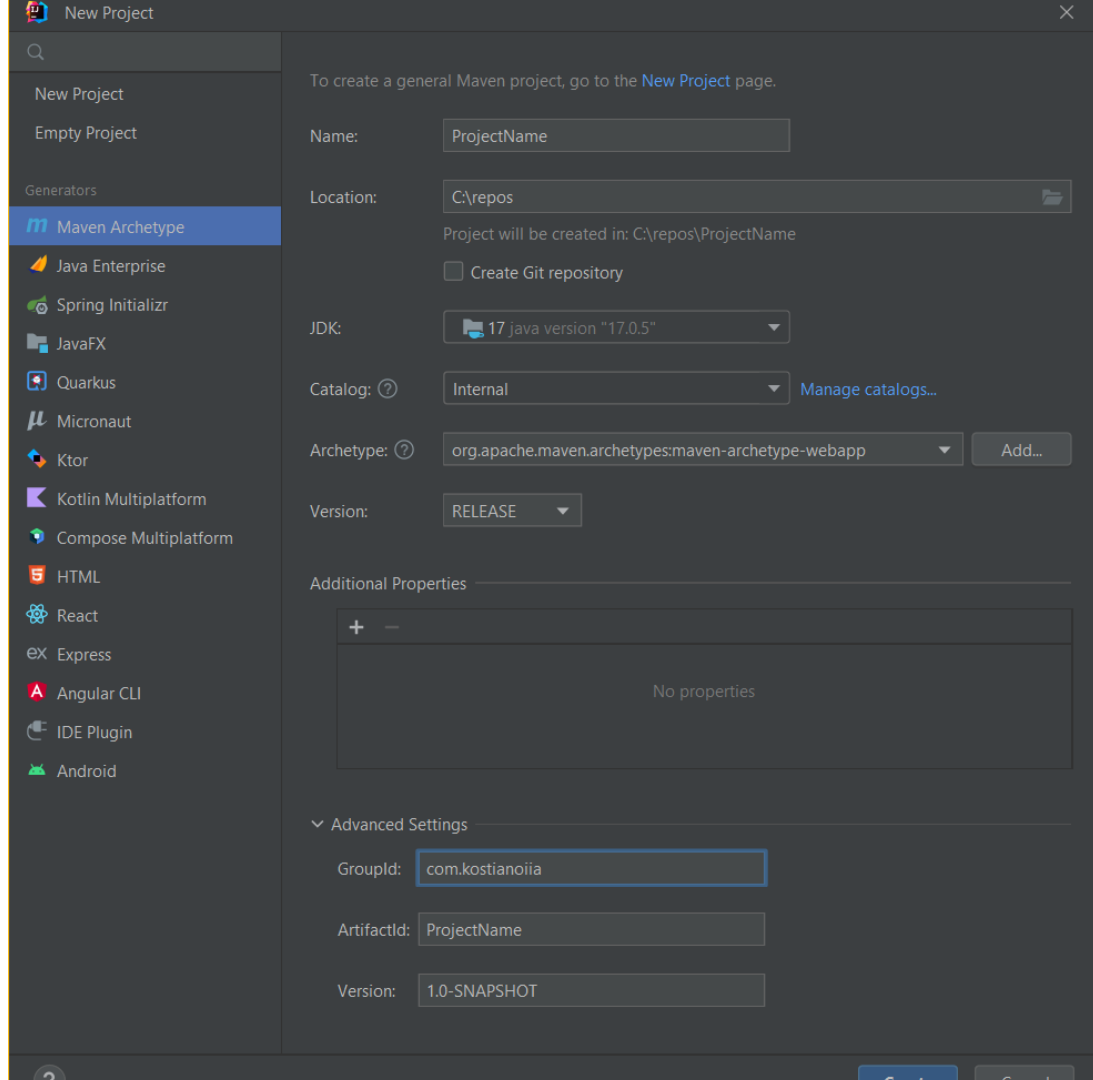
Контроллер должен обработать данные запроса и инициировать выполнение бизнес-логики. В зависимости от типа *API* контроллер либо отправляет результат работы модели клиенту в сыром виде (*JSON*, *XML*), либо через модуль представления (view resolver).



# Задание

1 Создайте *Spring*-приложение с *xml*-конфигурацией. Шаги создания:

- в *IntelliJ IDEA Ultimate* выберите «*New Project*».
- в шаблонах выберите Maven Archetype.
- в поле *Archetype* выберите *org.apache.maven.archetypes:maven-archetype-webapp*
- укажите версию JDK
- укажите имя проекта, папку размещения и данные артефакта





# Задание

2 Скачайте и установите web-сервер Apache Tomcat

<https://tomcat.apache.org/download-10.cgi>

Распакуйте архив в корень Вашего диска (например, диска C), т.к. есть проблемы с пробелами в пути к папке Tomcat.

Изучите RUNNING.txt в распакованной папке. В частности,

- убедитесь, что в операционной системе установлено значение переменной JAVA\_HOME;
- добавить переменную окружения CATALINA\_HOME с указанием ранее распакованной папки Tomcat. Например, C:\apache-tomcat-10.1.18.
- перейдите в распакованную ранее папку Tomcat, пройдите в папку bin и выполните под администратором catalina.bin (для Windows). В Mac OS/Linux нужно открыть консоль, перейти в консоли в папку *bin* ранее распакованной папки Tomcat и выполнить

*chmod +x catalina.sh*

# Задание

3 Настройте запуск web-сервера в IntelliJ IDEA

Ultimate: верхнее меню -> Run

-> Edit configuration -> + в

левом верхнем углу ->

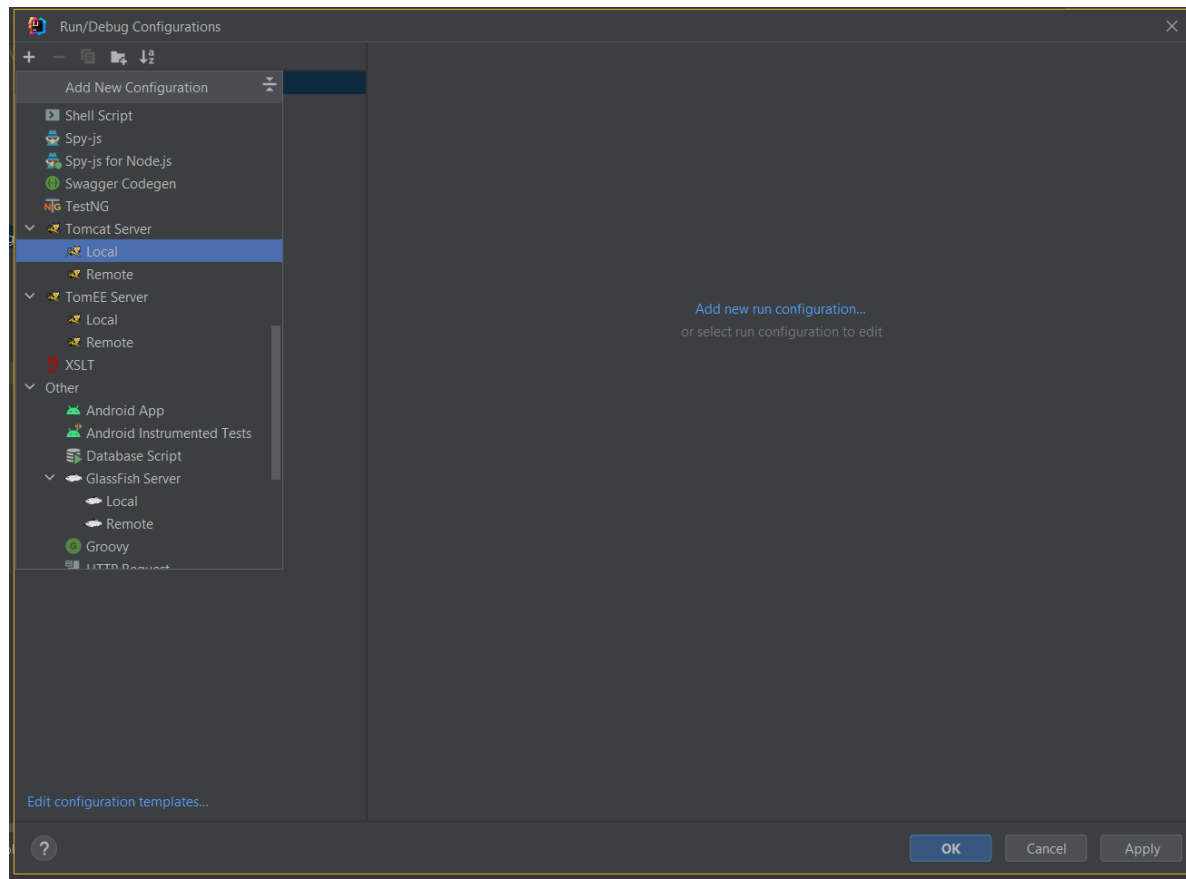
Tomcat Server -> Local

(локальный сервер, т.к.

работать будет на этом же компьютере).



Не перепутайте Tomcat Server и TomEE Server!



# Задание

4 Дайте название серверу.

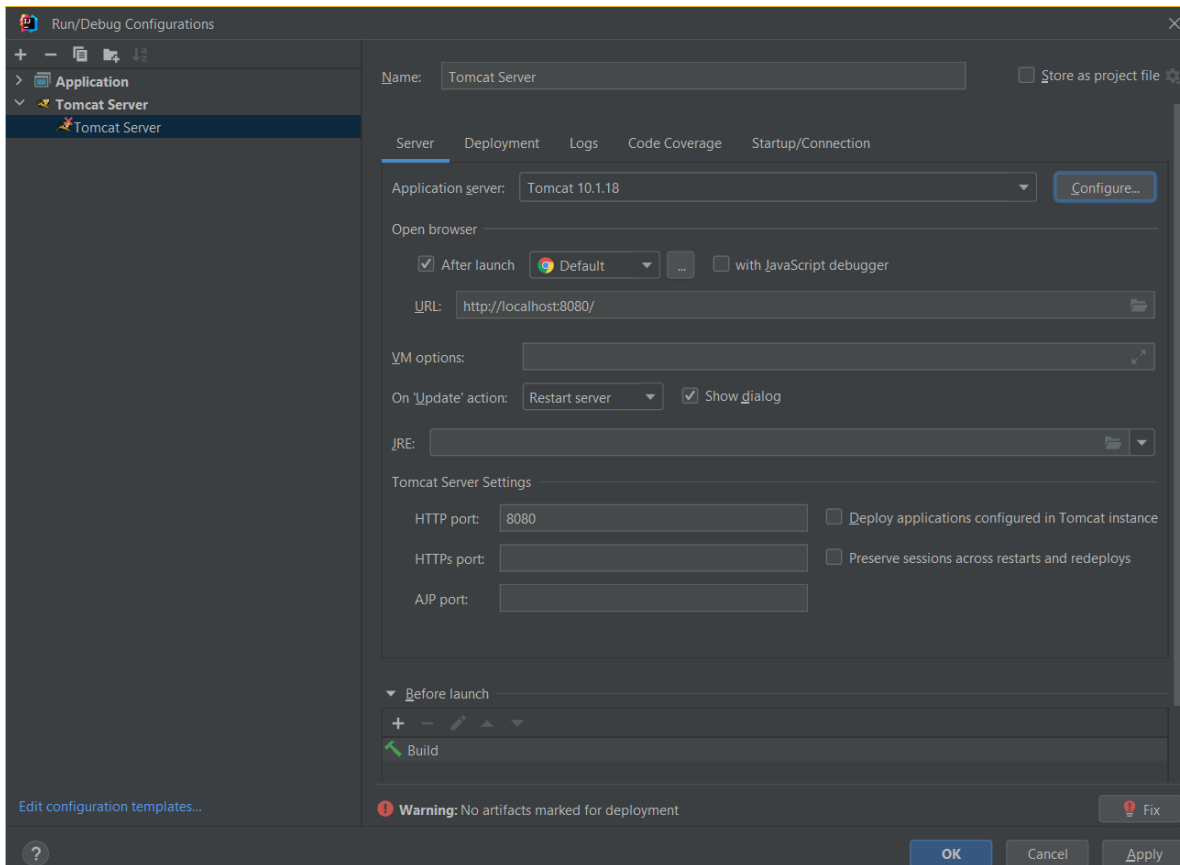
Укажите путь к

установленному ранее

серверу Tomcat, нажав на

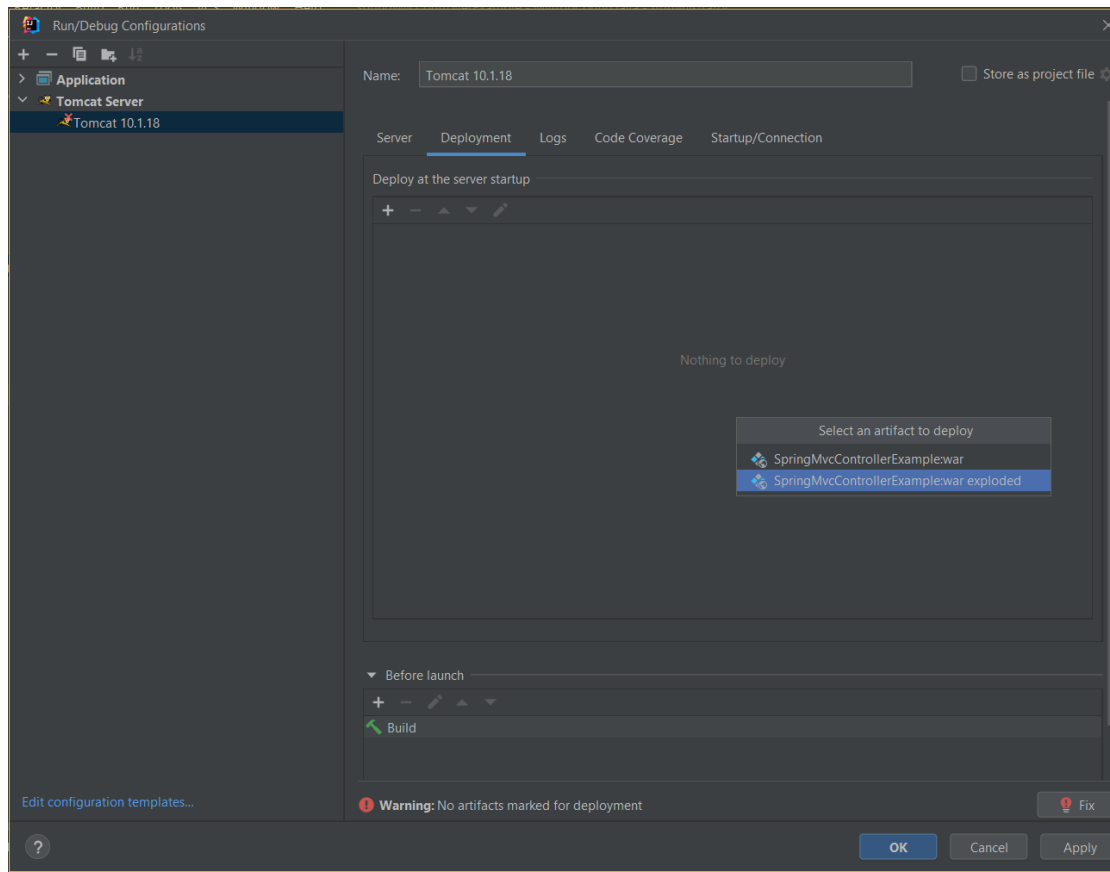
*Configuration*. Нажмите кнопку

*Fix*.



# Задание

5 Выберите вариант с *war exploded*. (*war* – это архив для веб-проектов). *Exploded* позволяет обновлять классы «на лету», не перестраивая весь архив.



# Задание

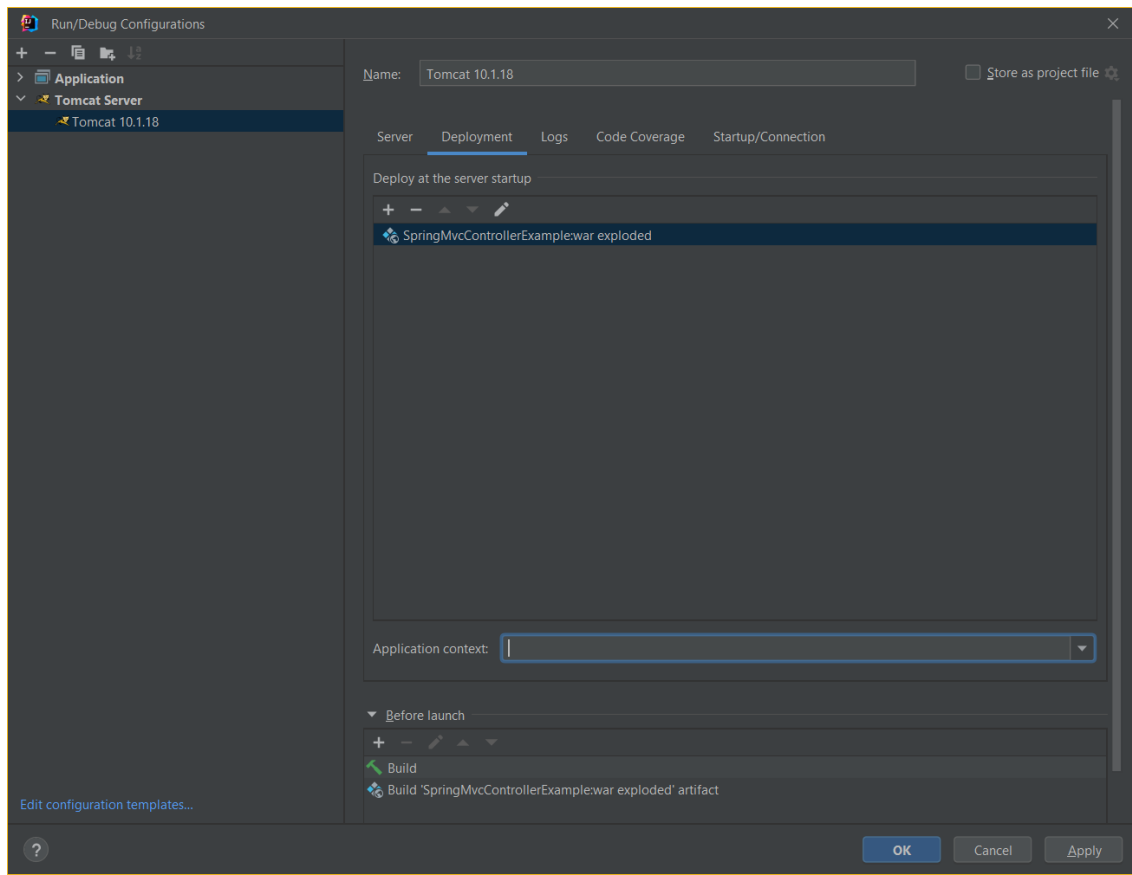
6 В поле Application context оставьте сделайте пустую строку, чтобы при обращении к нашему серверу после хоста не требовалось дописывать эту часть пути.

7 Нажимаем OK.



Перезапускаем IDE.

8 Запустите приложение (плей справа наверху). Если всё прошло успешно, то откроется браузер на странице <http://localhost:8080/>



# Задание

1 Добавьте зависимости на Spring Core, Spring Context

<https://mvnrepository.com/artifact/org.springframework/spring-core>

<https://mvnrepository.com/artifact/org.springframework/spring-context>

2 Добавьте зависимости Spring MVC

<https://mvnrepository.com/artifact/org.springframework/spring-web>

<https://mvnrepository.com/artifact/org.springframework/spring-webmvc>



Версии всех зависимостей Spring должны совпадать. Можно создать переменную в pom.xml:

```
<properties>  
  <spring.version>6.1.3</spring.version>  
</properties>
```

Переменную можно использовать для указания версии зависимости:

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-core</artifactId>  
  <version>${spring.version}</version>  
</dependency>
```

# Вебненький

# Конфигурация web-сервера

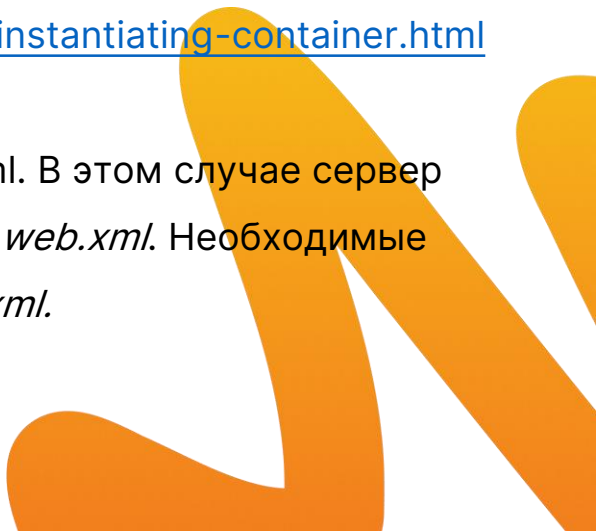
Конфигурацию web-сервера можно выполнить разными способами:

- xml-конфигурация
- Java-код конфигурация и аннотации.

Пример xml-конфигурации с комментариями можно найти здесь:

<https://docs.spring.io/spring-framework/reference/core/beans/java/instantiating-container.html>

Сегодня мы начнём конфигурировать Spring MVC с помощью xml. В этом случае сервер считает необходимые данные для создания *DispatcherServlet* из *web.xml*. Необходимые бины будут описаны в отдельном файле *applicationContextMVC.xml*.



# Задание

- 1 Заполните web.xml в соответствии с примером.
- 2 Рядом с web.xml создайте файл applicationContextMVC.xml, заполните его в соответствии с примером.
- 3 Попробуйте запустить приложение.

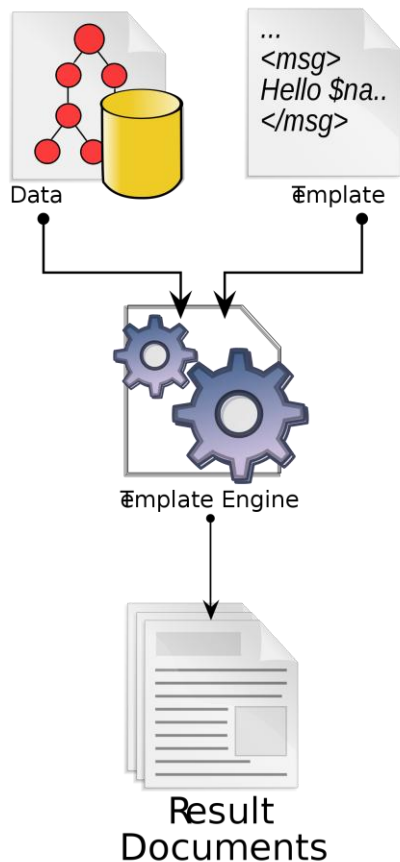


SpringMvcControllerExample.zip





# Вебненький Представление



За генерацию представления обычно отвечает frontend-приложение. Тем не менее, backend-может брать на себя роль создателя приложения. Для этого мы будем использовать шаблонизатор.

**Шаблонизатор** (*template engine*) – модуль приложения, который помогает формировать представления для клиента (чаще всего html, но также SVG, WML или XML). Шаблонизаторы позволяют веб-разработчикам создавать содержимое страниц, состоящее из *статических исходных данных* (скелет страницы) и java-элементов, которые *динамически принимают* значения в зависимости от переданного состояния модели.

# Вебненький Представление

Среди шаблонизаторов выделяют

- шаблоны разметки на языке **Groovy**;
- **JSP** (*JavaServer Pages*) – платформенно-независимая, переносимая и легко расширяемая технология для разработки веб-приложений, работающая на JVM;
- [Thymeleaf](#) – современный шаблонизатор Java на стороне сервера, который делает упор на естественные *HTML*-шаблоны, которые можно предварительно просмотреть в браузере двойным щелчком мыши, что очень удобно при самостоятельной работе над шаблонами пользовательского интерфейса без необходимости наличия работающего сервера.



# Задание

1 Добавьте шаблонизатор Thymeleaf Spring6

<https://mvnrepository.com/artifact/org.thymeleaf/thymeleaf-spring6>

2 Создайте каталог views в каталоге WEB-INF.

3 В каталоге views создайте страницу show\_truth.html. И заполните её как html-страницу.

Пример:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="ISO-8859-1">
    <title>First app</title>
  </head>
  <body>
    <h2>The truth is you're cool!</h2>
  </body>
</html>
```



# Вебненький @Controller

**@Controller** – аннотация, помечающая Java-класс, обеспечивающий

- обработку запроса от пользователя
- обмен данными с моделью
- передачу пользователю правильного представления состояния модели
- перенаправление пользователя на другой URL.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {

    @AliasFor(annotation = Component.class)
    String value() default "";
}
```

## Методы класса с @Controller

Методы контроллера помечаются аннотациями, уточняющими метод запроса (GET, POST, DELETE и т.д.):

- *@GetMapping*
- *@PostMapping*
- *@PutMapping*
- *@DeleteMapping*
- *@PatchMapping*,

а также путь в URL.

```
@Controller
public class UserController {
    @GetMapping("/users")
    public List<User> getUsers() { ... }

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable("id") long id) { ... }

    @PostMapping(path = "/users",
        consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public User createUser(UserDetails details) { ... }
}
```

Если в пути есть переменная часть, то её заключают в фигурные скобки {}, а перед аргументом метода указывают аннотацию *@PathVariable("имя переменной")*.

Вебненький


# Методы класса с @Controller

Другой способ связать метод класса-контроллера с методами протокола http – это аннотация *@RequestMapping(value = "путь", method = RequestMethod.ИМЯ\_МЕТОДА)*

```
@RequestMapping(value = "/users", method = RequestMethod.GET)
public Users getUsers() { ... }
```

```
@RequestMapping(value = "/users/{id}", method = RequestMethod.GET)
public User getUser(@PathVariable("id") String id) { ... }
```

```
@RequestMapping(value = "/users", method = RequestMethod.POST)
public User createUser(UserDetails details) { ... }
```

A large, stylized orange graphic resembling a thick, curved line or a stylized letter 'N' is positioned in the bottom right corner of the slide.

Вебненький

# Методы класса с @Controller

*@RequestMapping* позволяет также задавать атрибуты на уровне класса и транслировать их в аннотации методов (в т.ч. такие, как *@RequestMapping*).

```
@Controller
@RequestMapping(path = "/", produces = MediaType.APPLICATION_JSON_VALUE)
public class HomeController
{
    @PostMapping(path = "/members")
    public void addMember(@RequestBody Member member) {
        // code
    }
}
```

# Задание

- 1 Создайте класс FirstController и пометьте его аннотацией @Controller.
- 2 В классе создайте GET-запрос, ответом на который станет ранее созданный файл show\_truth.html. С учётом предыдущих настроек, достаточно вернуть из метода имя нужного представления (без пути, без расширения файла).
- 3 Запустите приложение и перейдите по URL, чтобы получить страницу show\_truth.html от сервера.
- 4 В приложение нужно добавить функциональность «говорить ложь». Доработайте приложение.





Вебненький

# Управление представлением

*@ModelAttribute* позволяет



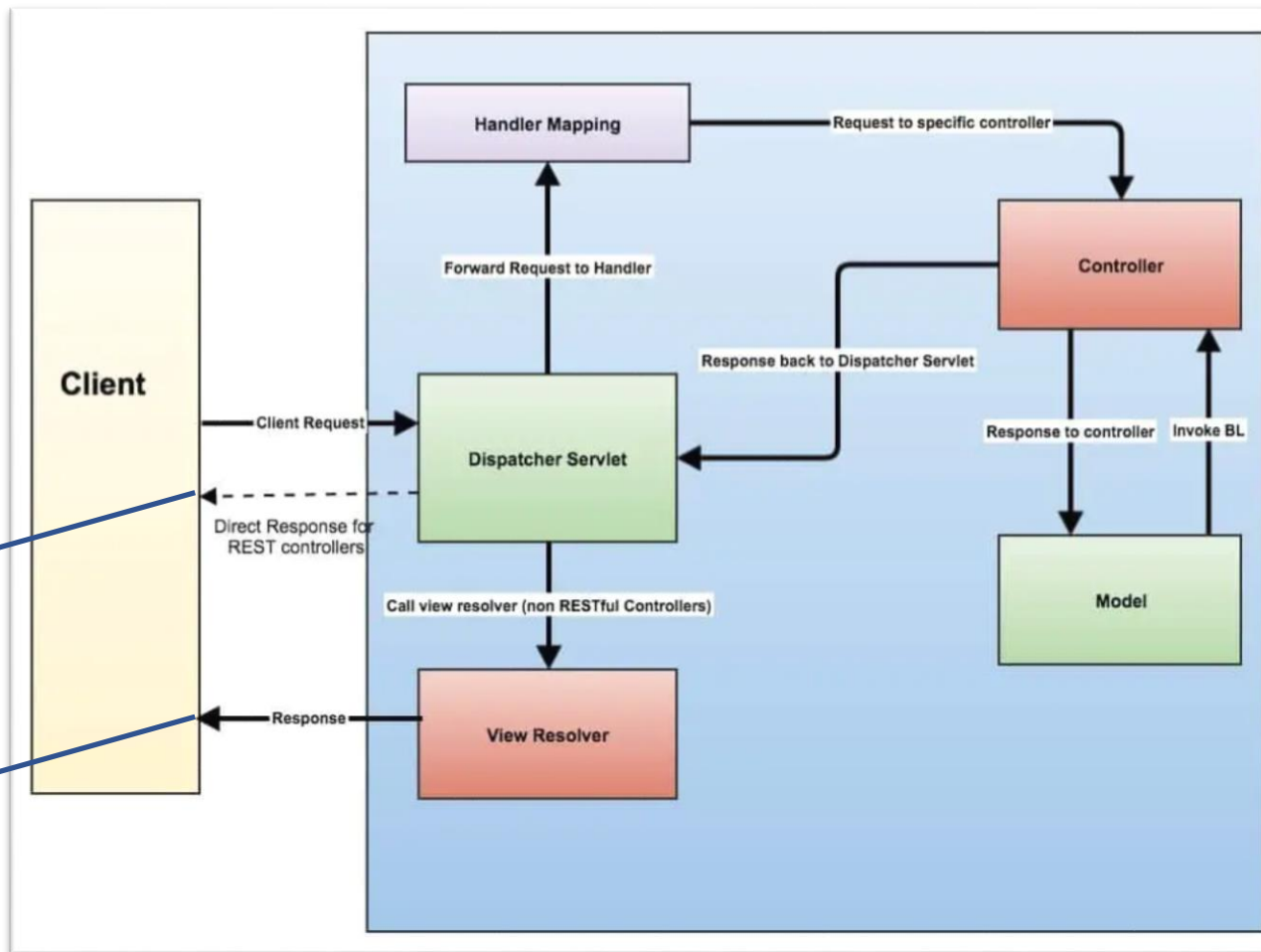
# Проблема

Перед вами архитектура  
Spring MVC приложения.

В чём отличие ответов 1 и  
2?

1

2



# Полный Rest

## Spring MVC vs REST

*DispatcherServlet* выполняет роль Front-контроллера. Традиционно Spring MVC-приложение предполагает наличие *ViewResolver*, который подготавливает представление, направляемое клиенту.

В отличие от традиционного Spring MVC-приложения *RESTful*-приложение возвращает сырые данные от Модели (*JSON*, *XML* и т.д.) и не требует участия *ViewRender*, т.к. клиентское приложение будет отвечать за представление данных из ответа.

Если требуется из всех методов контроллера возвращать объект доменной модели вместо представления, то вместо аннотации *@Controller* используют аннотацию *@RestController*.



# Полный Rest

## Spring MVC vs REST



*@Controller* – помечает, что класс является контроллером, возвращающим представления. Если какой-либо метод контроллера возвращает объект в виде JSON, то требуется использовать аннотацию *@ResponseBody* над методом. Для *@RestController* этого не требуется.

Если метод любого из контроллеров принимает аргументы, то нужно сопоставить, откуда брать данные для аргументов, поэтому каждый аргумент помечается одной из аннотаций:

- *@RequestBody* – взять из тела сообщения;
- *@RequestParam*("Имя\_параметра") – из параметра URL;
- *@PathVariable*("Имя\_переменной") – из переменной пути;
- *@RequestHeader*("Имя\_заголовка") – из заголовка;
- *@RequestHeader* – взять все заголовки в виде коллекции *MultiValueMap* или экземпляра *HttpHeaders*.

# Полный Rest Spring MVC vs REST

Некоторые из аннотаций поддерживают атрибут *required*, показывающий, строго ли требуется наличие данного элемента в запросе. А также параметр *defaultValue*, если нужно задать значение по умолчанию при отсутствующем атрибуте.

```
@GetMapping("/users")
public List<User> getUsers(
    @RequestParam(
        value = "lastNameStartsWith",
        required = false,
        defaultValue = "") String lastNameStartsWith
    ) {
    return userService.getUsers(lastNameStartsWith);
}
```

Полный Rest

# Пример использования аннотаций в контроллере



SpringMvcControllerAnnotationsExample.zip



# Полный Rest

## Бины запроса и ответа



Контроллеры позволяют работать с запросом и ответом как с объектами Java, что бывает удобно при тонкой настройке ответа или извлечения большого количества данных из запроса.

```
@RestController
public class SessionDemoController {

    private static final Logger LOG = LoggerFactory.getLogger(SessionDemoController.class);

    @GetMapping("/get-session-count")
    public String testSessionListner(HttpServletRequest request, HttpServletResponse response){

        HttpSession session = request.getSession(false);
        if(session == null){
            LOG.info("Unable to find session. Creating a new session");
            session = request.getSession(true);
        }
        return "Session Test completed";
    }
}
```

# Сервис

# Сервисы



Аннотацией *@Service* помечаются классы, выполняющие логику работы Модели. Сервисы получают данные от контроллера и, объединяя работу разных классов модели (в том числе и других сервисов при необходимости), получают результат вычислений – новое состояние модели.

С уровня сервисов осуществляется работа в том числе с сущностями, которые взаимодействуют с базой данных.

```
@Service
public class UserService {
    private final List<User> users;

    public UserService() {
        users = new ArrayList<>();
    }
    public List<User> getUsers(String lastNameStartsWith) {
        return lastNameStartsWith.isBlank() ? new ArrayList<>(users) : users.stream()
            .filter(u -> u.lastName().startsWith(lastNameStartsWith))
            .toList();
    }
}
```



3

# Домашнее задание

# Домашнее задание

Создайте простое веб-приложение, которое принимает POST-запрос с текстом и добавляет этот текст в список. В ответ на POST запрос приложение отправляет представление с текстом «Ваше сообщение принято». По GET-запросу приложение возвращает список всех сообщений в виде JSON.



# ЗАКЛЮЧЕНИЕ

