Spring Boot & REST

Spring Boot

- When deploying to the cloud it's easier to have a **single executable** to deploy
- Spring Boot aims to facilitate this:
  - Create a full application in one executable JAR
    - Including embedded Tomcat Web Server
  - Make it quick and easy to create such applications
    - Using **convention over configuration**

Opinionated View

- The official documentation says Spring Boot takes "an opinionated view of the Spring platform and third party libraries so you can **get started with minimal fuss**"

- These opinions are the conventions
  - So you have to do less configuration

Do less and accomplish more

Why not Earlier in the Course?

- Although Spring Boot is awesome
  - It **hides the details** of what is going on

- So far we've manually configured everything
  - So that you know how they work
  - If it breaks you are better equipped to fix things

Typical Spring Boot POM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>edu.mum.cs544</groupId>
	<artifactId>demo</artifactId>
	<version>0.0.1-SNAPSHOT</version>

	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.1.5.RELEASE</version>
	</parent>
	<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>
	</dependencies>
	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
</project>
```

Inherit POM config from Spring Boot

Typical dependencies for a web application

Package as an executable jar file

Spring Boot Starters

Table 13.1. Spring Boot application starters

Name	Description	Pom
`spring-boot-starter`	Core starter, including auto-configuration support, logging and YAML	Pom
`spring-boot-starter-activemq`	Starter for JMS messaging using Apache ActiveMQ	Pom
`spring-boot-starter-amqp`	Starter for using Spring AMQP and Rabbit MQ	Pom
`spring-boot-starter-aop`	Starter for aspect-oriented programming with Spring AOP and AspectJ	Pom
`spring-boot-starter-artemis`	Starter for JMS messaging using Apache Artemis	Pom
`spring-boot-starter-batch`	Starter for using Spring Batch	Pom
`spring-boot-starter-cache`	Starter for using Spring Framework's caching support	Pom
`spring-boot-starter-cloud-connectors`	Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku	Pom
`spring-boot-starter-data-cassandra`	Starter for using Cassandra distributed database and Spring Data Cassandra	Pom
`spring-boot-starter-data-cassandra-reactive`	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive	Pom
`spring-boot-starter-data-couchbase`	Starter for using Couchbase document-oriented database and Spring Data Couchbase	Pom
`spring-boot-starter-data-couchbase-reactive`	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive	Pom
`spring-boot-starter-data-elasticsearch`	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch	Pom
`spring-boot-starter-data-jpa`	Starter for using Spring Data JPA with Hibernate	Pom
`spring-boot-starter-data-ldap`	Starter for using Spring Data LDAP	Pom
`spring-boot-starter-data-mongodb`	Starter for using MongoDB document-oriented database and Spring Data MongoDB	Pom
`spring-boot-starter-data-mongodb-reactive`	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive	Pom
`spring-boot-starter-data-neo4j`	Starter for using Neo4j graph database and Spring Data Neo4j	Pom
`spring-boot-starter-data-redis`	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce	Pom

- There are many starters to choose
- You can also combine starters
- And add your own additional deps

See: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application-code

Version Numbers

- Best **not to give a version** to dependencies
- Spring Boot has will automatically select the best version to work with (based on parent)

```
<!-- needed for Spring MVC -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- needed for JSP -->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

No version on any dependency

Code & Config

- Make a class with a **main()** your **@Configuation**
  - Spring boot prefers Java Config
  - Spring boot requires a main() method
    - Starts your Spring Boot application

```
@Configuration
@ComponentScan
@EnableAutoConfiguration
@EnableWebSecurity
public class Application {

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

Spring Boot annotation to have it look through your JARs and configure it self based on what it finds

SpringApplication.run() is a Spring Boot method that requires the name of your primary configuration class and starts the application

@SpringBootApplication

- @SpringBootApplication is a **composite** of:
  - @Configuration
  - @ComponentScan
  - @EnableAutoConfiguration

Since all Spring Boot applications generally need all 3, might as well create a single composite

```
@SpringBootApplication
@EnableWebSecurity
public class Application {

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

Properties or YML

- Auto-configuration can't do everything
  - **Needs certain values** such as DB user / password

- These values can be stored in:
  - application.properties

Inside your resources dir

Examples

application.properties

```
spring.datasource.url = jdbc:mysql://localhost/cs544
spring.datasource.username = root
spring.datasource.password = root

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
spring.jpa.hibernate.ddl-auto = create-drop

spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp

logging.level.root=DEBUG
```

Additional Configuration

- Autoconfig takes care of most things
  - But if needed **you can add / override** config

- Can be in main or in extra config files:
  - @Import to include @Configuration files
  - @ImportResource can include XML config files

Profiles & External Configuration

- Spring Boot has **profile support**
  - Parts of your internal (in-project) configuration are active only in certain environments (dev / test / production / …)
- External config can be picked up in many ways
  - To indicate which environment you are in
  - Or to overwrite internal config values

Profiles

- Multiple profiles can be active, indicated by:

```
spring.profiles.active=dev,mysql
```

- Then includes profile specific properties files:

```
application-dev.properties
application-mysql.properties
```

- And activates configuration annotated for it

```
@Configuration
@Profile("production")
public class ProductionConfiguration {
```

Order of precedence

External Config Options

1) Devtools global settings properties on your home directory (~/.spring-boot-devtools.properties when devtools is active).
2) @TestPropertySource annotations on your tests.
3) properties attribute on your tests. Available on @SpringBootTest and the test annotations for testing a particular slice of your application.
4) Command line arguments.
5) Properties from SPRING_APPLICATION_JSON (inline JSON embedded in an environment variable or system property).
6) ServletConfig init parameters.
7) ServletContext init parameters.
8) JNDI attributes from java:comp/env.
9) Java System properties (System.getProperties()).
10) OS environment variables.
11) A RandomValuePropertySource that has properties only in random.*.
12) Profile-specific application properties outside of your packaged jar (application-{profile}.properties and YAML variants).
13) Profile-specific application properties packaged inside your jar (application-{profile}.properties and YAML variants).
14) Application properties outside of your packaged jar (application.properties and YAML variants).
15) Application properties packaged inside your jar (application.properties and YAML variants).
16) @PropertySource annotations on your @Configuration classes.
17) Default properties (specified by setting SpringApplication.setDefaultProperties).

Command Line
Specifying Profile(s)

- You can start a Spring Boot application as:

```
java -jar app.jar –spring-profiles-active=prod
```

- Or you can set a environment variable

```
SET SPRING_PROFILES_ACTIVE=prod
java -jar app.jar
```

Running

- During development run it in your IDE
  - Or from its own Maven target

$mvn spring-boot:run

- Include Spring **Boot DevTools**
  - Not included when packaging the JAR

Will not affect production env.

```
<!-- automatically reloads app when development files change -->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-devtools</artifactId>
   <optional>true</optional>
</dependency>
```

Optional makes it not go into JAR

Dev Tools Benefits

- Automatically **restarts** when it senses a change
  - On restart logs changes to autoconfig
- Option to configure resources:
  - That should not trigger a restart when changed
  - Or watch files outside the project that should trigger

Banner

Not practical, but kind of fun

- When starting Spring Boot shows a banner

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v2.1.5.RELEASE
```

- Customize the banner by adding to resources dir:
  - **banner.txt** (or .jpg, .gif, .png)
  - Or set spring.banner.location
  - Or turn it off with spring.main.banner-mode: off

Images are converted to ASCII

Summary

- Spring Boot uses convention over configuration
  - Use property for remaining values
- Provides dependency management through starters
  - You never specify version numbers
- Uses a @SprinBootApplication with a main() method
  - Which invokes SpringApplication.run()
- Has support for multiple configuration profiles
  - That can be activated through external configuration

CS544 EA

REST

Remote Invocation

- There are many forms of Remote Invocation
  - CORBA, RMI, XML-RPC, SOAP, …
  - Web services can be seen as remote invocation

- The most popular form of WebService today are **RESTful webservices**

Web Service

- A web service offers functionality that can be called by clients using standard protocols

- **RESTful** web services use **HTTP**
  - Are officially payload (data) agnostic
  - Most popular data format: **JSON**

HTTP

- With web pages we use GET and POST
  - GET to retrieve
  - POST to insert / update data
- RESTful uses **more HTTP methods**:
  - GET to *retrieve* data
  - POST to *create* (a new URI is assigned)
  - PUT to *replace* data at a URI
  - PATCH to *update* or *create* data at a URI
  - DELETE to *delete* data and its URI

JSON

- The JavaScript Object Notation
  - Created by Douglas Crockford
    JavaScript: the Good Parts
  - Human readable
    attribute / value pairs
  - A lightweight alternative to XML

```
{
        "firstName": "John",
        "lastName": "Smith",
        "isAlive": true,
        "age": 27,
        "address": {
                "streetAddress": "21 2nd Street",
                "city": "New York",
                "state": "NY",
                "postalCode": "10021-3100"
        },
        "phoneNumbers": [
                {
                        "type": "home",
                        "number": "212 555-1234"
                },
        ],
}
```

Spring MVC

- Spring MVC built-in support for RESTful web services

```
<dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
</dependency>
```

- When the Jackson library is on the classpath
  - Automatically configures the bean: `MappingJackson2HTTPMessageConverter`
    - Able to convert Java Objects to JSON text
    - Able to convert JSON text to Java Objects

@RestController is a composite of @Controller and @ResponseBody

Controller

```
@RestController
public class PersonController {
	@Autowired
	private PersonService personService;

	@GetMapping(value="/person/", produces="application/json")
	public List<Person> getAll() {
		return personService.getAll();
	}

	@GetMapping(value= "/person/{id}", produces = "application/json")
	public Person get(@PathVariable long id) {
		return personService.get(id);
	}

	@PostMapping(value="/person/", consumes = "application/json")
	public RedirectView post(@RequestBody Person person) {
		long id = personService.add(person);
		return new RedirectView("/person/" + id);
	}

	@PutMapping(value= "/person/{id}", consumes = "application/json")
	public void put(@PathVariable long id, @RequestBody Person person) {
		if (id != person.getId()) { throw new IllegalArgumentException(); }
		personService.update(person);
	}

	@DeleteMapping("/person/{id}")
	public void delete(@PathVariable long id) {
		personService.delete(id);
	}
}
```

Produces / Consumes optional

ResponseBody becomes like: {“id”: 1, “name”: “Test”, “age”: 28}

RedirectView “redirect: ” String won’t work!

RequestBody expected like: {“name”: “Other”, “age”: 27}

Person & Person Service

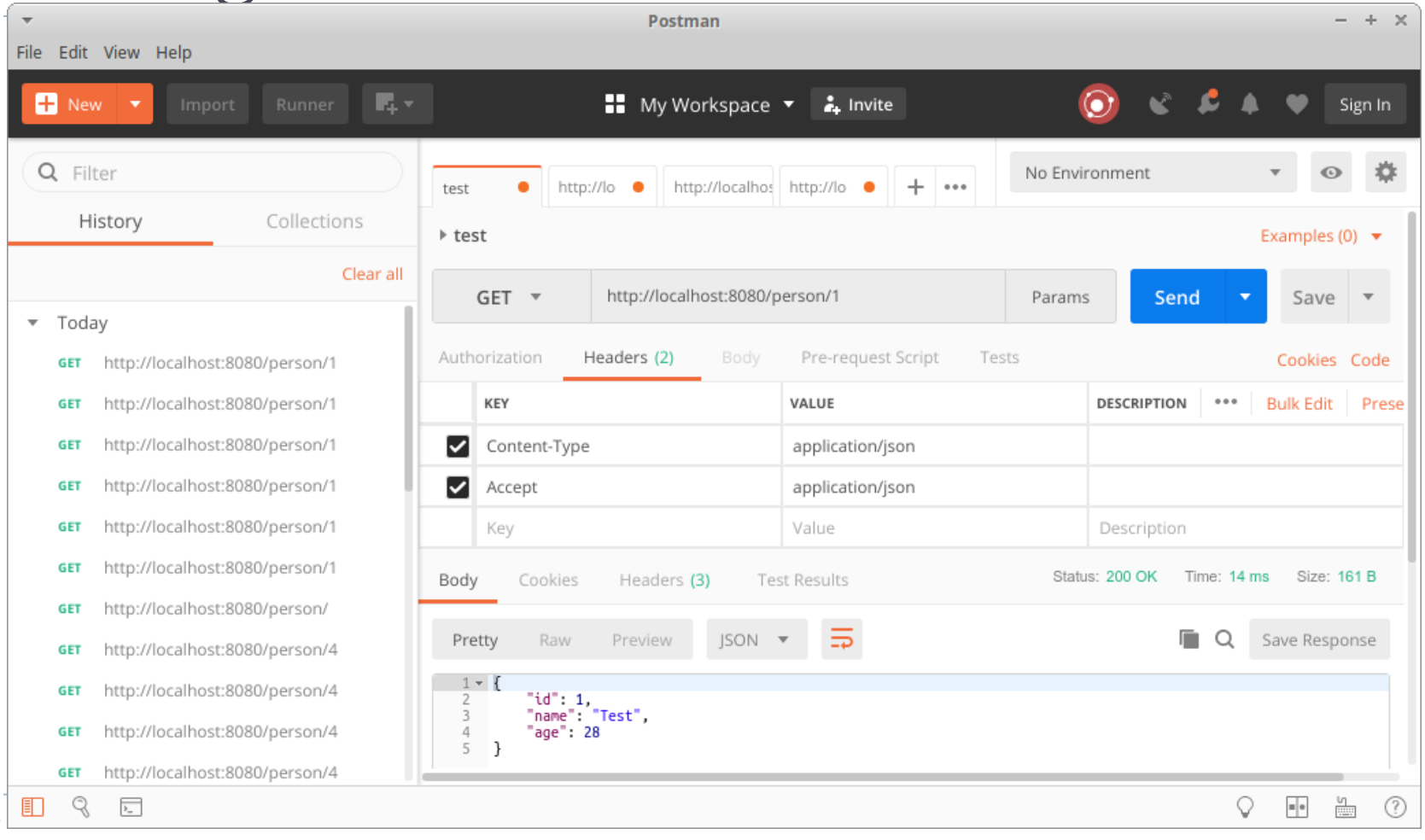
```java
@JsonIgnoreProperties({"hibernateLazyInitializer"})
@Setter
@Getter
@NoArgsConstructor
@Entity
public class Person {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private int age;

}
```

```java
public interface PersonService {
        Person get(Long id);
        List<Person> getAll();
        Long add(Person p);
        void update(Person p);
        void delete(Long id);
}
```

```java
@Service
@Transactional
public class PersonServiceImpl implements PersonService {

    @Autowired
    private PersonRepository personRepository;

    @Override
    public Person get(Long id) {
        return personRepository.getOne(id);
    }

    @Override
    public List<Person> getAll() {
        return personRepository.findAll();
    }

    @Override
    public Long add(Person p) {
        personRepository.save(p);
        return p.getId();
    }

    @Override
    public void update(Person p) {
        personRepository.save(p);
    }

    @Override
    public void delete(Long id) {
        personRepository.deleteById(id);
    }
}
```

Testing

Postman is popular for testing webservices



Send / Receive XML

- Enabling XML is simply adding a dependency

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

- With this added our webservice will be able to do both JSON and XML

Content Negotiation

- HTTP headers are used to indicate what the incoming and outgoing data types should be
  - **Content-Type** indicates what the browser sends
  - **Accept** indicates what the browser wants

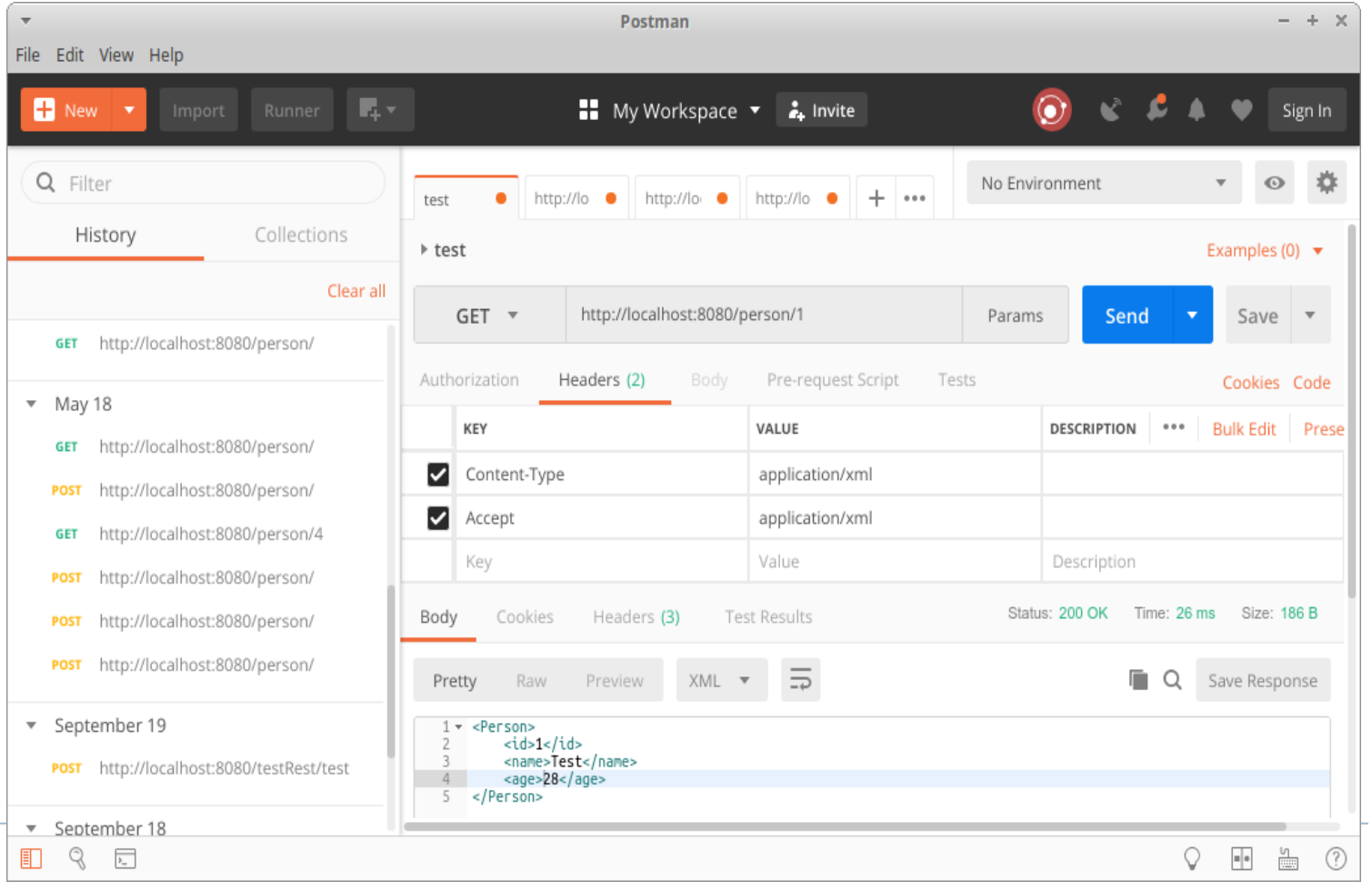
- The mime type for XML is:
  - application/xml

Controller

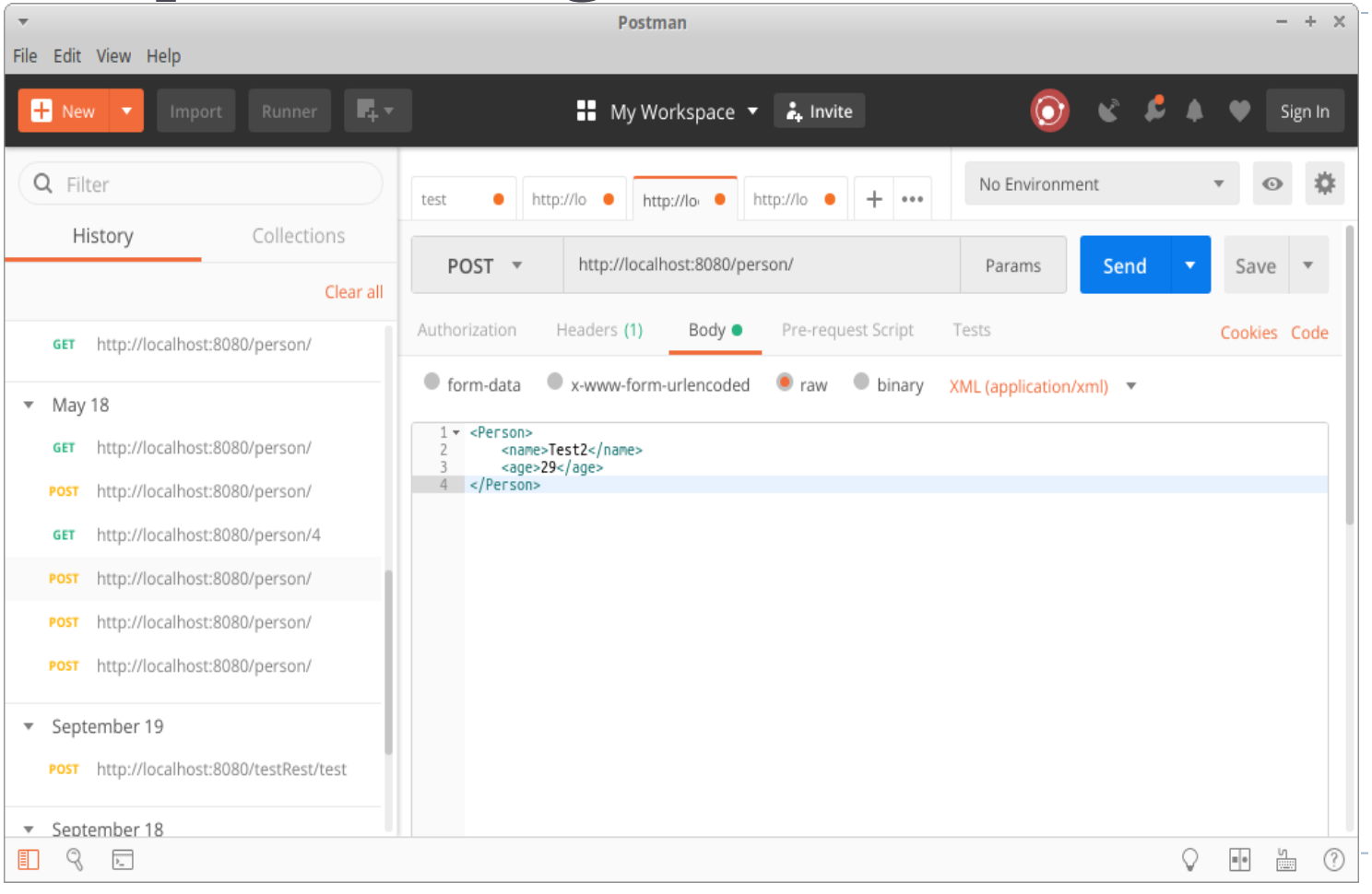
```
@RestController
public class PersonController {
	@Autowired
	private PersonService personService;

	@GetMapping("/person/")
	public List<Person> getAll() {
		return
		personService.getAll();
	}

	@GetMapping("/person/{id}")
	public Person get(@PathVariable long id) {
		return personService.get(id);
	}

	@PostMapping("/person/")
	public RedirectView post(@RequestBody Person person) {
		long id = personService.add(person);
		return new RedirectView("/person/" + id);
	}

	@PutMapping("/person/{id}")
	public void put(@PathVariable long id, @RequestBody Person person) {
		if (id != person.getId()) { throw new
		IllegalArgumentException(); }  personService.update(person);
	}

	@DeleteMapping("/person/{id}")
	public void delete(@PathVariable long
		id) {  personService.delete(id);
	}
}
```

By not specifying produces / consumes a controller method can be used for both XML and JSON

Example Receiving



Example Sending



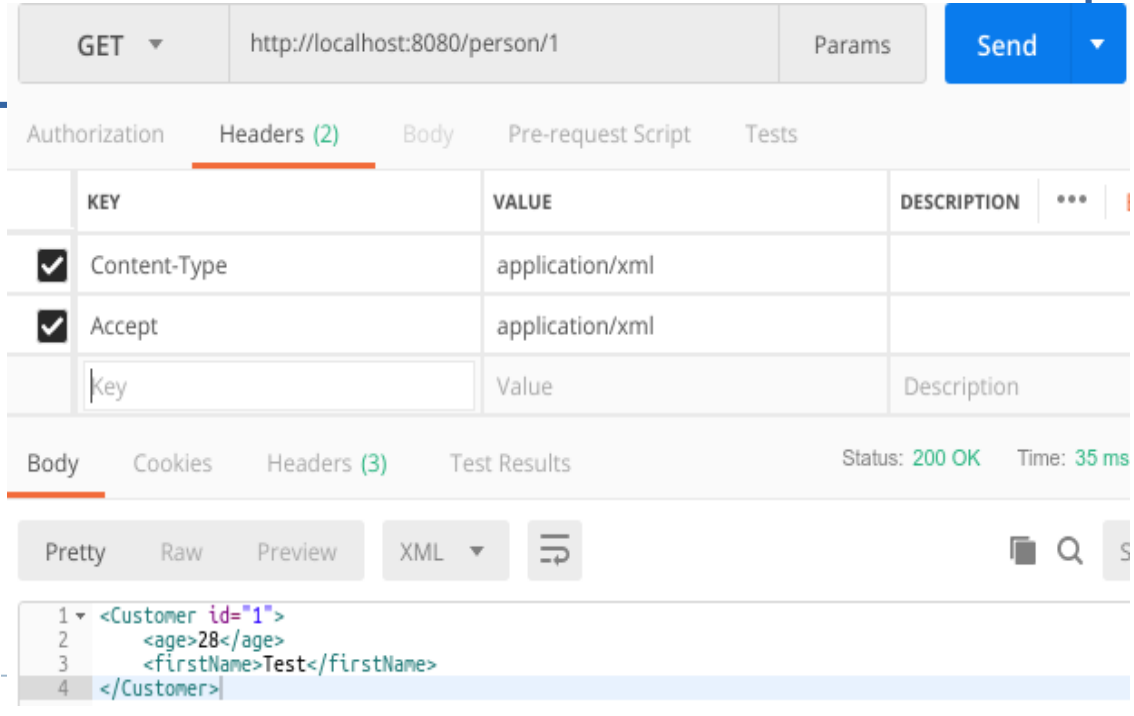
JAXB Mapping

- The Object to XML Mapping (OXM) defaults to:
  - Class get its own tag
  - Every property gets its own tag inside it
  - An object inside an object is nested

- If you want to change the name of a tag
  - Or use an XML attribute
  - Map them with annotations

Basic OXM Mapping

```java
@JacksonXmlRootElement(localName = "Customer")
public class Person {
    @JacksonXmlProperty(isAttribute = true)
    private Long id;
    @JacksonXmlProperty(localName = "firstName")
    private String name;
    private int age;
```



RestTemplate

- Spring provides RestTemplate for REST clients

Method	Description
`.getForObject()`	Sends a GET, returns a Java Object based on response body
`.getForEntity()`	Sends a GET, returns ResponseEntity (status/header/body)
`.postForLocation()`	Sends a POST, returns a URI (Location header from redirect)
`.postForObject()`	Sends a POST, returns a Java Object based on response body
`.postForEntity()`	Sends a POST, returns a ResponseEntity (status/header/body)
`.put()`	Creates or updates a resource using PUT
`.patchForObject()`	Sends a PATCH, returns Java Object
`.delete()`	Deletes the resource at the URI using DELETE
`.exchange()`	General / flexible method using RequestEntity / ReponseEntity

Needs extra HTTP lib to work

Lowerlevel .execute() method is also present, which we won't cover

Console Client Config

```java
@SpringBootApplication
public class ConsoleConfig {
	@Bean
	public RestTemplate getRestTemplate() {
		return new RestTemplate();
	}

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

Create the RestTemplate bean

Spring Boot executes all beans that implement CommandLineRunner

```java
@Component
public class ConsoleApp implements CommandLineRunner {
	@Autowired
	private PersonService personService;

	@Override
	public void run(String... args) {
		Person p = personService.get(1L);
		personService.add(new Person("Hello", 22));
		System.out.println(personService.getAll());
		p.setAge(33);
		personService.update(p);
		System.out.println(personService.getAll());
		personService.delete(2L);
		System.out.println(personService.getAll());
		p = personService.getAll().get(0);
		System.out.println(p.getName());
	}
}
```

pom.xml dependencies

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

application.properties

```
spring.main.web-application-type=NONE
```

Example

```java
@Service
public class PersonServiceProxy implements PersonService {
	@Autowired
	private RestTemplate restTemplate;
	private final String personUrl = "http://localhost:8080/person/{id}";
	private final String pplUrl = "http://localhost:8080/person/";

	@Override
	public Person get(Long id) {
		return restTemplate.getForObject(personUrl, Person.class, id);
	}
	@Override
	public List<Person> getAll() {
		ResponseEntity<List<Person>> response =
				restTemplate.exchange(pplUrl, HttpMethod.GET, null,
					new ParameterizedTypeReference<List<Person>>() {});
		return response.getBody();
	}
	@Override
	public Long add(Person p) {
		URI uri = restTemplate.postForLocation(pplUrl, p);
		if (uri == null) { return null; }
		Matcher m = Pattern.compile(".*/person/(\\d+)").matcher(uri.getPath());
		m.matches();
		return Long.parseLong(m.group(1));
	}
	@Override
	public void update(Person p) {
		restTemplate.put(personUrl, p, p.getId());
	}
	@Override
	public void delete(Long id) {
		restTemplate.delete(personUrl, id);
	}
}
```

Same PersonService interface

.getForObject(url, return type, ...path params)

Have to use .exchange() because .getForObject() cannot specify List<Person> as return type! (only List)

.exchange(url, method, reqData, respType)

.postForLocation(url, reqData)

.put(url, reqData, ...path params)

.delete(url, ...path params)

Summary

- RESTful webservices use HTTP, the most popular data format is JSON (other possible)
- Easy to setup with SpringMVC: include Jackson and use @ResponseBody / @RequestBody
- RestTemplate provides convenient methods to make calls to RESTful webservices
  - Best to wrap usage inside a service-proxy