TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

SWEN 301 : Scalable Software Development

Web Services

Jens Dietrich **(jens.dietrich@vuw.ac.nz)**

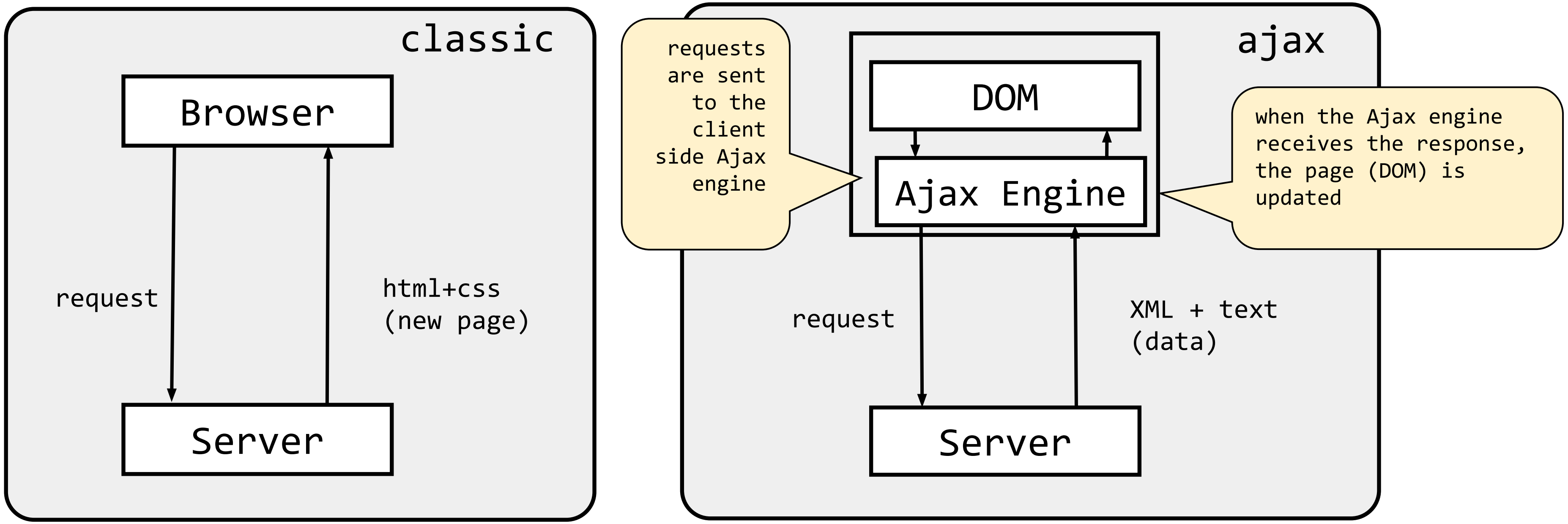
Overview

- from Websites to Services: AJAX
- history: CORBA and SOAP+WSDL
- HTTP services / REST
- serialising data: JSON & co
- testing HTTP Services, mocking
- service frameworks case studies: Javelin, Jersey and Jackson
- specifying APIs

From Webpages to Services: AJAX

- technology emerging around 2003 to make websites more interactive
- instead of replacing the entire page, JavaScript is used to fetch data and update the web page (by manipulating the Document Object Model - DOM)
- this means that web sites could gradually evolve without reloading, leading to "1-page web applications"
- used successfully first by gmail (2004) and Google maps (2005)
- AJAX = Asynchronous Javascript and XML
- however, XML is now often replaced by other formats (JSON or similar)

AJAX Principles


classic
Browser
request
html+css
(new page)
Server
ajax
requests are sent to the client side Ajax engine
DOM
Ajax Engine
when the Ajax engine receives the response, the page (DOM) is updated
request
XML + text
(data)
Server

- the ajax engine consists of a browser dependent **XMLHttpRequest** object
- the W3C standardizes this object: http://www.w3.org/TR/XMLHttpRequest/

Raw AJAX Example: Client (index.html)

```
<script language="javascript" type="text/javascript">
   var xmlHttp = new XMLHttpRequest();
   function check4matchingNames() {
      var t = document.getElementById("name").value;
      if ((t == null) || (t == "")) return;
      var url = "firstNames?name=" + t;
      xmlHttp.open("GET", url, true);
      xmlHttp.onreadystatechange = updatePage;
         xmlHttp.send(null);
      }
      function updatePage() {
          ...
```

AJAX JS object provided by browser

composing service URL

function performed when service call returns (callback)

https://github.com/jensdietrich/se-teaching/tree/main/ajax

Raw AJAX Example: Client (index.html)

```
 function updatePage() {
      if (xmlHttp.readyState == 4) {
         var responseData = xmlHttp.responseText;
            document.getElementById("matchingnames").value = responseData;
         }
      }
</script>

<form name="form1">Enter a name here:<br/>
 <input type="text" name="name" id="name" value="" size="50"
    onkeyup="check4matchingNames();"/>
 <textarea name="matchingnames" id="matchingnames" cols="40" row
</form>
```

update DOM (web page) element with data from service

DOM event triggering the service

https://github.com/jensdietrich/se-teaching/tree/main/ajax

Raw AJAX Example: Service (FirstNameService.java)

```java
public static final String[] FIRST_NAMES_DB = {"Joel","Michael","Rachel",..};
public void doGet(HttpServletRequest request, HttpServletResponse response){
     String n = request.getParameter("name");
     if (n==null) {
          response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
          return;
     }
     response.setContentType("text/plain");
     PrintWriter out = response.getWriter();
     String names = Arrays.stream(FIRST_NAMES_DB)
          .filter(name -> name.startsWith(n)).collect(Collectors.
     out.println(names);
     out.close();
    }
```

if name attribute is missing, return status code 400 - BAD REQUEST

return a whitespace-separated list of matching names

AJAX Summary

- HTTP is used to transfer data (not entire web pages)
- web servers can provide functionality as a service to be integrated into different web pages
- this is **technology-agnostic**, it does not matter how the server is implemented
- the logical next step is to use this approach outside web pages

AJAX and Web2.0

- Ajax was technology used to implement **Web2.0 features**
- term made popular by Tim O’Reilly in early 2000s
- user created content, web sites are **dynamic aggregation** of information from different sources (blogs, social media, ..) -- “mashups”
- technical focus shifts to interoperability of services

Jeff Bezos Memo (around 2002)

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- Anyone who doesn't do this will be fired. Thank you; have a nice day!

https://www.cio.com/article/3218667/have-you-had-your-bezos-moment-what-you-can-learn-from-amazon.html

Early Attempts: CORBA (from 1991)

- CORBA - common object request broken architecture
- standardised by OMG
- objects in a program are proxies for remote objects, potentially implemented in another PL
- object interfaces are described in a PL agnostic interface definition language (IDL), binary network protocol (IIOP)
- problems:
    - not firewall friendly (many firewalls by default prevent non-HTTP traffic)
    - poor performance of inital implementation, programming overhead
    - lack of support from Microsoft
    - complexity of using OO paradigm (managing distributed state, distributed GC)
- Protocol Buffers is a similar approach developed and used by Google

Early Attempts: SOAP + WSDL + UUDI

- XML-based protocol stack, can use HTTP
- SOAP - simple object access protocol - transport protocol to encode service invocations, developed by MS in late 90ties
- WSDL - web service description language to describe services
- UDDI - universal description, discovery and integration - yellow page - like service registry
- some initial success, then decline due to complexity
- Amazon offered APIs for both REST and SOAP, in 2006: 80% , 20% SOAP https://aws.amazon.com/blogs/aws/rest_vs_soap/
- UDDI services closed around 2010

RESTful Services

- modern HTTP services are often referred to as RESTful services
- REST refers to “representational state transfer ” a term coined in Roy Fielding’s PhD thesis in 2000 (note: Roy Fielding is Kiwi !)
- describes desirable principles of a web applications:
    - resources are identified by resource identifiers (URI), preferably with semantic (readable) URL: http://www.vuw.ac.nz/staff/jens
    - resources are manipulated by methods as defined in HTTP verbs: GET, POST, PATCH, ..
    - the resource representation (state) is transferred back to the client initiating the request
    - the resource stays on the server, the client will receive a representation of the resource

Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.

REST Principles

- **client-server architecture** facilitating the separation of concerns (e.g., UI on client, persistency on server)
- **statelessness** -- no client context is stored on the server
- **cachability** -- caching is actively used to increase scalability, responses are in control of cachability (e.g. with response headers)
- **uniform interface** -- resource can be identified by a universal scheme, URIs. Clients hold a representation of resources sufficient to modify or delete the resource. Clients can use hypermedia and metadata to gather more information as needed.

Encoding Data: Binary

- binary protocols: effective but issues around interoperability and trust
- examples:
  - IIOP (CORBA) - language-agnostic
  - Java binary serialization (RMI) - Java
  - protocol buffers (Google) - language-agnostic
  - ...

Encoding Data: XML

- XML: HTML-like markup language to describe a document tree (DOM), from mid 90ties
- standardised (W3C)
- protocol / technology stack:
    - schema to describe structure of a class of documents: DTD, XMLSchema, ..
    - XML-based transformation language (XSLT)
    - query language(s): XPath, XQuery
    - parsers for all major languages using different approaches (DOM, SAX, pull)
    - databinding to automatically translate objects graphs to/from XML, parser generators (JAXB)
    - magic databinding based on reflection and conventions: XMLDecoder/XMLEncoder in java.beans

Encoding Data: JSON

- uses JavaScript literal syntax to encode objects
- proposed by Douglas Crockford in the early 2000s
- later standardised as RFC 8259 and ECMA-404 in 2017
- became popular amongst web developers as it is:
    - compact
    - readable
    - easy to parse with JQuery (now all browser support this natively):
    - var p = JSON.parse(jsonstring);
- (2019) parsers exist for all major languages
- databinding frameworks such as Jackson

JSON Example

```
{
  "firstName": "Tim",
  "lastName": "Jones",
  "age": 42,
  "address": {
    "streetAddress": "1 Main Street",
    "city": "Wellington"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "04-12345"
    },
    {
      "type": "mobile",
      "number": "022-654321"
    }
}
```

encodes objects: key-value map like structure

nested objects

arrays of nested objects

JSON as JavaScript Literals

```
var tim = {
  "firstName": "Tim",
  "lastName": "Jones",
  "age": 42,
  "address": {
    "streetAddress": "1 Main Street",
    "city": "Wellington"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "04-12345"
    },
    {
      "type": "mobile",
      "number": "022-654321"
    }
}
```

JSON Limitations

- originally developed to be deliberately schemaless
- now JSON Schema languages have been developed – https://json-schema.org/
- this makes it possible to define:
  - valid attributes in a given context (a Person has attributes firstName, lastName, ..)
  - data types (firstName is a string, age a number, address an object of custom type Address, ..)
- no object references in “pure” JSON -- e.g., multiple persons cannot share *same* address
- *validators* can be used to check JSON against schemas, such as

  https://mvnrepository.com/artifact/com.networknt/json-schema-validator

Interacting with JSON in Java

- multiple JSON parsers: json.org, gson
- databinding frameworks: jackson
- Jackson / Gson uses reflection to automatically map objects to JSON (both ways)
- similar to **`java.beans.XMLDecoder`** / **`Encoder`** for XML
- this works extremely well for simple objects (flat structure, complying to bean model), processing is more complicated and must be customised for complex models with annotation or custom serializers

Jackson Databinding Example

- **`Order`** is a simple class with string and numeric properties, exposed with setters and getters
- convention of configuration in action (again)

```
import org.codehaus.jackson.map.ObjectMapper;
..
ObjectMapper mapper = new ObjectMapper();

Order order = .. ;
String json = mapper.writeValueAsString(order);

Order order2 = mapper.readValue(json, Order.class);
```

the mapper will try to parse the json string into an instance of the class passed as second argument

https://github.com/jensdietrich/se-teaching/tree/main/service-rest-frameworks

YAML

- **YAML** extends JSON (it is a superset of JSON)
- it has a comprehensive type system, references and comments
- it is popular to write configuration files (like CI configs like gitlabs **`.gitlab-ci.yml`**)

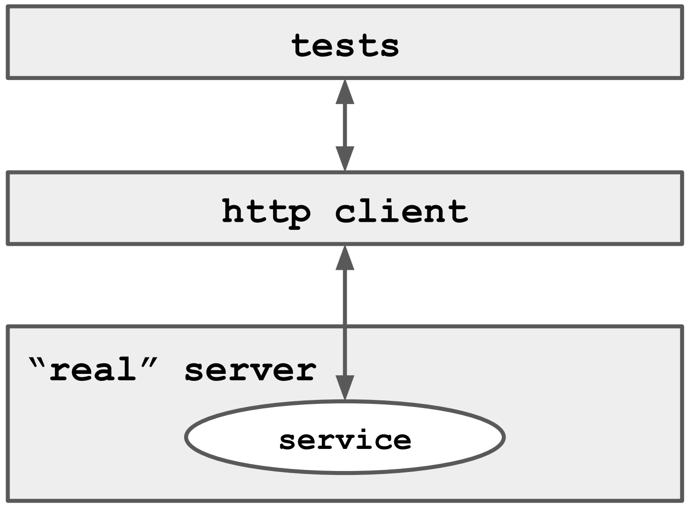
The Perils of Expressive Serialization

- versioning must be managed: if the data format changes, all parties need to use compatible versions
- encoding function calls is tempting, but exposes software to injection attacks
- deserialisation may automatically invoke functionality that can lead to vulnerabilities
- problem in Java deserialization:
  - https://speakerdeck.com/frohoff/appseccali-2015-marshalling-pickles-how-deserializing-objects-can-ruin-your-day
- but also common file formats like JSON: https://nvd.nist.gov/vuln/detail/CVE-2022-25845 , see also https://github.com/jensdietrich/xshady/tree/main/CVE-2022-25845

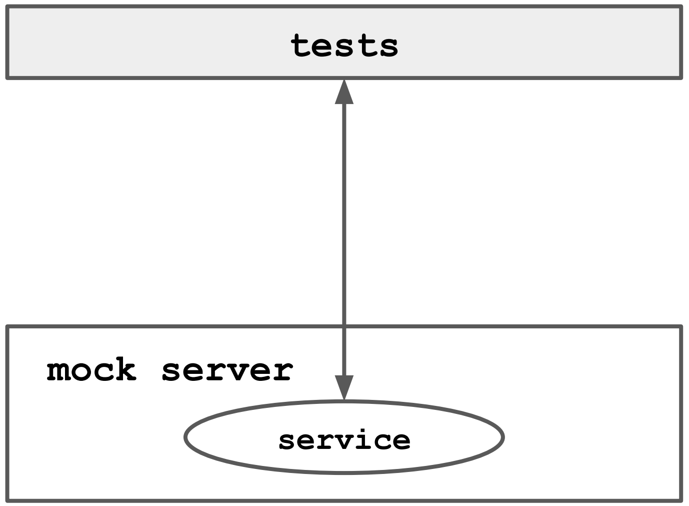
The Perils of Expressive Serialization (ctd)

- references may lead to algorithmic complexity vulnerabilities
- exhaustion of CPU, heap and stack memory during deserialisation
- can be exploited for denial-of-service attacks
- Java (de)serialisation:
  - https://en.wikipedia.org/wiki/Billion_laughs_attack
  - Dietrich, Jens, et al. "Evil pickles: DoS attacks based on object-graph engineering." (ECOOP 2017).
- other formats (XML-SVG, YAML, PDF)
  - Rasheed, Shawn et al. "Laughter in the wild: A study into dos vulnerabilities in YAML libraries." TrustCom'2019.
  - https://www.npmjs.com/advisories/788
  - many recent CVEs

Testing Services (and Web Applications)



blackbox testing: service or application is deployed in a real server, and a network (http) client is used by tests



whitebox testing: tests interact directly with the service (java) API, server functionality is "mocked" (faked) if necessary

Selenium: Automated In-Browser Testing

- for web applications there is another blackbox testing approach that utilises the web browser
- tests use a browser as HTTP client, and can also test client-side aspects of a web application (state of the DOM)
- example: **selenium**
- tests can be recorded and replayed
- through browser plugins, selenium scripts can be integrated with PL-specific testing frameworks

Blackbox Testing with Apache HTTP Client

- application is deployed on real server
- tests creates HTTP requests and send them to server
- assertions are made on the responses exposed as instance of **`HttpResponse`**
- examples (using the firstName service from the ajax example):
  - `assertEquals(200,response.getStatusLine().getStatusCode());`
  - `assertTrue(response.getFirstHeader("Content-Type").getValue().startsWith("text/plain"));`
- **`EntityUtils.toString()`** utility can be used to convert response body (entity) to string to analyse data returned
- example: https://github.com/jensdietrich/se-teaching/tree/main/service-blackboxtesting

Dealing with Server Dependencies when Testing

- blackbox tests rely on the server running
- this poses two challenges:

  1. start / stop the server in fixture
  2. reporting test results when server is not running

Managing Server Lifecycle in Test Fixture

```
@Before public void startServer() throws Exception {
    Runtime.getRuntime().exec("mvn jetty:run");
    Thread.sleep(1000);
}

@Before public void stopServer() throws Exception {
    Runtime.getRuntime().exec("mvn jetty:stop");
    Thread.sleep(1000);
}
```

- requires access to an API to start / stop server
- in example, the native API is used, servers starts/stops in background
- server process does not return (`waitFor` does not work) -- imperfect solution is to put in some sleep time to wait for the server to become available / unavailable
- may cause flakiness
- example: https://github.com/jensdietrich/se-teaching/tree/main/service-blackboxtesting

Managing Test results when Server is not Running

- what should be outcome of a test **when the server is not available**?
- should not be failed or success -- the outcome is unknown
- Junit offers assumptions for this purpose: (pre)conditions to be checked whether test is meaningful
- the semantics of violated assumptions may depend on the test runner used, most test runners flag those tests as **skipped**

Managing Server Lifecycle in Test Fixture

```
private boolean isServerReady() throws Exception {
     // try to connect to URL, return false if error occurs
     // or if response error code is returned by server
}
@Test public void test1() throws Exception {
     Assume.assumeTrue(isServerReady());
     // testing code
}
@Test public void test2() throws Exception {
     Assume.assumeTrue(isServerReady());
     // testing code
}
```

example: https://github.com/jensdietrich/se-teaching/tree/main/service-blackboxtesting

Whitebox Test

- in whitebox tests, the actual server is replaced by a fake “mock” server
- the mock server provides functionality provided by the server and required by the test
- mocking relies on abstraction used in server API: abstract types like **`javax.servlet.http.HttpServletRequest`** are provided by mock classes implementing the respective interfaces

Dynamic Mocking

- in dynamic mocking, interfaces are implemented “on-the-fly”
- dynamic proxies can be used for this
- rules are provided in the testing code to specify the behaviour of mocks
- example: mockito

the iterator interface is (magically) instantiated, i is an object of some class implementing Iterator

```
Iterator<String> i = Mockito.mock(Iterator.class);
when(i.next()).thenReturn("foo")
```

the mocked iterator captures calls to next(), and returns “foo” -- behaviour is implemented with simple rules

Static Mocking

- with static mocking, and abstract types are implemented by static classes
- instances of these classes can hold some state (e.g., the context type set in a response) and provide additional functionality to facilitate testing
- example: spring web mocks

Whitebox Testing Example

classes implementing the servlet API provided by static sprint web mock framework are directly instantiated

```
@Test
public void testInvalidRequestResponseCode1() throw    OException {
    MockHttpServletRequest request = new MockHttpServletRequest();
    MockHttpServletResponse response = new MockHttpServletResponse();
    // query parameter missing
    FirstNameService service = new FirstNameService();
    service.doGet(request,response);
    assertEquals(400,response.getStatus());
}
```

the servlet is directly instantiated and doGet is invoked

here we test for an expected error code as the request lacked the expected query parameter

example: https://github.com/jensdietrich/se-teaching/tree/main/service-whiteboxtesting

Whitebox Testing Example (ctd)

```
@Test public void testReturnedValues() throws IOException {
    MockHttpServletRequest request = new MockHttpServletRequest();
    request.setParameter("name","J");
    MockHttpServletResponse response = new MockHttpServletResponse();
    FirstNameService service = new FirstNameService();
    service.doGet(request,response);
    String result = response.getContentAsString();
    String[] names = result.split(" ");
    Set<String> set = Arrays.stream(names).collect(Collectors.toSet())
    assertTrue(set.contains("Joshua"));
    ..
}
```

the request is initialised with a query parameter

the mock implementati
ons provide additional methods to facilitate testing, e.g. to retrieve content

parse document into set of strings for tests

example: https://github.com/jensdietrich/se-teaching/tree/main/service-whiteboxtesting

RESTful Programming with Frameworks

- specialised frameworks have been developed to facilitate service programming
- common patterns:
    - language-agnostic (e.g. support multiple JVM languages -- Java, Kotlin, Scala, ..)
    - fluent APIs / use of builder design patterns and/or lambdas
    - build executables instead of server applications
    - integrated JSON databinding acknowledging JSON as the default data format
    - focus on low memory-footprint / fast startup time to quickly startup new instances on demand
    - exposes routing as central element: the mapping of URL/HTTP method (resource-verb) to functions
- the last trend reflects that the platforms for multi-tenancy have changed
- instead of having an application server shared between multiple applications, each application has its own (but often virtualised, docker etc) server

Case Study: Order Management

- use a simple `Order` domain model with string and numeric properties `id, product, amount, price`
- complies to Java bean model: public constructor without parameters, fields exposed by setters and getters
- this facilitates reflection and automated (de)serialisation from/to JSON
- `OrderDB` has CRUD like (simulated) persistence methods: `create, update, delete, getOne, getAll`
- CRUD - create / read / update / delete standard operations on persistent data
- https://github.com/jensdietrich/se-teaching/tree/main/service-rest-frameworks

Service Implementation with Javelin

- https://javalin.io/
- example for a modern framework that creates “executable services”
- Java and Kotlin APIs
- uses ***embedded*** Jetty
- direct support for common programming patterns like CRUD

Routing

```
public static void main(String[] args) {
    Javalin app = Javalin.create().start(8080);
    app.routes(() -> {
        crud("orders/:order-id", new OrderDB());
    });
}
```

“fluent” API

lambdas to configure core functionality

example: https://github.com/jensdietrich/se-teaching/tree/main/service-rest-frameworks

- **`main`** will start an embedded Jetty server
- Javelin will route requests identified by method / URL to a provided CRUD API **`OrderDB`**

The CRUD Interface

```
public class OrderDB implements CrudHandler {
     // simulate db
     public static Map<String, Order> ORDERS = new ConcurrentHashMap<>();

     @Override public void getAll(Context context) {
          Collection<Order> orders = ORDERS.values();
          context.json(orders);
     }
     @Override public void getOne(Context context,String resourceId) {
          Order order = ORDERS.get(resourceId);
          if (order==null) {context.status(SC_NOT_FOUND);}
          else {
               context.json(order); context.status(SC_OK);
          }
     } ..
```

integration into framework via implementation of interface

magic serialization of domain object(s) (Order) to/ from JSON via Jackson

CRUD Routing Table

OrderDB method	URL (42 - example id)	HTTP Method
`getOne(context,id)`	`localhost:8080/orders/42`	`GET`
`getAll(context)`	`localhost:8080/orders`	`GET`
`create(context)`	`localhost:8080/orders`	`POST`
`update(context,id)`	`localhost:8080/orders/42`	`PATCH`
`delete(context,id)`	`localhost:8080/orders/42`	`DELETE`

- Javelin will route requests identified by method / URL to a provided CRUD API **`OrderDB`**
- typical routing table for REST frameworks

Service Implementation with Jersey Client

- fluent APIs
- mapping to JSON also via Jackson, a bit more explicit

Order System Client

```
public static Order fetchById(String id) throws Exception {
    ClientResponse response = client
        .resource("http://localhost:8080")
        .path("orders").path(id).accept("application/json")
        .get(ClientResponse.class);
    if (response.getStatus()==404) {
        System.err.println("Order " + id + " not found");
        return null;
     }
    else {
        String s = response.getEntity(String.class);
        return mapper.readValue(s, Order.class);
    }
}
```

“fluent” API

magic deserialization of domain object(s) (Order) from JSON via Jackson

example: https://github.com/jensdietrich/se-teaching/tree/main/service-rest-frameworks

Specifying Services

- many organisations now offer Service APIs to integrate services into 3rd party applications
- example: github API https://developer.github.com/v3/
- APIs describe the structure of requests and responses, including:
  - service URLs and methods
  - request and response metadata (headers)
  - authentication schemes used
  - service restrictions (e.g., quota)
  - content encoding: content type
  - further constraints on content, described using a schema (e.g. names and types of child elements and attributes in a JSON document)

OpenAPI and Swagger

- the OpenAPI initiative aims at standardising API specifications, similar to IDLs (dedicated Interface Definition Language(s))
- the specification is available at https://github.com/OAI/OpenAPI-Specification , see also https://swagger.io/specification/
- the specification was originally developed as part of the Swagger product (https://swagger.io/)
- swagger offers tooling to develop, audit and share APIs (“SwaggerHub”)
- additional tooling to support the generation of language specific client stubs, tests etc from APIs: API-driven development

Swagger Example

https://app.swaggerhub.com/apis/jensdietrich/resthome4logs/1.3.0

- specifies an API to store logs
- **`POST /logs`** operation to store logs
- **`GET /logs`** operation to retrieve logs
- **`GET /stats`** operation to retrieve stats in spreadsheet format
- structure of JSON representation of log is defined by a schema
- specifies some conditions for error response codes